Université Ibn Tofail
Faculté des Sciences, Kénitra

Mémoire de Projet de Fin d'Etudes

Master Intelligence Artificielle et Réalité Virtuelle

AI-based Contracts Generation System

Établissement d'accueil : Laboratoire SETIME

Elaboré par : Mr LAYACHI Mohamed Amine

Encadré par : Mme BOUKIR Khaoula (ENSC Université Ibn Tofaïl)

Mr GUROV Dilian (KTH Royal Institute of Technology)

Soutenu le 25 juillet 2024, devant le jury composé de :

- Mr MESSOUSSI Rochdi (FSK Université Ibn Tofaïl)
- Mme BOUKIR Khaoula (ENSC Université Ibn Tofaïl)
- Mr BOUJIHA Tarik (ENSA Université Ibn Tofaïl)
- Mr NOURI Anass (FSK Université Ibn Tofaïl)

Année universitaire 2023/2024

Master thesis on the Application of Artificial Intelligence in Formal Methods

Ibn Tofail University

AI-based Contracts Generation System

Mohamed Amine Layachi

Supervisor: Khaoula Boukir, PhD, Ibn Tofail University

Co-Supervisor: Dilian Gurov, PhD, KTH Royal Institute of Technology

July 2024

Ibn Tofaïl University

Abstract

Software Verification is a crucial part of development cycles that ensures the robustness and correctness of software with respect to its specification. There are two fundamental approaches when it comes to verifying software: first off, there's static verification, which includes paradigms such as Model Checking, Static Analysis, and Formal Verification. Secondly, there's dynamic verification, which encompasses Unit Testing, Integration Testing, and many more. While the latter approach is more commonly used than the former, it doesn't guarantee correct software. To have a complete proof of correctness, we have to leverage the mathematical modeling involved in Formal Verification, which offers ways - Contracts - to ensure a complete verification pipeline that oversees all possible cases of the software. However, writing verification contracts is a tedious task that requires significant expertise and resources, but since AI has started its rapid progress and expansion, several questions have risen about its applicability in formal verification. This report introduces an AI-based contract generation system to generate ACSL specifications to verify programs. It achieves this by incorporating a Neurosymbolic architecture that synergizes state-of-the-art techniques in Generative AI with systematic approaches in Language and Compiler Designs, generating more complete specifications.

Keywords: Formal Methods, Neurosymbolic, Generative AI, Language Design, Compiler Design

Résumé

La vérification des logiciels est un élément crucial des cycles de développement qui garantit la robustesse des logiciels par rapport à leurs spécifications. Il existe deux approches fondamentales pour vérifier les logiciels : tout d'abord, il y a la vérification statique, qui comprend des paradigmes tels que la vérification des modèles, l'analyse statique et la vérification formelle. Ensuite, il y a la vérification dynamique, qui englobe les tests unitaires, les tests d'intégration et bien d'autres. Bien que cette dernière approche soit plus utilisée que la première, elle ne garantit pas qu'un logiciel est correct. Pour avoir une preuve complète des comportements corrects, nous devons tirer parti de la modélisation mathématique impliquée dans la vérification formelle, qui offre des moyens - les contrats - pour garantir un pipeline de vérification complet qui supervise tous les cas possibles du logiciel. Cependant, la rédaction de contrats de vérification est une tâche fastidieuse qui nécessite une expertise et des ressources importantes, mais depuis que l'IA a commencé sa progression et son expansion rapides, plusieurs questions se sont posées quant à son applicabilité à la vérification formelle. Ce rapport présente un système de génération de contrats basé sur l'IA pour générer des spécifications ACSL afin de vérifier des programmes. Il y parvient en incorporant une architecture neurosymbolique qui synergise les techniques de pointe en matière d'IA générative avec des approches systématiques en matière de conception de langage et de compilateur, générant des spécifications plus complètes.

Keywords: Vérification Formelle, Neurosymbolique, IA Générative, Conception de Langage, Compilateur

Dedication

I would like to dedicate this work to my grandmother, *Lakbira El Miraoui*, without whom my path over the past five years would have been uncertain.

To my beloved family, for your sacrifices, unwavering support, and encouragement of my curiosity.

To my friends, for the laughs we shared, the games we played, and the bond we created.

To all my dear ones, each and every one of you.

Thank you!

Acknowledgement

I would like to express my sincere gratitude to:

- My professor, Khaoula Boukir, for her unwavering support over the past two years and for creating the perfect environment for thoughtful inquiry.

- Mr. Dilian Gurov, for establishing a promising collaboration.

- My Master's professors, for their dedication to our class, ensuring a safe space and a high-quality education.

Contents

List of Figures

Chapter 1

Introduction

1.1 Overview

During the late 20th century, software engineering experienced rapid evolution that led to increasingly complex systems. As software grew more intricate, so did the challenges of ensuring its reliability, correctness, and security. High-profile failures, such as the Therac-25 radiation therapy machine accidents in the 1980s and the Ariane 5 rocket failure in 1996, underscored the potentially catastrophic consequences of software errors. These incidents highlighted the limitations of traditional testing and debugging methods, which often fail to detect subtle yet critical flaws. Formal methods emerged to solve these problems by providing a mathematical framework for specifying and verifying software behavior. Formal methods have multiple approaches for software verification, including but not limited to testing, simulation, and formal proofs, and each has distinct advantages and limitations. Testing is practical and straightforward, allowing for automated detection of real-world issues, but it cannot cover all possible inputs and states, leaving some bugs undetected. Simulation helps explore system behavior under varied conditions early in the design phase, but its accuracy is model-dependent and cannot guarantee correctness. Formal proofs provide the highest assurance of system correctness by rigorously verifying adherence to specifications, but they are complex, time-consuming, especially when verifying hard real-time systems that must be reliable at all times such as spaceships, airplanes, autonomous vehicles, etc. As the software code grows in size and complexity, it becomes clear that you can't rely solely on the developer to write correct software code. As such, many companies and organizations are hiring teams of formal methods experts to verify the written software. However, the AI "boom" that we all have seen these past few years has offered some of the most potent solutions to some of the most complex problems in the world, ranging from climate change modeling, protein folding, medical imaging, and famously, text understanding and generation. With the recent breakthroughs in sequence models such as Transformers [1], the industry is now able to overcome this challenge and open a wide variety of applications, one of which was showcased with the emergence of Large Language Models (LLM).

As emphasized, writing Formal Proofs can be a tedious process that requires significant expertise. However, some works have been proposed by the AI research community to automate this process using generative modeling to generate contracts for a given program's specification. Though these works assume an existing program's implementation, it is efficient to categorize them into two categories: the first category represents white-box approaches that partially take the code as an input, whereas the second one represents black-box approaches that do not rely on the software's implementation. As AI-based contract generation is a relatively new question in research, there are not many approaches that handle black-box frameworks, and to the best of our knowledge, the system introduced in this report is one of the first works that leverage generative AI to tackle black-box settings, and the first to combine it in a hybrid architecture with systematic methods to improve the correctness and completeness of the generated contracts.

1.2 Motivation and Objectives

Bringing user-friendly software verification tools to developers is a motivating goal for building this system. With this system, we aim to improve software reliability and safety, reducing development costs and setting a safety standard for all emerging applications. By achieving these objectives, reliable software will be more accessible to organizations and large or small companies alike. Furthermore, we envision this work laying a foundation for future intelligent systems to emerge and iteratively improve upon what this system offers.

1.3 Report Structure

This thesis is structured to provide a comprehensive exploration of AI-based Contract Generation. It begins with Chapter 1, the Introduction, establishing the context, motivation, and objectives. Chapter 2, Background, defines key concepts and offers a thorough literature review, setting the theoretical groundwork. Chapter 3, Contract Generation System, presents an overview of the system's architecture, discussing design principles and component interactions. Then, it focuses on the design of The Abstract Description Language, detailing its features and syntax. Then, the proceeding section explores the development of Pyramid, the conversion engine of this language. Sequential Reasoning Strategies describe a concept to mitigate the ambiguity and incompleteness of users' specifications, intuitions, and integration. Then, the ADL Generator's section focuses on building the generative model that translates sequential reasoning strategies to abstract descriptions in the ADL format, from data collection to fine-tuning. Next, Chapter 4, Results, presents snippets for each component, an evaluation of the ADL Generator, and a commentary on the system's performance. Finally, Chapter 5, Perspectives and Conclusion, offers reflections on the overall study and a conclusion to this report.

Chapter 2

Background

2.1 Insights into Formal Methods

- **Specifications: What a program is supposed to do?**

  Specifications are formal descriptions of a system's behavior and properties, translated from informal descriptions like diagrams, tables, or text into a well-defined language. They provide a concise, high-level overview and allow for formal reasoning and deductions about the system.

  Specifications can be categorized into three types: informal, formatted, and formal. Informal specifications use natural language, which can be ambiguous and disorganized, potentially leading to incompleteness, inconsistency, and misunderstandings. Throughout this report, we also refer to this category of specifications as descriptions. Formatted specifications employ standardized syntax that offers basic consistency and completeness checks but still allows for imprecise semantics, which can introduce errors. Formal specifications are rigorously defined in both syntax and semantics, often using mathematical precision to eliminate imprecision and ambiguity. While they provide a strong foundation for verifying the equivalence between specification and implementation, they can be difficult to read without specialized training due to their complexity and semantic distance.

- **What is a contract anyway?**

  The purpose of a function contract is to specify the expected properties of the input and, in return, the guaranteed properties of the output. The expected properties are known as the **precondition**, while the guaranteed properties of the output are referred to as the **postcondition**. This theoretical framework is practically applied using specification languages such as ACSL[2] or JML[3]. These languages extend the core language with features like logic constructs, connectors, primitive types, and built-in predicates to facilitate writing precise specifications.

- **ACSL**

  ACSL[2] (ANSI/ISO C Specification Language) is a formal specification language designed for use with C programs. It provides a way to specify the behavior and properties of C functions and programs using annotations written in comments within the source code. These annotations describe preconditions, postconditions, invariants, and other properties that the program should satisfy. ACSL is used to facilitate formal verification and static analysis, enabling the detection of errors, proving correctness, and ensuring that the implementation adheres to its specifications. In practice, ACSL introduces preconditions using the "**requires**" clauses, and postconditions using the "**ensures**" clauses. The following code snippet resembles ACSL contracts for a C program that returns the absolute value of an integer.

```c
/*@
    requires INT_MIN < val;
    ensures \result >= 0;
    ensures (val >= 0 ==> \result == val)
    && (val < 0 ==> \result == -val);
*/
int abs(int val){
    if(val < 0)
        return -val;
    return val;
}
```

2.2 Insights into Artificial Intelligence

- **Large Language Models**

  Large Language Models (LLMs) are advanced artificial intelligence systems designed to understand, generate, and manipulate human language. Trained on vast datasets containing diverse text, these models utilize deep learning techniques, particularly neural networks with many parameters, to capture the nuances of language. They can perform a variety of tasks, including text completion, translation, summarization, and answering questions, by leveraging their ability to model the statistical properties and contextual relationships within the language. LLMs have revolutionized natural language processing by achieving state-of-the-art performance in numerous language-related applications.

- **Data Augmentation**

  Data augmentation is a technique used in machine learning and artificial intelligence to increase the diversity of data available for training models without actually collecting new data. By applying a variety of transformations to existing data, such as rotations, translations, flips, scaling, cropping, and noise addition, it generates new data samples that maintain the original information content. This process helps improve the generalization ability of models, reduces overfitting, and enhances robustness by exposing the model to a broader range of variations during training. Data augmentation is particularly common in image processing but is also applied in other domains such as text, where techniques like synonym replacement, random insertion, random swap, and random deletion are used, as well as in audio and time-series data.

2.3 Literature Review

Within the first category of white-box approaches, much of the existing works mostly rely on pretrained LLMs to generate contracts for a given program's implementation, and then using engineered prompts or custom selection heuristics to refine those contracts. However, when it comes to the second category of black-box approaches, we haven't found any existing work that tackles this branch of contract generations. Thus, to the best of our knowledge, this work is the first to tackle black-box frameworks.

To present some of the literature we investigated: first, there is SpecGen [4], Lezhi Ma et al. (2024). In this paper, the authors decompose the task into two types of generations. The first phase employs a conversational approach in which a prompt is given to the LLM as few-shot examples, using ChatGPT [5] as an interface to the GPT-3.5-turbo LLM. They then leverage feedback from the specification verifier to provide more cues to the model, allowing it to generate better specifications. The second type is designed to tackle cases where the large language model fails to generate correct specifications. It follows a custom paradigm that the authors call mutation-based specification generation. In this paradigm, the verification-failed results generated by the LLM are tweaked using four kinds of mutation operators, constructing a set of all possible variants of the generated specification. Furthermore, once the latter set is constructed, a novel selection heuristic is used. SpecGen assigns a selection weight to each of these variants, allowing it to choose a subset of variants that is most likely to pass the verification test. The results of SpecGen are promising. The authors tested this approach on a benchmark dataset containing 120 Java programs against three baseline approaches: Pure-LLM, Daikon, and Houdini. SpecGen successfully generated JML specifications for 100 out of 120 programs, while Pure-LLM (conversation-based only), Daikon, and Houdini generated correct JML specifications for 72, 42, and 21 programs out of 120, respectively. This demonstrates the superior performance of SpecGen.

Secondly, there is a paper that shares some similarities with our approach, titled *Can ChatGPT support software verification?* [6], Christian Janßen et al. (2023), in which the authors tried to evaluate the ability of large language models, specifically ChatGPT (GPT-3.5), to generate invariants for programs. There is no prompt engineering behind this approach; they provide ChatGPT with a basic prompt like "Compute a loop invariant for the following program!" along with the C code snippet. However, they include a subtle indication or mask within the snippet to provide a hint to the LLM on where to place the specification. The results of this approach closely resemble the 'Naive' Pure-LLM approach that SpecGen was compared with. The authors demonstrate that ChatGPT can indeed generate valid loop invariants for given C programs. Empirically, ChatGPT generated 75 out of 106 valid loop invariants, which were validated using Frama-C[1] [7]. However, the authors also note that some loop invariants generated by ChatGPT were not validated by Frama-C due to technical reasons, but these results are still meaningful.

[1]Frama-C is a set of interoperable program analyzers for C programs developed by the French Commissariat à l'Énergie Atomique (CEA).

Chapter 3

Contracts Generation System

3.1 Architecture

In the related works we reviewed, while they use custom selection heuristics to validate contracts, they heavily rely on the generative model's capabilities to generate them. However, the system we designed, while also relying on generative models throughout its processes, implements a hybrid architecture that allows the generation of contracts to get distributed across multiple generative and systematic components. Additionally, it leverages generative models to refine contracts as much as possible, thereby increasing the chances for correct formal specifications.

This section examines how the system is built and provides a comprehensive overview of its components.

3.1.1 The Input

The contract generation process in this system begins with getting the user's description for the program they wish to verify as input, an informal specification expressed in natural language that describes what the program does. An example of such a specification could be as follows:

A program that returns the maximum of two numbers

3.1.2 Conserving the verbosity

While the description above may seem self-explanatory, it might not be valuable for a generative model to know how to deal with it, especially if the program is complex. Therefore, we need to inject some verbosity by introducing more linguistic semantics into the input, and this is what a Sequential Reasoning Strategy tries to do. It provides a step-by-step reasoning scheme to follow while generating contracts for the program and to build a sequential reasoning strategy for a program, the system prompts ChatGPT to use its capabilities to write one, as shown below.

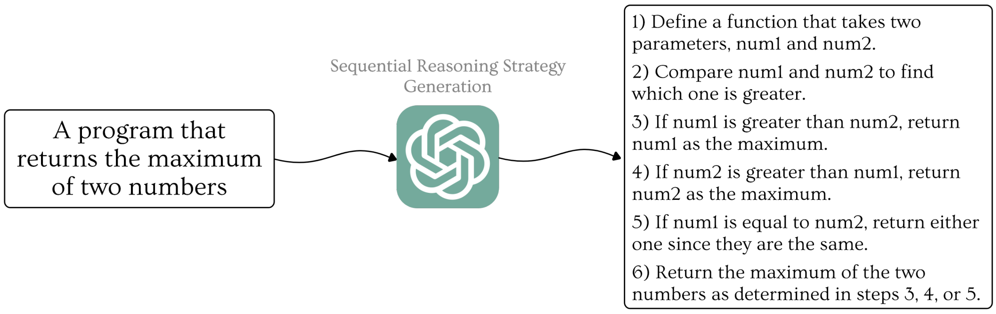

Figure 1: From description to Sequential Reasoning Strategy via ChatGPT

3.1.3 From one language to another

Right after the sequential reasoning strategy gets generated by ChatGPT, it will passed as input for a proprietary large language model, ADL Generator, that translates it from natural language to another language we've designed, called the Abstract Description Language (ADL).

ADL is a language that describes a less formal specification, such as a user's program description or sequential reasoning strategy, in a high level of abstraction with a formal syntax. The figure below illustrates this concept.

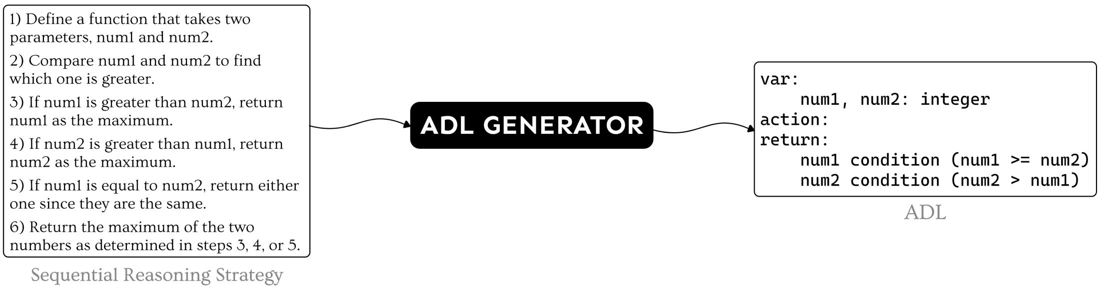

Figure 2: The ADL version of the sequential reasoning strategy

- **Syntax Correction**

  As impressive as LLMs show in myriad of applications, they still have flaws. As such, when the ADL generator produces an output, we can't know for sure that the output obeys the ADL syntax. Therefore, we rely on a bigger model, ChatGPT, in a conversational approach to correct any syntax errors given an ADL's template.

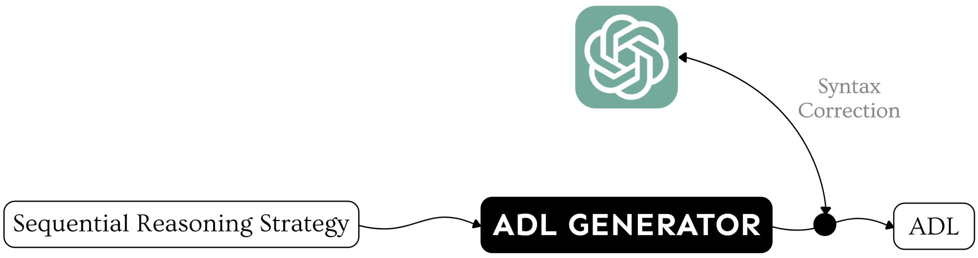

Figure 3: The Syntax Correction Loop

  In the case of the figure above, there are no syntax errors to fix in the ADL description. However, when there are syntactic errors, it will be helpful to correct them because otherwise, the description will not get processed by the conversion engine.

3.1.4 Contract Generation

The final step of this pipeline is contract generation, carried out by Pyramid, the conversion engine of ADL. Pyramid is a transpiler (source-to-source compiler) that takes in the ADL description generated by the ADL Generator and systematically converts it into ACSL contracts.

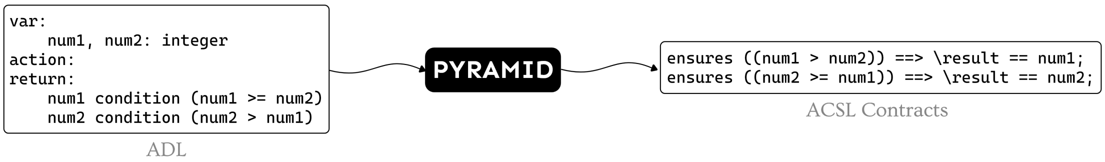

Figure 4: ACSL Contracts generated via Pyramid's translation of ADL

- **Contract Refinement**

  While Pyramid is a dynamic conversion engine capable of generating comprehensive and correct contracts, it still doesn't guarantee completeness. As such, we establish a contract refinement loop between Pyramid and ChatGPT, in which we guide the latter to generate other contracts that Pyramid might have missed or that are too complex for it.

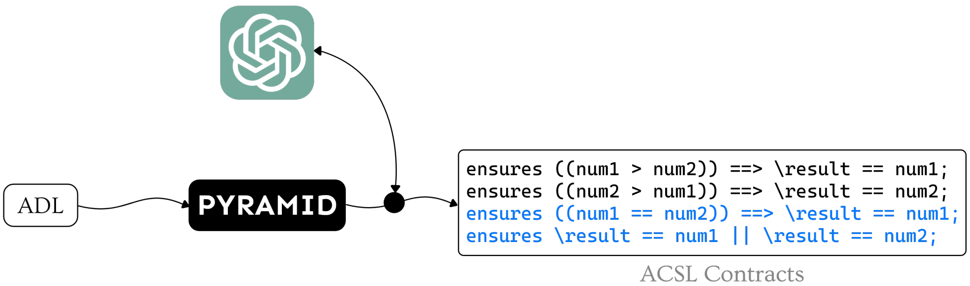

Figure 5: The Contract Refinement Loop

  As shown in the figure above, ChatGPT generate contracts based on whatever Pyramid has generated, and this is helpful because the refinement loop plays a role of guiding ChatGPT towards the correct contracts.

3.1.5 The Architecture's Assembly

The proposed contract generation system, utilizing a hybrid architecture that synergizes generative modeling with systematic processing, provides a glimpse at what an industry-scale verification software may look like, which will set a high standard for software safety.

In conclusion, with all the components defined, we are now poised to assemble the system's architecture.

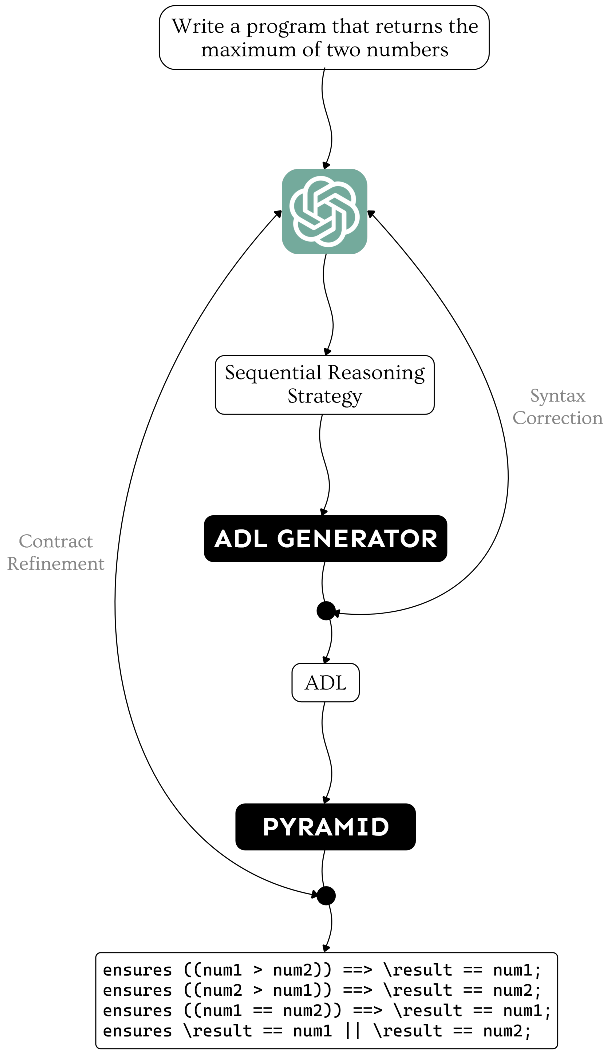

Figure 6: A view on the system's global architecture

The following sections will explore the various components of the architecture in greater detail, providing in-depth analysis and insights into their design, functionality, and integration. Each section will meticulously dissect the underlying principles, offering a comprehensive understanding of the architecture's intricacies.

3.2 The Abstract Description Language

There are multiple reasons why large language models might fail to capture the nuances that collectively make contracts correct and complete. First of all, given the environments in which software verification is conducted and its sensitive nature, there is a severe scarcity of data that these LLMs can train on, and this data resembles the code and its corresponding contracts. Moreover, the code's inherent verbosity tends to amplify the perplexity of these models for code analysis on a low level, which decreases the likelihood of generating correct and complete contracts.

The system proposed in this report differs from the existing works in multiple ways. First, instead of relying on the existence of an implementation for a program, it only needs a description - expressed in natural language - for what the program does. Then, using some concepts introduced in the later sections of this chapter, it generates the corresponding ACSL contracts for the given program. Secondly, whereas the existing approaches employ a large language model for contract generation, the system introduced in this report uses it differently; instead of having the contract generation process performed by a large language model, it is done systematically through a proprietary source-to-source compiler.

Establishing a direct conversion procedure from natural language to ACSL contracts is indeed a complex process, and it gets amplified by the restrictions highlighted above. However, an idea has provided some insights on how to mitigate them.

When a formal methods expert tries to write contracts for a program, they often start by questioning how the program should behave and not by delving into the implementation. This intuition implies a high level of abstract reasoning about the program because they know the behavior should be the same, whatever the code might be. Of course, some implementation-dependent contracts can only be determined if the code exists, but the core elements that define the program's behavior should produce consistent results.

This intuition has enabled the creation of a formal domain-specific language by which the system mimics the abstract reasoning necessary to write program contracts. This section introduces the Abstract Description Language, or ADL for short, offering a comprehensive look at how it is defined, the grammar that governs it, a study of its expressiveness, and some examples to improve the reader's understanding of the language.

3.2.1 Definition

The Abstract Description Language (ADL) is a formal language designed to formalize the description of a program, expressed in natural language, by retaining a high level of abstraction that eliminates the need for an actual implementation of the program. In other words, when writing an abstract description in ADL, they only need to express how the program should behave regardless of its implementation.

When we say contract generation, we mean generating preconditions and postconditions. Therefore, an abstract description should be constrained to the inputs and outputs of the program.

3.2.1.1 Structure

The ADL's structure decomposes the behavior of the whole program into three sub-classes of behaviors:

1. **var**, encapsulates all the variables necessary for the program to behave as it should.
2. **action**, contains all the assignment operations that change the memory state of one or more variables declared in the **var** block.
3. **return**, consists of all the returns or outputs of the program.

As the action and return blocks might be conditional, ADL also implements a way to control these behaviors, that is by using a condition that indicates that a particular assignment (*resp.* output) is executed (*resp.* returned) only if it satisfies a specified condition, or else, satisfying any condition.

Formally, an abstract description δ is an ordered set containing all the three behavior sub-classes of the program such that,

$$\delta = \{\mathbb{V}, \mathbb{A}, \mathbb{R}\}$$

Where $\mathbb{V}$ denotes the set of variables, $\mathbb{A}$ denotes the set of actions, and $\mathbb{R}$ denotes the set of returns. These sets are also formally defined as follows:

$$\forall i \in \mathbb{N} \quad \mathbb{V} = \textstyle\bigcup_i \{v_i\}$$

$$\forall i \in \mathbb{N} \quad \exists \kappa \in \mathbb{K} \quad \mathbb{A} = \textstyle\bigcup_i \{\alpha_i, \kappa_i\}$$

$$\forall i \in \mathbb{N} \quad \exists \kappa \in \mathbb{K} \quad \mathbb{R} = \textstyle\bigcup_i \{\rho_i, \kappa_i\}$$

Where $\mathbb{K}$ is the set of conditions.

3.2.2 Grammar

The Abstract Description Language follows a well-defined grammar that dictates how it should be written. Additionally, as this language was designed to reduce the verbosity of programs, it enjoys a simple syntax that leads to quick learning. The following list contains all the grammar for ADL.

- **Variable Declaration**

  $\langle \textit{var} \rangle$ ::= $(\langle \textit{identifier} \rangle$ `,`$)^*$ $\langle \textit{identifier} \rangle$: $\langle \textit{type} \rangle$

- **Actions Statements**

  $\langle \textit{action} \rangle$ ::= $\langle \textit{assignment} \rangle$ `condition` $\langle \textit{condition} \rangle$ | ϵ

- **Return Statements**

  $\langle \textit{return} \rangle$ ::= $\langle \textit{output} \rangle$ `condition` $\langle \textit{condition} \rangle$ | ϵ

- **Conditions**

$\langle \mathit{condition} \rangle ::= (\langle \mathit{comparison} \rangle\ \langle \mathit{andor} \rangle)^{*}\ \langle \mathit{comparison} \rangle$

$\mid\ \texttt{ANY}$

$\mid\ \texttt{!}\ \langle \mathit{condition} \rangle$

$\mid\ \epsilon$

- **Types**

$\langle \mathit{type} \rangle ::= \texttt{integer} \mid \texttt{pointer} \mid \texttt{real}$

- **Assignment**

$\langle \mathit{assignment} \rangle ::= \langle \mathit{identifier} \rangle\ \texttt{=}\ \langle \mathit{parameters} \rangle$

$\mid\ \langle \mathit{identifier} \rangle\ \texttt{=}\ \langle \mathit{boolean} \rangle$

- **Outputs**

$\langle \mathit{output} \rangle ::= \langle \mathit{expression} \rangle$

$\mid\ \langle \mathit{boolean} \rangle$

$\mid\ \texttt{None}$

- **Comparisons**

$\langle \mathit{comparison} \rangle ::= \texttt{(}\ \langle \mathit{parameters} \rangle\ \langle \mathit{comp\text{-}op} \rangle\ \langle \mathit{parameters} \rangle\ \texttt{)}$

$\mid\ \texttt{(}\ \langle \mathit{expression} \rangle\ \texttt{is}\ \langle \mathit{boolean} \rangle\ \texttt{)}$

$\mid\ \texttt{!}\ \langle \mathit{comparison} \rangle$

$\langle \mathit{comp\text{-}op} \rangle ::= \texttt{==} \mid \texttt{!=} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{>} \mid \texttt{<}$

- **Logical Operators**

$\langle \mathit{andor} \rangle ::= \texttt{and} \mid \texttt{or}$

- **Parameters**

$\langle \mathit{parameters} \rangle ::= \langle \mathit{identifier} \rangle \mid \langle \mathit{expression} \rangle \mid \langle \mathit{function} \rangle$

- **Functions**

$\langle \mathit{function} \rangle ::= \texttt{(}\ ((\langle \mathit{expression} \rangle\ ,)^{*}\ \langle \mathit{expression} \rangle)?\ \texttt{)}$

- **Boolean Values**

  $\langle \mathit{boolean} \rangle$ ::= `true` | `false`

- **Expressions**

  $\langle \mathit{expression} \rangle$ ::= $\langle \mathit{term} \rangle$
  | $\langle \mathit{expression} \rangle$ `+` $\langle \mathit{term} \rangle$
  | $\langle \mathit{expression} \rangle$ `-` $\langle \mathit{term} \rangle$
  | $\langle \mathit{expression} \rangle$ `|` $\langle \mathit{term} \rangle$

- **Terms**

  $\langle \mathit{term} \rangle$ ::= $\langle \mathit{factor} \rangle$
  | $\langle \mathit{expression} \rangle$ `*` $\langle \mathit{factor} \rangle$
  | $\langle \mathit{expression} \rangle$ `/` $\langle \mathit{factor} \rangle$
  | $\langle \mathit{expression} \rangle$ `%` $\langle \mathit{factor} \rangle$
  | $\langle \mathit{expression} \rangle$ `&` $\langle \mathit{factor} \rangle$
  | $\langle \mathit{expression} \rangle$ `^` $\langle \mathit{factor} \rangle$
  | $\langle \mathit{expression} \rangle$ `>>` $\langle \mathit{factor} \rangle$
  | $\langle \mathit{expression} \rangle$ `<<` $\langle \mathit{factor} \rangle$

- **Factors**

  $\langle \mathit{factor} \rangle$ ::= $\langle \mathit{element} \rangle$
  | `-` $\langle \mathit{factor} \rangle$
  | `+` $\langle \mathit{factor} \rangle$
  | `~` $\langle \mathit{factor} \rangle$

- **Elements**

  $\langle \mathit{atom} \rangle$ ::= $\langle \mathit{number} \rangle$
  | $\langle \mathit{identifier} \rangle$
  | $\langle \mathit{function} \rangle$
  | `(` $\langle \mathit{expression} \rangle$ `)`

- **Identifiers**

  $\langle \mathit{identifier} \rangle$ ::= `id`

3.2.3 Example

To clear things up a little bit, let's consider the following example and approach it step by step:

Write an abstract description in ADL that describes the behavior of a program that takes two pointers and swap the values of the memory locations they point by to.

1. The program requires two variables of a pointer type.

   To declare variables in ADL, you need the name and type of the variable, then you declare them as follows:

   ```
   x: pointer
   y: pointer
   ```

   Alternatively, there is a shortcut in ADL that allows variables of same type to be declared in one line, separated by commas.

   ```
   x, y: pointer
   ```

2. The program performs two assignment operations. A valuable intuition a user can leverage when writing ADL descriptions is if the behavior is described directly in a literal manner, the user can write it exactly as described instead of writing the implementation that leads to that behavior. For instance, in this example, swapping two numbers implies that by the end of the program, the first memory location must have the second value. Similarly, the second memory location must have the value of the first. As such, there is no need to use intermediate variables or arithmetic operations to express this behavior. Therefore, we can express this behavior by writing $x = y$ and $y = x$. It's worth noting that the pointer type in ADL doesn't work the same way as in the C or C++ programming languages. In ADL, there are neither dereferencing nor arrow operators to manipulate pointers. However, the conversion engine of this language handles them differently than the other types.

3. Given that the program relies on pointer variables to swap those values, there should be no need to return anything as an output since the swap operation will be done in place. In this case, the Abstract Description Language introduces a keyword that indicates that a particular program doesn't return anything, which is defined as ***None***.

4. The program's description doesn't enforce any conditions that control the execution of any action or return. Therefore, these statements execute under any condition. Thus, ADL introduces a keyword that resembles this case, which is defined as **ANY**.

   There is a specific structure one must follow to write an ADL description. First, it must have a section where all the variables are declared. The variable declaration section always starts with the macro "var:" and for this example, the abstract description is written as follows:

```
var:
    x, y: pointer
```

The variable declaration section always starts with the macro "var:".

Secondly, it must have a section containing all the actions of the program and the conditions that control its execution. The actions section always starts with the macro "action:", and for this example, it is written as follows:

```
action:
    x = y condition ANY
    y = x condition ANY
```

Lastly, an abstract description must have a section containing all the return statements of the program and the conditions that control their return. The returns section always starts with the macro "return:", and for this example, it is written as follows:

```
return:
    None condition ANY
```

All the analysis and steps we went through above culminated in a concise ADL description, and it is written as follows:

```
var:
    x, y: pointer
action:
    x = y condition ANY
    y = x condition ANY
return:
    None condition ANY
```

3.2.4 A Study of Expressiveness

While the Abstract Description Language expresses a program in a high level of abstraction, it cannot express programs whose behavior is determined by repetition because recursions and loops are not allowed in ADL. Therefore, it's not a Turing-complete language.

Lemma 3.1. *Let $\mathcal{T}$ be a function that maps an ADL description to a C program, such that:*

$$\mathcal{T}\colon \mathcal{P}(A) \to \mathcal{P}(C)$$

$\mathcal{T}$ is not a surjective function.

Proof. Suppose $\mathcal{T}$ is a surjective function. Then,

$$\forall x \in \mathcal{P}(C) \quad \exists y \in \mathcal{P}(A) \quad s.t \quad \mathcal{T}(x) = y$$

However, given that ADL is not a Turing-complete language. Then, there must be a C program whose description in ADL doesn't exist, which contradicts the definition.

Therefore, $\mathcal{T}$ is not surjective. $\square$

Corollary 3.1.1. *Let E be a function that measures the expressiveness of a language to describe programs. Then,*

$$E(\mathcal{A}) < E(\mathcal{C})$$

Where $\mathcal{A}$ denotes the Abstract Description Language (ADL) and $\mathcal{C}$ denotes the C Programming Language.

Proof. From *Lemma 4.1* we have $\mathcal{T}$ a non-surjective application, which implies that it's not bijective either. Thus, $|\mathcal{P}(\mathcal{A})| \neq |\mathcal{P}(\mathcal{C})|$.

Given that ADL is not a Turing-complete language while C is. Then, $|\mathcal{P}(\mathcal{A})| < |\mathcal{P}(\mathcal{C})|$.

Therefore,

$$E(\mathcal{A}) < E(\mathcal{C})$$

□

Although the expressiveness of ADL is much smaller than C, there's a good reason why it should be. In a paper called "On the Expressive Power of Programming Languages," [8] the author, Matthias Felleisen, proved (*Theorem 2.9*) that an increase of expressive power may destroy the semantic properties of the core language, and this implies that if the expressiveness of ADL increases, it will lose its abstraction quality, which is what ADL was developed to have.

3.3 Pyramid: The Conversion Engine

Despite having defined a vital part of this system, that is the Abstract Description Language. Its ultimate benefit and premise necessitate the creation of a conversion engine to convert an abstract description of a well-defined function or program into ACSL contracts. As reiterated throughout the preceding sections, this type of conversion is a sophisticated process. However, this section introduces a contract generator called Pyramid, for which several ideas have been developed to mitigate the complexity of constructing such a conversion engine.

In the compiler design field, a "compiler" is an umbrella term that describes a tool that takes in a source code written in one language and translates it into another language. In practice, the output of this tool is often machine code, for instance: **gcc**, which takes C source code and produces machine code. Though we are not interested in translating into machine code, we are actually interested in translating into another high-level language. Fortunately, a "transpiler" is a term that describes such a tool. It is a subset of compilers that translate from one source to another, and they are, in fact, often called source-to-source compilers. Within the context of the conversion engine, Pyramid builds a basis from this category of compilers, such that it constructs a dynamic pipeline that generates ACSL contracts from an ADL description.

3.3.1 Parser

When building any compiling process, it always starts by doing a syntactic analysis for the source language and parses it to get every token that obeys the grammar that defines the language. Often, every sophisticated compiler implements a parsing procedure developed from scratch to optimize the program's performance most efficiently. However, in our case, there was no need for a heavily customized parser since the output is not something a machine can run. Therefore, Pyramid uses a third-party tool called Lark to generate the parse tree from the ADL grammar. Then, when an abstract description gets parsed, Pyramid employs a series of procedures to analyze the description and builds a structure of information about it, which then gets used by intermediate pipelines to generate the description's corresponding ACSL contracts. The following figure is a visual explanation of how the parser is constructed.

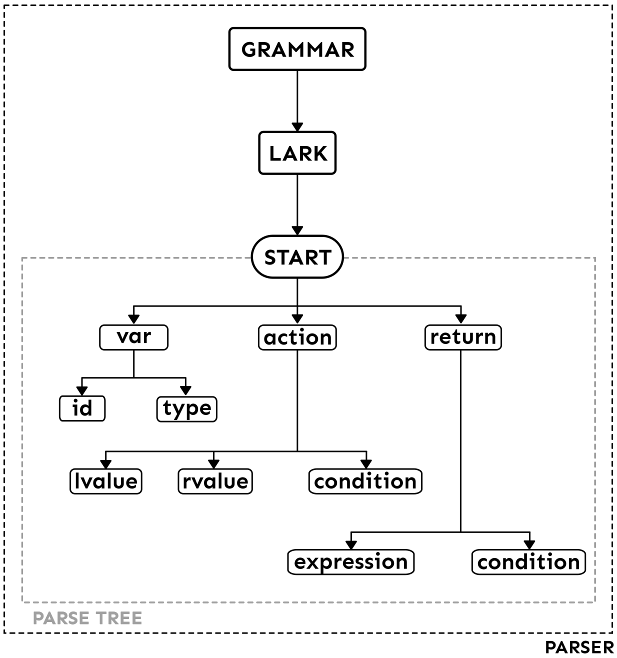

Figure 7: Diagram of the ADL's parser showing the construction of the parse tree.

3.3.2 Knowledge-Sharing Mechanism

When Pyramid starts to generate contracts, it uses the three $\{var, action, return\}$ blocks of an abstract description to get every possible contract of each block individually. However, if we consider each block as an independent node, we will definitely not cover most of the necessary contracts, and that's because some contracts can only be generated if there is some sort of connection between blocks, and this is what the Knowledge-Sharing Mechanism tries to solve. Before providing a formal definition for what the Knowledge-Sharing Mechanism actually is, it would be helpful to start with an intuition that hopefully will help illustrate the idea.

Imagine that you, the reader, are practicing an exercise in mathematics. You continue solving questions until you stumble on a question that requires you to prove some result. The idea here is that if you have solved the questions sequentially, you will be able to use whatever results you proved to solve this question.

Inspired by the intuition above, we can define the Knowledge-Sharing Mechanism (KSM) as a way of that allows sharing information between nodes in a complete a graph constructed using the three ADL blocks $\{var, action, return\}$. Practically, Pyramid implements procedures to get each block to communicate with another by leveraging the edges between all nodes, also called Knowledge-Sharing Channels. Each channel dictates how two nodes communicate. For instance, the communication between *var* and *action* revolves around checking for overflows, memory separation, pointer validity, and many more.

3.3.3 Design

After defining a core concept for the generation process, it makes sense to introduce the design behind the conversion engine. First, Pyramid takes in an ADL description and parses it using the parse tree constructed using Lark. Then, the parsed syntax gets transformed using intermediate pipelines to get all the $\{var\}$, $\{action\}$, and $\{return\}$ objects in the abstract description, which have built-in methods to self-convert into ACSL contracts. Next, all objects share information appropriately using three Knowledge-Sharing Channels. Finally, all the generated contracts get passed to the Contract Builder, which performs a clean build by eliminating any redundancy and any inherent issues of the pipeline. The following diagram illustrates how Pyramid works.

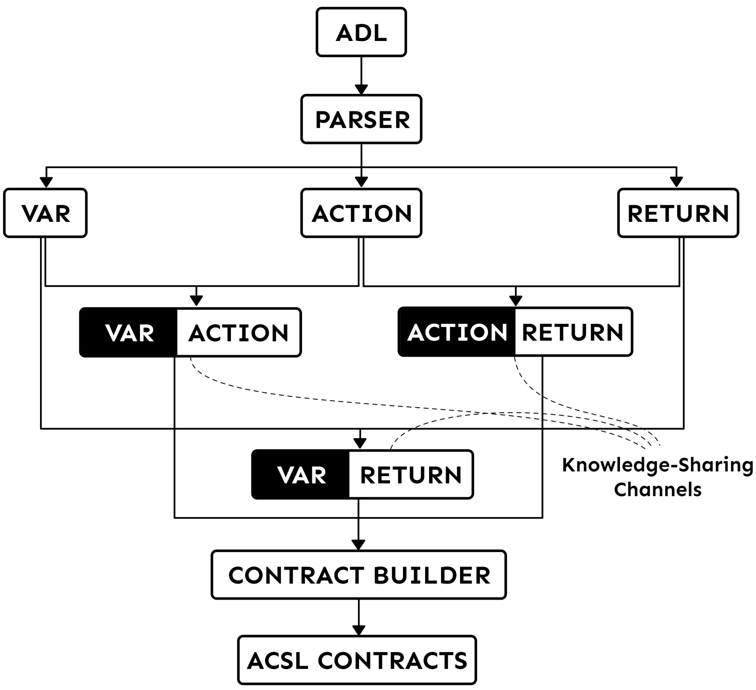

Figure 8: The Architecture of Pyramid

3.3.4 Implementation

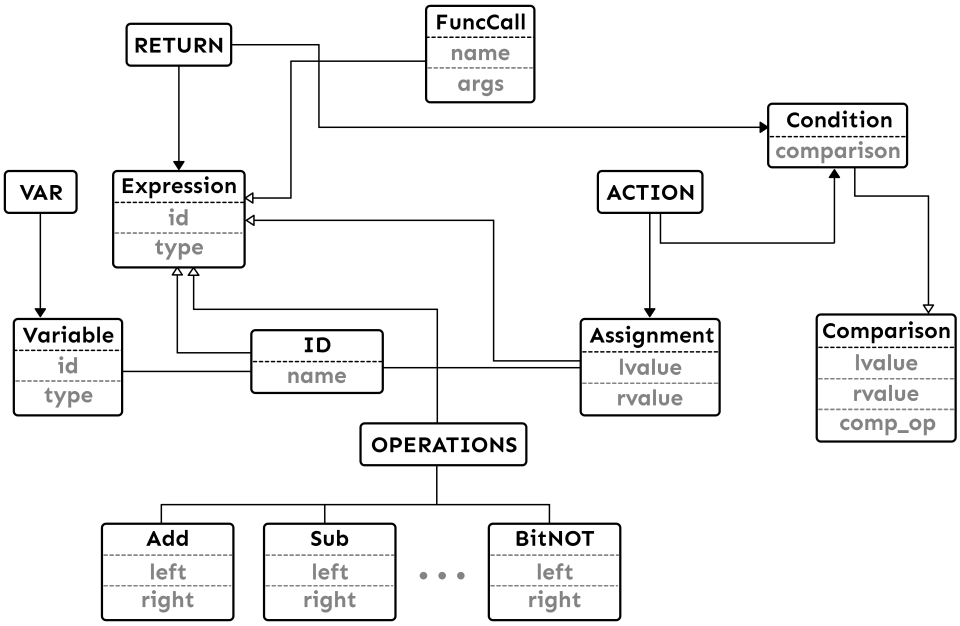

Figure 9: A class diagram explaining the OOP Design of Pyramid

As far as the implementation of transpilers goes, it is usually a sophisticated one, and Pyramid is no exception. The latter is implemented in Python with an OOP design that facilitates code management. The central class of this design is the Expression class since it covers the construction of most things, ranging from IDs, Assignments, Operations, and many more. Additionally, each object in Pyramid has a method to reconstruct the original information.

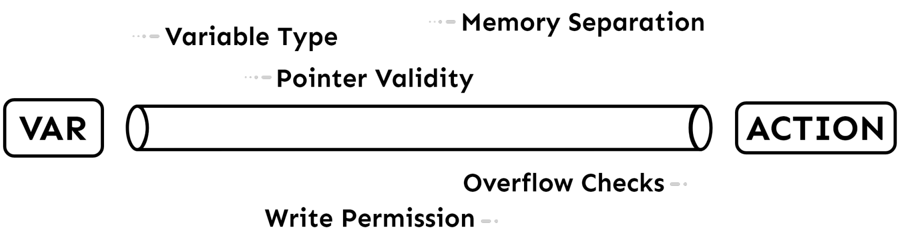

Figure 10: Representation for the Knowledge-Sharing Channel between *var* and *action*

As for the Knowledge-Sharing, the figure above shows how the Var and Action blocks communicate via a channel that takes care of several things, such as Pointer Validity, Memory Separation, and related concepts for generating more complete contracts later.

Finally, after all the knowledge is shared and contracts get generated, they go for the final step: Contract Building. This process eliminates any redundancy and inconsistency of contracts to maintain a logically correct set of contracts to establish a complete verification.

3.3.5 Limitations

While Pyramid highlights a glimpse towards a systematically dynamic contract generation process, it's still limited to what is possible to express with the Abstract Description Language. The following list is all the verification constructs that Pyramid can generate.

- Preconditions (**requires**)
  - Overflow Checks
  - Pointer Validity, as well as the read-validity checks
- Postconditions (**ensures**)
- Predicates
  - Memory Separation
  - Assignments
  - Old Values

3.3.6 Beyond Limitations: Contract Refinement

Relying on systematic methods to generate contracts is an effective solution to achieve predictability and correctness, but it doesn't guarantee completeness. However, it can be used as an intermediate way to achieve it. Large language models, given how much data they are trained on, are great for coming up with conclusive answers given the evidence. In this case, the evidence is whatever contracts Pyramid generates. The probabilistic nature of LLMs enables them to be guided when evidence is present, allowing them to deduce things far more complex than the evidence itself. As such, our idea is to establish a refinement loop to improve the completeness of contracts and to handle edge cases that might not be possible to generate with Pyramid. In this refinement loop, Pyramid interacts with ChatGPT, which takes in whatever contracts the former has generated under a prompt template that gives the sequential reasoning strategy and asks ChatGPT to improve the results. Consequently, the latter tries to analyze and reason about the program and hopefully will generate more complete contracts.

3.4 Sequential Reasoning Strategies

One of the difficulties we face in formal methods is mapping a program's description into an implementation that satisfies the correctness metric in the verification process. This difficulty is non-reversible because finding an implementation doesn't imply its correctness, and what's more challenging is that it's a procedure that involves strict logical reasoning about the implications of each aspect of the description.

To understand the need for the concept introduced in this section, we must emphasize reasoning as a conceptual practice that algorithmically guides machines to map correspondences between descriptions stated in natural language and their ADL versions. When given a program's statement, we always try to reason on the crucial parts by analyzing the statement and constructing a reasoning scheme to understand the program's inner workings in a high level of abstraction, which guarantees the expected behavior of the program without delving into the implementation details.

This section introduces Sequential Reasoning Strategies, an intermediate step toward reaching a fully autonomous contract generation system.

3.4.1 Intuition

Before providing a formal definition for Sequential Reasoning Strategies, it would be helpful to start with an intuition that hopefully will help illustrate the idea.

Let's recall the program's description used in the section on the Abstract Description Language.

> *Write an abstract description in ADL that describes the behavior of a program that takes two pointers and swap the values of the memory locations they point by to.*

Generating contracts just from a simple description, such as the one above, is a complex task to be solved in one piece because the pipeline requires a level of verbosity that allows the consideration of crucial information. When the above description is presented to a formal methods expert, they can easily write contracts for the description's corresponding program. However, the task becomes significantly more challenging when assigned to an AI model. In fact, it is quite daunting even for humans, as we cannot assume everyone is proficient in Mathematics and Computer Science. Therefore, efficiency in this task requires addressing multiple layers of complexity one at a time, and one solution to the first layer is Sequential Reasoning Strategies.

This concept will guide the AI model to learn how to reason with the given description of a program. Subsequently, it will learn to generate correct ADL descriptions that inherit a high level of abstraction and formality while including an appropriate level of verbosity.

3.4.2 Definition

A *Sequential Reasoning Strategy* ($\mathcal{R}_\mathcal{S}$) is an algorithmic representation - expressed in natural language - of a program's description that mimics the reasoning process involved in implementing the corresponding program sequentially through steps s, such that:

$$\mathcal{R}_{\mathcal{S}} = \bigcup_{s} \{s\}$$

The reason why it is expressed in natural language is to avoid the injection of code in the input of the large language model used to translate Sequential Reasoning Strategies into ADL descriptions, which will increase the level of bias in those models, thereby generating incorrect translations. Furthermore, there's no simpler input for the LLM we use than natural language because even pseudo-code is more complex than natural language. Also, given the way LLMs are trained, the amount of text in the dataset is way bigger than the amount of code. Therefore, it's much more efficient to use that to our advantage.

3.4.3 Example

To clear things up, consider the following example of a sequential reasoning strategy that corresponds to the description of the program above.

s_1 *Define a function that takes two numbers, 'num1' and 'num2', as input.*

s_2 *Add 'num1' and 'num2' together and store the result in 'num1'.*

s_3 *Subtract 'num2' from the updated 'num1' and store the result in 'num2'.*

s_4 *Subtract the original value of 'num2' (now stored in 'num1') from the updated 'num1' and store the result in 'num1'.*

s_5 *Return the updated values of 'num1' and 'num2'.*

s_6 *Return the updated values of 'num1' and 'num2'.*

This strategy breaks down the reasoning process into simple steps that specify what's relevant but preserve the abstraction. This is important because it helps the model develop a chain of thought in the data that our ADL Generator [1] will train on, improving its performance.

[1] A large language model, fine-tuned to translate a given sequential reasoning strategy to ADL

3.4.4 Integration

When the user wishes to generate contracts for a program, they need to describe what the program must do. Our system takes in the description provided by the user as input and prompts ChatGPT to generate a sequential reasoning strategy for it. Then, the pipeline continues by using the generated strategy as an input of our ADL generator, which translates the strategy from natural language to ADL.

The following figure shows how Sequential Reasoning Strategies are integrated in the system.

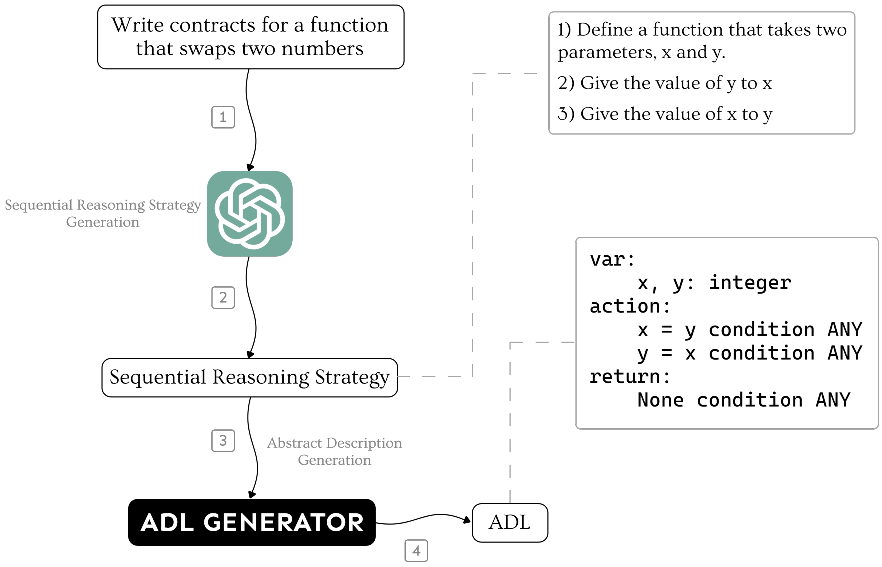

Figure 11: A diagram that explains the integration of Sequential Reasoning Strategies in the system.

As a final note for this section, since users' descriptions for a program are usually abstract, Sequential Reasoning Strategies serve as a conservation of verbosity to reduce the complexity required for the ADL generator to translate the given description to ADL. The next section delves into what goes into the ADL Generator with more detail.

3.5 ADL Generator

Training an AI, especially a large language model, is not easy because it is a resource-intensive task. It requires a massive amount of data and a large amount of compute in order to achieve a good performance. In our case, we had none of these resources. First, we have defined an entirely new language - ADL - for a model to learn, hindering our ability to have a large amount of data to train the model. Secondly, training an LLM requires a cluster of Graphical Processing Units (GPUs) to speed up the training, something we don't have. These restrictions have significantly impacted the approach used to develop the model.

The ADL Generator is a large language model developed to translate a sequential reasoning strategy to its corresponding ADL description. It happens prior to the conversion engine, Pyramid, and after the sequential reasoning strategy generator, ChatGPT. In this section, we will do a deep dive into the inner workings of the ADL Generator model, from data collection to evaluation.

3.5.1 Data Collection

In modern deep learning architectures, the availability of high-quality data is necessary to develop a highly-performing model. As large language models are becoming more data-intensive, the task of training an LLM is restricted by the ability to collect massive amounts of data, which is questionable the smaller the team responsible for data collection is. This was the precise challenge we encountered when we began developing the ADL Generator. The latter required a lot of data to be fine-tuned and achieve the performance that we expected it to have, and that was not possible in such a small team and period of time.

However, this process alleviates when we set to fine-tune the LLM instead of training it. As the number of parameters in the model increases, the amount of compute required to train the model will be huge because when training a model, it optimizes every one of its parameters. Whereas, fine-tuning, in practice, will only train the parameters of the last layer of the model, thereby reducing the number of trainable parameters and the number of GPUs required to train the model. Additionally, when a model is fine-tuning, it will usually need less data than to train it, which is exactly what we want.

3.5.1.1 Inputs and Outputs

To fine-tune a model, one needs to specify the input and output of the model. In the context of the ADL Generator, as the name suggests, it generates an abstract description in ADL as an output, and takes in a sequential reasoning strategy as input. As such, the data collection process consists of writing sequential reasoning strategies and their corresponding ADL translations.

However, given that each sequential reasoning strategy consists of steps, it's more efficient to label the ADL translation of each step as well; this way, the ADL generator will have more diverse data to train on and will help develop a sense of reasoning when it tries to generate an ADL description, thus improving its accuracy.

Strategy	ADL

1) Define a function that takes two parameters, num1 and num2.
2) Compare num1 and num2 to find which one is greater.
3) If num1 is greater than num2, return num1 as the maximum.
4) If num2 is greater than num1, return num2 as the maximum.
5) If num1 is equal to num2, return either one since they are the same.
6) Return the maximum of the two numbers as determined in steps 3, 4, or 5.
	var:
	    num1, num2: integer
	action:
	return:
	    num1 condition (num1 >= num2)
	    num2 condition (num2 > num1)

Figure 12: Strategy-based Labeling

Steps	ADL
1) Define a function that takes two parameters, num1 and num2.	var: num1, num2: integer
2) Compare num1 and num2 to find which one is greater.	condition (num1 > num2) condition (num2 > num1)
3) If num1 is greater than num2, return num1 as the maximum.	return: num1 condition (num1 > num2)
4) If num2 is greater than num1, return num2 as the maximum.	return: num2 condition (num2 > num1)
5) If num1 is equal to num2, return either one since they are the same.	return: num1 condition (num1 == num2)
6) Return the maximum of the two numbers as determined in steps 3, 4, or 5.	return: {num1, num2} condition ANY

Figure 13: Step-based Labeling

The figures above illustrate how the dataset is collected using two categories of labeling: Strategy-based labeling and step-based labeling.

3.5.1.2 Early Experiments

After a manual collection that resulted in a total of 200 sequential reasoning strategies and 784 steps with their ADL translations, we conducted some fine-tuning experiments to see the extent of intervention required to achieve a good performance. Though, we knew from the beginning that even a thousand training examples is not sufficient enough to fine-tune a model. Nevertheless, we started experimenting with different open-source models, namely: Llama 3-8b from Meta AI, and Gemma 2b from Google.

As expected, these experiments resulted in a disastrous performance, indicating that the amount of intervention will be really high. However, as tedious as this task may seem, we came up with a solution that relatively mitigated the impact of having a small amount of data, and that is the augmentation system.

3.5.1.3 The Augmentation System

3.5.1.3.1 Overview

This system is a method of augmentation that we've designed to help create some diversification in our dataset and consequently collect more examples for our fine-tuning process. The idea behind it is applying a two-layer augmentation procedure incorporating a large language model and systematic techniques that we've developed to improve the results further. The first layer that comes in this procedure is the use of an LLM[2] - through Groq's API[3] - to generate different versions of each example in our dataset. Then, each branch generated by the model will go through the second layer of this system, which is systematic augmentation. The latter uses an algorithm designed to randomly change the names of the variables

[2]We've used Llama 3 from Meta AI as our LLM.

[3]We used the API provided by Groq, which enables the integration of state-of-the-art large language models such as Llama-3 with minimal latency by using the LPU Inference Engine.

for each generated branch. Therefore, we ultimately created various versions of the same generated branch but with other variable names. However, an important question arises. Why do we have to apply the second layer? The answer to this question is that it helps the model generalize beyond variable names and try to think about how to use those variables rather than copy-pasting their names. The following figure gives a visual explanation on how the augmentation system works.

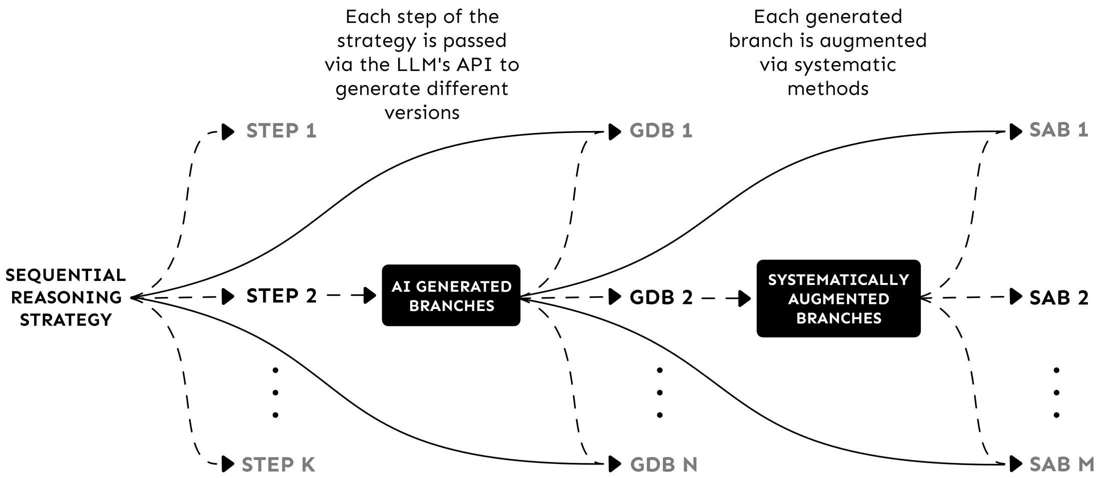

Figure 14: The Augmentation System

3.5.1.3.2 AI-Generated Branches

The augmentation process starts by taking each step k in the sequential reasoning strategy and constructing a prompt for the large language model - via Groq's API - that will generate n different versions of the step's description. The prompt used during this process is as follows:

prompt: *Generate a JSON list of* ***n*** *different and diverse versions for the following description:* ***{Step's Description}*** *Such that the JSON Object would be as follows: {"branches":* $VERSION_1$*,* $VERSION_2$*, ...}. Don't include function name, just write "function", "procedure", or anything appropriate instead. Also, don't write any code, just diversify the natural language expression. Preserve any variable names included in the description.*

The result of this operation is a *JSON* object that contains a list of n generated versions of the step's description from the strategy. This procedure is applied to all the strategies' steps. Finally, we construct a matrix of all the generated branches. The size of this matrix equals $k \times n$, where k is the number of steps in the strategy, and n is the number of versions generated by the LLM.

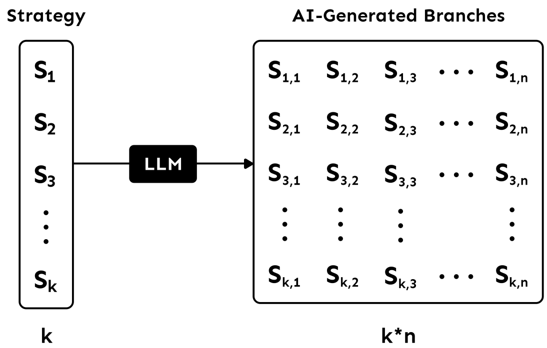

Figure 15: The Matrix of Generation

3.5.1.3.3 Systematically-Augmented Branches

After the matrix of generation gets constructed, the process continues via systematic methods that create more diversity by generating m mutations of variable names for each element in the matrix. The function designed to mutate variable names incorporates a distribution mask by which it selects a random range of characters from the ASCII Encoding. Then, while respecting the regular expression of identifiers, it generates a random sequence of characters from the selected range, resulting in random variable names. The question that arises from this technique is why using randomness as a form of mutation? Well, there's no simple answer to this question because it depends on the context of its application. The intuition for this comes from the fact that we want the ADL Generator to generalize beyond variable names and reason about them semantically. The reasoning to further understand this technique is after the ADL Generator fine-tunes, it should respect the syntactic structure of the variable names included in the step description; however, it should also use them appropriately according to the semantic meaning of the whole sequential reasoning strategy, this is achievable through randomness because we introduce noisy data to the LLM, which helps it generalize and improve its performance. Furthermore, it has been shown that introducing noise through the model's architecture is a good regularization technique to achieve generalization[4]. Though, we are not going to inject noise in the architecture but rather in the data.

Finally, we construct a tensor of all the systematically augmented branches, including the source of the augmentation, i.e., the original branch that was mutated m times. The size of this tensor equals $k \times n \times (m + 1)$, where k is the number of the steps in the strategy, n is the number of versions generated by the LLM, and m is the number of systematic mutations.

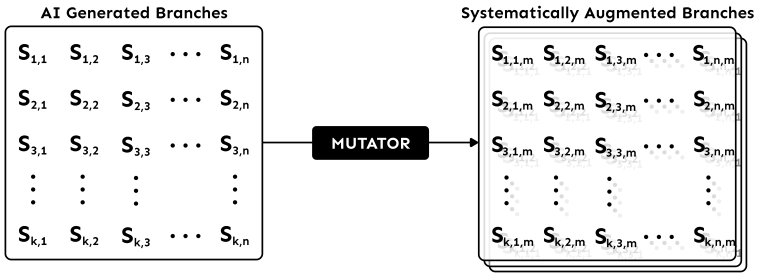

Figure 16: The Tensor of Mutation

Once the mutation tensor is created, we can collect more sequential reasoning strategies. This process is done by iterating through the tensor and considering each column as a strategy. Therefore, for one strategy, we ended up with $n \times (m + 1)$ more strategies.

The following figure illustrates the process:

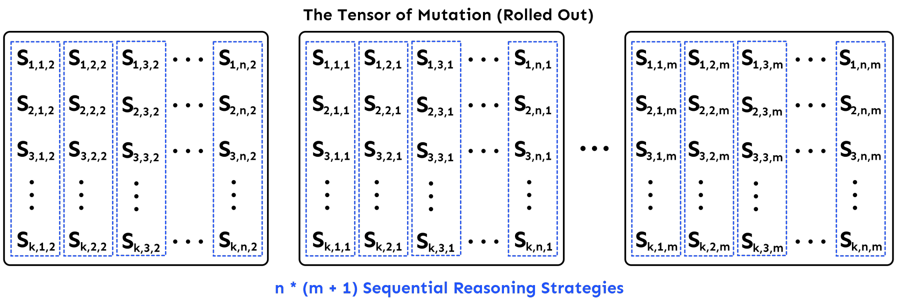

Figure 17: The Tensor of Mutation

[4]In the Dropout [9] paper, the authors showed that the model generalizes well when noise is introduced.

After the augmentation system's implementation, it was used over the manually collected data with $n = 10$ and $m = 2$, which led to the generation of 4000 Sequential Reasoning Strategies and 15680 Steps, each with their corresponding ADL translation. Assembling the dataset, we had to balance the ratio between Strategy-based Labeling and Step-based labeling. As such, we conducted other experiments of fine-tuning smaller models to choose the best ratio. Consequently, we found that a 40% Strategies and 60% Steps combination in the dataset will lead to better performance. It's worth noting that we didn't experiment with larger combinations since the time required to fine-tune the model gets longer as the size of the dataset increases.

- **How to select a good mutation number m:**

  We've conducted multiple fine-tuning experiments with $\{m = 0, m = 2, m = 3\}$, and we found that the choice for the best m value is counter-intuitive. When we evaluated the model with $m = 0$, it was not generalizing under particular inputs, but with $m = 2$, it showed a good generalization factor of 12%. However, with $m = 3$, there was a performance drop of about 7% from the experiment with $m = 2$, and we suspect that the issue goes to overfitting because, essentially, we have created way more paths to the output. Therefore, we speculate that the choice of m depends on the context of usage and the amount of diversity in the dataset.

3.5.1.4 Issues

Despite the effort that went into developing the augmentation system, and increasing the diversity of examples that allowed the collection of ~ 14000 training examples in the final dataset, it improved the performance only by an incremental amount that guarantees that the model will fail to generate consistent results, and that's due to the following reasons:

1. If the core dataset is small, then no matter how much the augmentation gets performed, it will not lead to stellar results because it doesn't increase the content's diversity. Instead, it only increases the linguistic diversity of the

training example, which means that it creates multiple paths for the same output. The figure below illustrates the increase in path density when doing such augmentation.

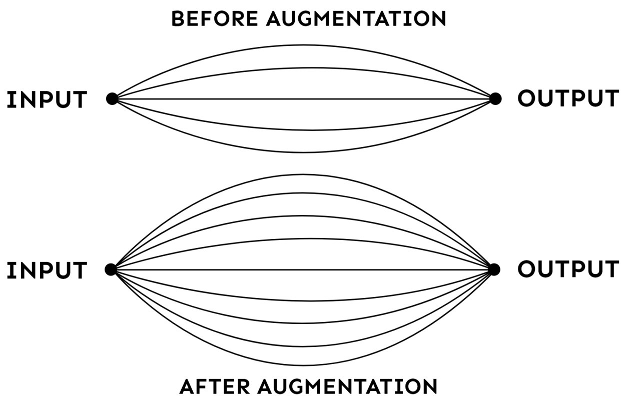

Figure 18: Before and After Augmentation

2. Generating a code-formatted language such as ADL is a really complex problem for a large language model to solve. Therefore, some form of guidance is necessary to help mitigate these issues.

3.5.1.5 Instruction-Specific Prompting

When the performance of an LLM is hindered by the lack and complexity of data, a valuable technique to use when training or fine-tuning, is to guide the model using what's called a System Prompt. It is a way to give the model some instructions on how to deal with the given input, which will drastically improve the ability to generate consistent results.

Generally, when a large-scale LLM is pretrained[5], they often have one very detailed system prompt to guide their generation process. Moreover, especially in these pre-trained models, their system prompts usually serve as guardrails to avoid giving sensitive and private information to the user. In our case, we thought of it as a guide for the ADL generator to adapt, syntactically and semantically, to the ADL format. However, given the diversity of the instructions, we thought of a technique

[5]GPT, Llama, and Claude are all examples of a pre-trained LLM.

we call Instruction-Specific Prompting, in which we classified each instruction in our dataset with a specific class, resulting in a set of 32 classes. Next, for each class, we wrote a system prompt with an average of 900 tokens[6]. Finally, when the model starts fine-tuning, each training example will get processed with its system prompt, thereby learning multiple ways to deal with each instruction. To exemplify, here's a snippet from the system prompt of the *var* class:

```
How variables are handled in ADL?

In the Abstract Description Language, there is a section called 'var', it is where all the variables
are declared.

The syntactic structure of the 'var' section comes as follows:

"""
var:
    <var_name> : <var_type>
"""

If we have multiple variables of the same type, a shortcut is using commas to separate variable names
and keeping just one <var_type> as shown in the following structure:

"""
<var_name>, <var_name> : <var_type>
"""
...
```

Figure 19: A snippet from the *var* class' system prompt

3.5.1.6 More Experiments

After we wrote the system prompts for all 32 instruction classes, we conducted more fine-tuning experiments to see how well the model performed. Surprisingly, the results were vastly different. They became more consistent, syntactically formatted, and semantically correct. This shows how much Instruction-Specific Prompting has impacted the training and, consequently, the generation of ADL translations.

3.5.2 Fine-Tuning

After all the steps that went into the data collection phase, we started the fine-tuning process of a larger ADL Generator model. This process consisted of multiple steps:

[6] We had used the tokenizer of the pre-trained model that we fine-tuned, Llama 3.

3.5.2.1 Model Selection

Before fine-tuning a model, one should select the model offering the best performance for their downstream task. Furthermore, since open-source models are more accessible than ever, the difficulty of choice is now alleviated. Within the context of the ADL Generator, we experimented with multiple open-source models, namely Llama 3[7][10] Instruct 8b, Gemma[11] 2b and 7b, and Starcoder 2[12].

1. **Starcoder 2**

   This model was a big achievement in the AI community when it first launched. It is an LLM trained with a large amount of code in multiple languages, including Python, C, C++, C#, Java, JavaScript, and many more, for code generation tasks. However, when we fine-tuned it on our data, it didn't generalize as much as we expected, and it often hallucinates and tries to inject programming languages in ADL's syntax.

2. **Gemma Models**

   Gemma models are smaller-scale LLMs from Google, trained on trillions of tokens with small architectures (with 2 and 7 Billion Parameters) to solve general tasks. Unfortunately, when we experimented with them, they showed a poor performance that didn't satisfy our needs[8].

3. **Llama 3 Instruct**

   Llama 3 is the new open-source large language model from Meta AI. We experimented with the 8 Billion parameters model, and surprisingly, it showed good generalization and accuracy in translating sequential reasoning strategies into ADL. Therefore, we chose this LLM as our base model for fine-tuning.

[7] At the time of writing, the only paper available is the Llama 2 paper.

[8] These models have also experienced some issues that were fixed later, regarding their context length.

3.5.2.2 Template

When a large language model trains, it usually has a prompt template to train it efficiently. As such, when someone wishes to fine-tune that model on their data, they should use the exact template it was trained on. Therefore, we processed our dataset by adapting it to the template of Llama 3, which is specified as follows:

```
<|start_header_id|>system<|end_header_id|>
{{ .System }}<|eot_id|>{{ end }}{{ if .Prompt }}<|start_header_id|>user<|end_header_id|>
{{ .Prompt }}<|eot_id|>{{ end }}<|start_header_id|>assistant<|end_header_id|>
{{ .Response }}<|eot_id|>
```

Figure 20: The fine-tuning template of Llama 3

- **{{ .System }}** corresponds to the system prompt. In this case, that would be one of the 32 system prompts we wrote.
- **{{ .Prompt }}** corresponds to the input prompt. In this case, it is the sequential reasoning strategy.
- **{{ .Response }}** corresponds to the output of the input prompt. In this case, it is the ADL translation of the given sequential reasoning strategy.

3.5.2.3 Hyperparameters and Techniques

Fine-tuning large language models such as Llama 3 with a size of 8 Billion parameters can be computationally expensive and resource-intensive, often requiring substantial hardware and long training times. Fortunately, thanks to recent contributions, we finally get to leverage techniques like LoRA[13] (Low-Rank Adaptation) and QLoRA[14] (Quantized Low-Rank Adaptation) to address these challenges by significantly reducing the number of trainable parameters and the memory footprint during the fine-tuning process. LoRA achieves this by decomposing weight updates into low-rank matrices, which simplifies the update process and makes it more efficient. QLoRA further optimizes this by applying quantization techniques, reducing the precision of certain computations and thus further lowering the memory and computational requirements.

The following list offers the configuration of our fine-tuning:

- **Rank** (r) The rank of the low-rank matrices used to decompose weight updates. This controls the trade-off between approximation accuracy and parameter efficiency. We set it to $r = 64$.
- **Alpha** (α) A scaling factor applied to the low-rank matrices. This helps in controlling the magnitude of the updates. We set it to $\alpha = 16$.
- **Learning Rate** The learning rate for training the model. We set it to 2×10^{-4}.
- **Quantization Bit-width** The number of bits used for quantizing the model parameters. We set it to 4-bit.

3.5.3 Limitations

Despite the impressive capabilities of large language models, they still suffer from several limitations, and the ADL Generator is no exception.

- The likelihood of hallucination increases proportionally with the complexity of the sequential reasoning strategy. Moreover, its inherent bias will force the model to write irrelevant code that most likely doesn't obey the syntax of ADL.
- The ADL Generator's capabilities are limited by the size of the base model. As the model grows, its capabilities increase. To exemplify this, when given a mathematical program that uses division operations, sometimes the model will miss putting a condition on the non-zero nature of a variable in the denominator and assume that it is not. As such, when these issues accumulate, they will hurt the correctness of the model by a significant factor.

3.5.4 Beyond Limitations: Syntax Correction

Given the model's limitations and syntactic error susceptibility, making it difficult to process in the conversion engine, we thought of a way to mitigate occurring syntax errors. It establishes a syntax correction loop in which the ADL Generator interacts with ChatGPT to correct syntactic errors in the generated ADL, which is done by giving a grammar template of the Abstract Description Language to ChatGPT and using it to detect any text that doesn't obey the ADL grammar. Then, it tries to correct it depending on its semantics. This syntax correction loop increases the chances of generating correct ADL descriptions, which consequently leads to better results in the conversion phase.

Chapter 4

Results

4.1 Snippets of Sequential Reasoning Strategies

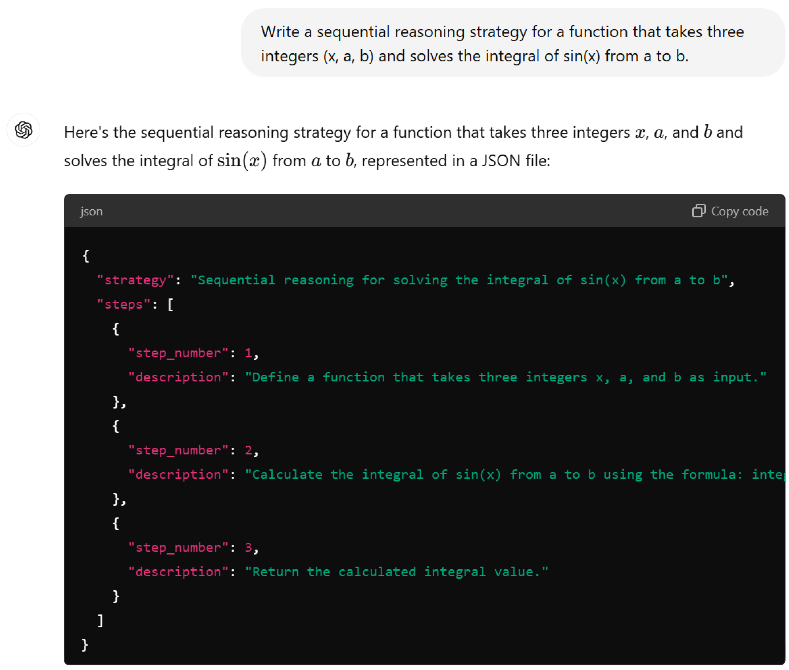

Figure 21: A snippet for a sequential reasoning strategy that computes the integral of $\int_a^b sin(x)\,dx$ from a to b written by ChatGPT (in JSON Format).

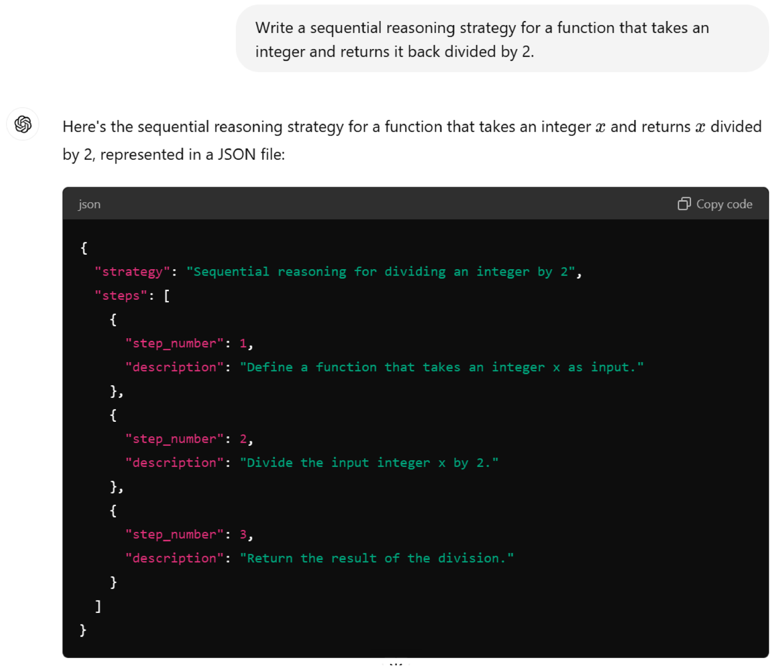

Figure 22: A snippet for a sequential reasoning strategy that computes the division of an integer by 2 written by ChatGPT (in JSON Format).

4.2 Evaluating the ADL Generator

We fine-tuned the model for one epoch of training (about 3 hours and 37 minutes) using an NVIDIA A100 GPU from Google via Colab Pro Notebooks, which resulted in a significant training loss drop as shown in the figure below.

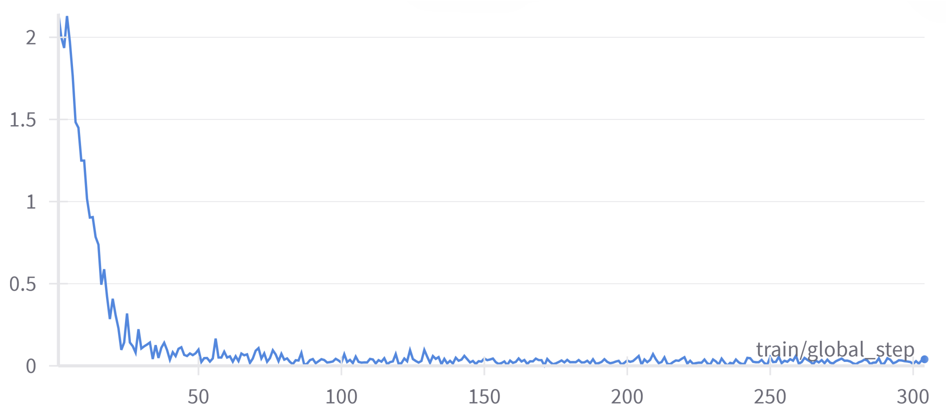

Figure 23: A plot depicting the drop in training loss of the model as it undergoes fine-tuning.

It's worth noting that to enhance efficiency, we used the Unsloth notebook to fine-tune the model.

Generally, when evaluating large language models, it is usually standard to have a benchmarking dataset on which we measure how well the model performs, such as Math, Reasoning, Coding, etc. In our case, there's no benchmarking dataset other than the validation set we split from the training set. Furthermore, each benchmarking dataset corresponds to a metric we can use to measure the model's performance on that dataset quantitatively, and there are several metrics such as BLEU[15], ROUGE[16], Accuracy, etc. However, as the contract generation process - via Pyramid - depends on a strict syntactic format, we haven't found any efficient metric for such evaluation. Nevertheless, we employed a manual approach, examining dozens of examples and scoring each on a scale of 1 to 5. When scoring, we look for syntactic correctness and then for semantic correctness. Finally, after the manual individual evaluation, we average the scores to get a global performance measurement. As a result, over 60 samples, the ADL Generator achieved **81.6%** average correctness on the manual evaluation.

4.2.0.1 A Snippet

````
1) Create 'a' procedure that accepts three inputs: the lengths of the sides 'a', 'b', and 'c' of the triangle.
2) Determine the sine of angle C by dividing the length of side 'c' by that of side 'a', giving'sin_c' = 'c' / 'a'.
3) Retrieve the trigonometric sine of angle C, stored in variable'sin_c'.
<|eot_id|><|start_header_id|>assistant<|end_header_id|>

Based on the provided sequential reasoning strategy, I will convert it to ADL. Here is the ADL translation:

```
var:
    a, b, c: integer
action:
    sin_c = c / a condition ANY
return:
    sin_c condition ANY
```
````

Figure 24: An actual output of the ADL Generator for a sequential reasoning strategy.

The figure above is the ADL Generator's response for a sequential reasoning strategy, and it shows a good syntax and semantic generalization of the model. However, as was emphasized in the limitations, the model's capabilities are limited by size. Thus, it is susceptible to weakly expressed conditions. For example, the response above assumes that 'a' is not zero, which is not a strong aassumption, especially if we seek specification completeness.

4.3 Pyramid and Contract Refinement

After a sequential reasoning strategy gets translated into ADL, the latter will get converted into ACSL contracts via Pyramid.

```
ensures ((num1 > num2)) ==> \result == num1;
ensures ((num2 >= num1)) ==> \result == num2;
```

Figure 25: Actual contracts generated by Pyramid for an ADL of a program that returns the maximum of two numbers.

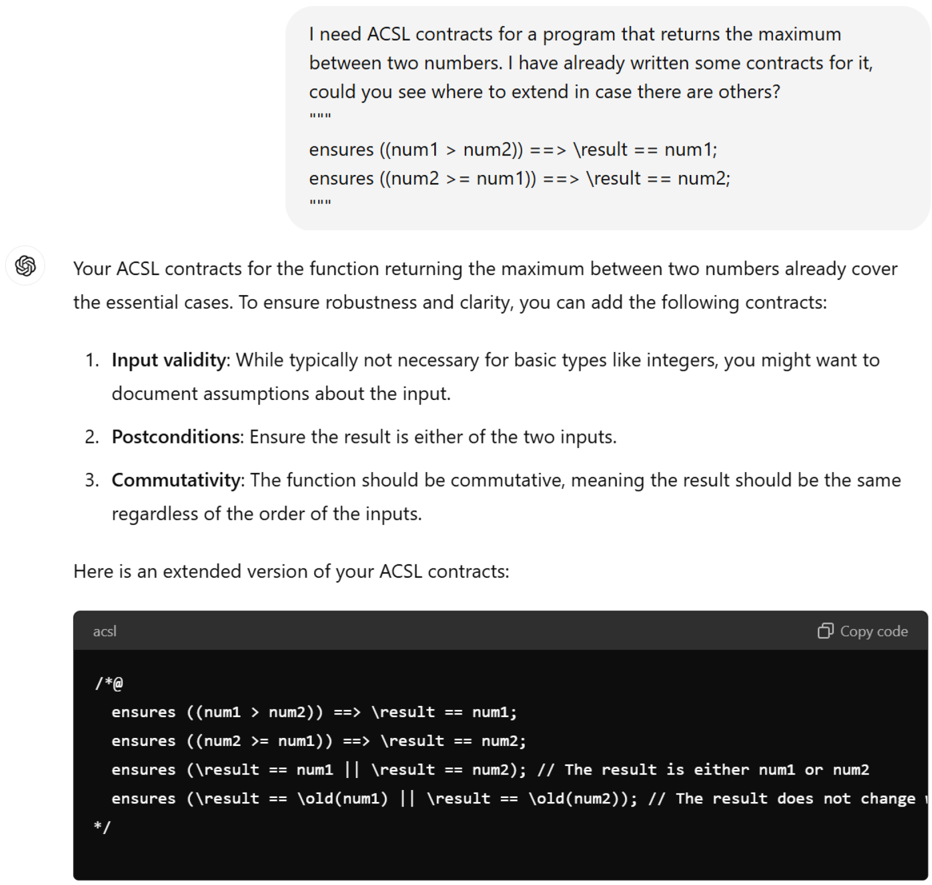

Figure 26: The ChatGPT's refined version of The contracts generated by Pyramid above.

As illustrated in the figure above, to transcend the limitations of the conversion engine and achieve a closer approximation of completeness, the generated contracts are subsequently sent to ChatGPT, which refines them into a more comprehensive set of contracts

4.4 On The Performance of The System

After a sequential reasoning strategy gets translated to ADL, contract generation becomes deterministically systematic via the conversion engine. As such, we could claim that the system's performance is the same as the ADL Generator's performance. However, when measuring the performance of a contract generation system, it's up to the developers to decide what metric of priority they need. In other words, sometimes, they could prioritize the completeness of the specifications, and other times, they could prioritize a particular spectrum of specifications. Therefore, evaluating the system as a whole necessitates the development of efficient metrics. Nevertheless, from the prototype results we got in the development phase, we could see a glimpse of hope surrounding this system.

Chapter 5

Perspectives and Conclusion

5.1 Perspectives

To compensate for the completeness of our work, we envision continuing the progress as soon as possible, as we already have plans for more contributions that will help improve the results of the system. Firstly, in the experiments that we conducted, we relied on Frama-C to check whether the generated contracts are correct or not. However, as programs get complex, the verification runtime of Frama-C would get longer as well. The issue here is a longer runtime will hinder our ability to establish an autonomous feedback loop between the system and Frama-C, which can provide a way to guide the model toward correct contracts. Secondly, as was proven in the ADL's study of expressiveness, increasing the verbosity of ADL would contradict the goal of developing the language in the first place. As such, we are trying to find ways to inject language constructs that maximize our use of ADL while also maintaining the same level of abstraction. Moreover, while the completeness of Pyramid is not guaranteed, we have some ideas to expand the set of contracts that it can generate and improve the robustness of the contract refinement loop that we have established with ChatGPT. Ultimately, these plans will be carried out with careful precision and compiled in a research publication, including a thorough evaluation to gauge the system's potential in industry-scale environments.

5.2 Conclusion

In conclusion, formal methods play a significant role in the development of reliable, robust, and safe software. It contributes to the endeavors of building technologies that change our lives for the better, and with the rise of intelligent systems, safety is needed more than ever. Autonomous vehicles and Robots are rapidly advancing to become normalized in our society. Given the potential integration of these technologies into our daily lives, it's becoming more prominent to develop guardrails for them. Yet, these guardrails are complex to build because they rely on a high level of rigorousness that necessitates significant expertise. However, as AI continuously shows impressive solutions to a wide variety of complex problems, it became clear that we can help small teams to compensate for the requirement of expertise, enabling a budget reduction for safety and thereby setting a safety standard for all emerging applications.

Bibliography

[1] Vaswani, A. *et al.* Attention is all you need **30** (2017). URL https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[2] Baudin, P. *et al.* Acsl: Ansi/iso c specification language (2008). URL https://api.semanticscholar.org/CorpusID:122818859.

[3] Leavens, G. T. Tutorial on jml, the java modeling language 573 (2007). URL https://doi.org/10.1145/1321631.1321747.

[4] Ma, L., Liu, S., Li, Y., Xie, X. & Bu, L. Specgen: Automated generation of formal program specifications via large language models (2024). URL https://arxiv.org/abs/2401.08807. 2401.08807.

[5] OpenAI *et al.* Gpt-4 technical report (2024). URL https://arxiv.org/abs/2303.08774. 2303.08774.

[6] Janßen, C., Richter, C. & Wehrheim, H. Can chatgpt support software verification? (2023). URL https://arxiv.org/abs/2311.02433. 2311.02433.

[7] Cuoq, P. *et al.* Frama-c: a software analysis perspective 233–247 (2012). URL https://doi.org/10.1007/978-3-642-33826-7_16.

[8] Felleisen, M. On the expressive power of programming languages. *Science of Computer Programming* **17**, 35–75 (1991). URL https://www.sciencedirect.com/science/article/pii/016764239190036W.

[9] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. & Salakhutdinov, R. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research* **15**, 1929–1958 (2014). URL http://jmlr.org/papers/v15/srivastava14a.html.

[10] Touvron, H. *et al.* Llama 2: Open foundation and fine-tuned chat models (2023). URL https://arxiv.org/abs/2307.09288. 2307.09288.

[11] Team, G. *et al.* Gemma: Open models based on gemini research and technology (2024). URL https://arxiv.org/abs/2403.08295. 2403.08295.

[12] Lozhkov, A. *et al.* Starcoder 2 and the stack v2: The next generation (2024). URL https://arxiv.org/abs/2402.19173. 2402.19173.

[13] Hu, E. J. *et al.* Lora: Low-rank adaptation of large language models (2021). URL https://arxiv.org/abs/2106.09685. 2106.09685.

[14] Dettmers, T., Pagnoni, A., Holtzman, A. & Zettlemoyer, L. Qlora: Efficient finetuning of quantized llms (2023). URL https://arxiv.org/abs/2305.14314. 2305.14314.

[15] Papineni, K., Roukos, S., Ward, T. & Zhu, W.-J. Bleu: a method for automatic evaluation of machine translation 311–318 (2002). URL https://aclanthology.org/P02-1040.

[16] Lin, C.-Y. ROUGE: A package for automatic evaluation of summaries 74–81 (2004). URL https://aclanthology.org/W04-1013.

While formal verification experts face myriad challenges in writing contracts, AI offers an accelerated doorway to achieving a certain level of automation within the task of contract generation. This report introduces a Neurosymoblic system that synergizes state-of-the-art techniques in Generative AI with systematic approaches in Language and Compiler Designs to generate ACSL specifications, thereby facilitating software verification.

Alors que les experts en vérification formelle sont confrontés à une multitude de défis lors de la rédaction de contrats, l'IA offre une voie accélérée vers l'obtention d'un certain niveau d'automatisation dans la tâche de génération de contrats. Ce rapport présente un système neurosymoblique qui synergise les techniques de pointe en IA générative avec des approches systématiques en matière de conception de langage et de compilateur pour générer des spécifications ACSL, facilitant ainsi la vérification des logiciels.

SETIME

جامعة ابن طفيل
Ibn Tofaïl University
Faculté des Sciences