Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Semantic performance-analysis of LLM-based autoformalization

Tesis de Licenciatura en Ciencias de la Computación

Juan Manuel Baldonado

Director: Víctor A. Braberman
Codirector: Flavia Bonomo
Buenos Aires, 2024

RESUMEN

En los últimos años, los LLMs (Large Language Models) han experimentado un enorme crecimiento en popularidad, en parte debido a su versatilidad para abordar una gran variedad de tareas "downstream" sin necesidad de reentrenamiento. Esto se logra con el uso de distintas técnicas de ≪prompt engineering≫, que permiten condicionar las respuestas del modelo en función de la tarea que se desea resolver. Consecuentemente, se ha iniciado una revolución en términos de desarrollo de un tipo de software (el ≪promptware≫) que utiliza LLMs para resolver las más variadas funcionalidades. Sin embargo y a pesar de los constantes avances, desarrollar software basado en interactuar con LLMs carece de teoría y métodos que soporten enfoques disciplinados. De hecho, un área con carencias significativas es la evaluación (y mejora) de performance de un LLM para una tarea dada. Muchas veces no se tiene en cuenta la naturaleza estocástica del proceso generativo subyacente y la competencia de formas superficiales en las que se pueden presentar los resultados a una pregunta. En este trabajo analizamos la distribución de las respuestas generadas por LLMs en función de su contenido semántico. Estudiamos la performance de una tarea desde la perspectiva de la propiedades de la distribución "clusterizada" resultante, el vínculo con los resultados esperados y los tipos de errores de alineamiento. Usamos esas observaciones para ejemplificar mecanismos más disciplinados de mejoras basadas en la descomposición de tareas. Nos centramos en la tarea de auto-formalización, que consiste en generar una representación formal de una descripción en lenguaje natural. En particular, el problema a analizar será el de generar especificaciones de programas a partir de su documentación.

Palabras claves: Autoformalización, Large Language Models, Clustering Semantico, Dafny.

ABSTRACT

In recent years, Large Language Models (LLMs) have experienced a significant growth in popularity, partly due to their versatility in tackling a wide range of downstream tasks without the need for retraining. This is achieved through various techniques of "prompt engineering," which allow the model's responses to be conditioned for on the task at hand. Consequently, a revolution has begun in the development of a new type of software (*promptware*) that leverages LLMs to perform a wide array of tasks. However, despite the constant advancements, developing software that interacts with LLMs lacks the theoretical foundations and methods to support disciplined approaches. In fact, one area with significant gaps is the evaluation (and improvement) of an LLM's performance on a given task. Often, the stochastic nature of the underlying generative process and the alternative variants in which answers to a question can be presented are not taken into account. In this work, we analyze the distribution of responses generated by LLMs based on their semantic content. We study the performance of a task from the perspective of the properties of the resulting "clustered" distribution, its alignment with expected results, and types of alignment errors. We use these observations to exemplify more disciplined improvement mechanisms based on task decomposition. We focus on the task of auto-formalization, which involves generating a formal representation of a description in natural language. Specifically, the problem to be analyzed will be generating program specifications from their documentation.

Keywords: Autoformalization, Large Language Models, Semantic Clustering, Dafny.

AGRADECIMIENTOS

A Víctor y Flavia, por todo su apoyo, guía y motivación.
A mis padres, mi hermano y mis amigos.
Al Departamento de Computación y a la Universidad de Buenos Aires.

CONTENTS

1. INTRODUCTION

In recent years, we have seen the emergence of Large Language Models, which have demonstrated great potential for various applications. These models are large neural networks, often made up of tens or even hundreds of billions of parameters trained on vast, unlabeled text corpora to predict the next word in a piece of text based on the preceding context. Notable examples of these models include OpenAI's GPT-4[24], Meta's Llama[29], and Google's Gemini[28] which have gained significant popularity both in the scientific community and among the general public.

These models have an impressive performance in natural language-related tasks like text summarization, question-answering, and text completion [8]. The success of these models is often attributed to in-context learning. Several studies have shown that by simply conditioning these models on a set of instructions describing the task (zero-shot) or on a small set of examples (few-shot), they are able to perform a diverse variety of tasks effectively [33].

Large language models have been widely applied on the field of software engineering and formal verification [6]. Their effectiveness has been studied in various areas, such as unit-test generation[2], fuzzing [16], penetration testing [9], root-cause analysis [37], vulnerability detection [12] and program verification [26], [34].

Despite its recent significant breakthroughs, working with LLMs remains challenging due to various factors. These models consist of complex parameterized functions with sometimes tens or even hundreds of billions of parameters, making it difficult to pinpoint exactly how they arrive at a particular response. Additionally, LLMs exhibit nondeterministic behavior, which arises from the sampling process used during the generation phase for positive temperature values, where the sampling distribution often depends on the choice of prompt and context. Most notably, these models are prone to a phenomenon usually known as hallucinations in which the generated content contains factual or logical errors.

Recent research by Farquhar explored a novel approach aimed at detecting hallucinations by measuring what they define as semantic entropy [11]. This method evaluates the uncertainty at the semantic level rather than focusing on specific word sequences. Their approach involves clustering the outputs generated by the model into equivalence classes based on their semantic similarity and then computing the entropy of the clustered distribution. They evaluated this metric on a range of tasks and found that semantic entropy outperforms other popular approaches like self-consistency or a self-check using the model itself.

In this thesis, we explore a similar framework for analyzing LLM-based approaches in the context of auto-formalization, a task that involves generating formal representations from informal descriptions or problem statements. We focus on auto-formalization because the domain of formal specifications is especially suited to semantic analysis. Formal languages can be equipped with an equivalence relationship that enables semantic clustering of formal statements, allowing us to assess not only the quality of particular instances but also the uncertainty of the solutions generated by a language model.

A key aspect of our investigation is that we intend to analyze performance in terms which differentiates from the traditional performance metrics used in other studies. In this case, we analyze performance in terms of the distribution of the model's outputs across a clustered support, rather than a singular metric of performance or accuracy. We examine how the concentration of outputs—i.e., the language model's tendency to produce similar responses—relates to the quality of the generated formalizations. With this qualitative analysis we look to identify common failure modes and how concentration relates to the quality of the generated formalizations.

We also aim to perform a re-engineering of existing LLM-enabled solutions integrated with formal verification tools, such as Dafny [20], in order to refine the solution approach. We seek to understand whether the insights gained from this analysis can guide improvements in prompt-based design, ultimately contributing to a more systematic and robust framework for prompt engineering and LLM-based solutions.

To this end, we pose the following research questions that will guide our analysis of the interaction between alignment, concentration, and the quality of auto-formalizations produced by an LLM-based approach:

1. **Can we observe a link between alignment and concentration on the auto-formalization task?** We seek to determine whether there is a measurable connection between alignment, how closely the generated specifications match the intended formal specifications, and concentration (the consistency between the generated specifications). Additionally, we want to investigate whether other potential measures can serve as indicators for performance on the auto-formalization task.

2. **What happens when concentration and non-alignment occur?** By looking into the specific instances where outputs are concentrated but not aligned with the reference specifications, we hope to uncover if there are specific logical structures or types of annotations that are consistently misinterpreted by the model.

3. **Can these observations inform targeted improvements in the model's auto-formalization capabilities?** We want to explore how the insights gained from our analysis can inform the development prompting methods that could potentially lead to an improvement of the capabilities of an LLM-based auto-formalization system.

2. LARGE LANGUAGE MODELS

In the recent decades, statistical language modeling has become fundamental to many natural language processing tasks, ranging from speech recognition, machine translation, to information retrieval. Statistical language modeling essentially looks to capture the regularities of natural language by modeling the probability distribution of various linguistic units, such as words and sentences.

Earlier language models were based on the Markov assumption, stating that the probability of the next word in a sentence can be characterized using the most recent context [18]. Mathematically, this can be expressed as:

$$P(w_n|w_1,..,w_{n-1}) = P(w_n|w_{n-k},..,w_{n-1}) \tag{2.1}$$

where w_n is the next word and k is the size of the context window. This means we only consider the last k words when estimating the probability of w_n. On these earlier models, this distribution was typically estimated from the observed text frequencies on a text corpus. While this simplification made early language models computationally feasible, it limited their ability to capture long-range dependencies and more nuanced semantic relationships between words which often resulted in generated text that lacked coherence and fluency.

Modern approaches often use an intermediate numerical representation for text known as word-embeddings. In this process, text is segmented into tokens which consists of smaller linguistic units such as words, sub-words, or punctuation marks. Each token is then projected or embedded into an n-dimensional vector space vector through techniques like Word2Vec or transformers based approaches [21], [31].

In recent years, neural language models emerged, which essentially model this conditional distribution using neural networks[5]. These models, though often more complex and with more parameters allow us to model more complex and non-linear functions. These models are functions parameterized by a set of parameters θ which are often found by optimizing an objective or loss function for a particular task. In the case of language models, where the task is generally next-token prediction, the objective is to minimize the negative log-likelihood of the true next token given the previous tokens

$$L(\theta) = -\sum_{t=1}^{T-1} \log P_\theta(x_{t+1}|x_{1:t}) \tag{2.2}$$

where P_θ is our neural language model under training, with parameters θ. This loss function depends on both the parameters θ and the data $x_1, \ldots, x_T$. By minimizing this loss, we effectively maximize the likelihood of the observed sequence of tokens, improving the model's capacity to predict future tokens accurately.

The recent advances on more efficient neural architectures like the transformer, combined with distributed training methods allowed language models to scale to tens or even hundreds of billions of parameters which are trained on large Web-scale text corpora. This

new class of language models, now often referred to as Large Language Models (LLMs) have significantly extended the capabilities and performance of statistical language models beyond the task for which they were originally trained on. Moreover, their generative nature allows them to create coherent and contextually relevant text that can mimic various styles, tones, and formats. Not only this, but also numerous they seem to be capable of following human instructions and performing multi-step reasoning.

2.1 Text Generation and Sampling

As generative language models, large language models not only capture the statistical properties of language but also enable the generation of text by using the distribution learned during the training phase. This process involves using a given context as input, from which the model computes the likelihood of all tokens in its vocabulary that could follow. This is, given a context x, the language model's output is not exactly a single token but rather a probability distribution over the set of tokens or vocabulary. Let $P(t_k|x)$ denote the probability of the $k-th$ token t_k given the context x. The LLM then selects the token t_k with the probability according to:

$$t_k \sim P(t_k|C, t_{k-1}, .., t_{k-w}) \tag{2.3}$$

This process continues until either the number of sampled tokens exceeds a threshold value or until a special token, the EOS (End-of-Sentence) token is sampled which indicates the end of the text. One parameter is often used to control the randomness of the distribution is the temperature parameter T. This parameter essentially adjusts the probability output probability distribution:

$$P(t|x) = \frac{\exp(\frac{\log P(t|x)}{T})}{\sum_{t' \in V} \exp(\frac{\log P(t'|x)}{T})} \tag{2.4}$$

In essence, higher temperature values ($T > 1$) flatten the probability distribution by making it more likely to choose less probable tokens while lower temperature values ($T < 1$) concentrate the probability distribution over the most probable tokens leading to more deterministic outputs.

Another refinement technique is top-k sampling, which limits the number of tokens considered for sampling to the top-k tokens with the highest k logit values. This essentially limits the number of tokens considered for sampling to the top k tokens with the highest probabilities:

$$V_k = \{t_1, t_2, \ldots, t_k\} \quad \text{where} \quad P(t_i|C) \geq P(t_{i+1}|C) \tag{2.5}$$

2.2 The prompting paradigm

The prompting paradigm rose to prominence with the release of GPT-3 series of models by OpenAI. In prompting, a pre-trained language model is provided with a prompt or instruction in natural language that describes a specific task. In this way, the language model is conditioned to generate probable responses (i.e sequences with a high likelihood) based on what was learned from the training corpus. Since their release, people have begun

to explore diverse conversational strategies, from straightforward queries to more nuanced dialogues that involve context switching, role-playing, and iterative feedback. This has led to the emergence of multiple patterns of interaction based on their conversational nature and the dynamic way they engage with users.

One of the most-commonly employed techniques, *zero-shot* prompting or in-context learning, consists of providing just the instruction and particular instance of the task to be solved. Despite its broad applicability, some tuning of the prompt is required to enhance performance, especially for complex tasks. Previous research studies have identified general principles or guidelines for designing effective prompts which include specifying clear output patterns, breaking down the task and expliciting output constraints.

Zero-shot Prompt Example

Instruction: Determine if the sentiment of this review is positive or negative.
Input: The product exceeded my expectations and I am extremely satisfied with my purchase.

Above we show an example to demonstrate a zero-shot prompt where the model is asked to determine the sentiment of a review without any prior examples. Even though this method is applicable to many problems, earlier language models frequently encounter difficulties with more complex tasks.

A technique known as *few-shot* prompting addresses this last issue by including some examples of the task along with the instruction in order to help guide the model's responses. The main challenges with this prompting strategy often revolve around choosing examples that will effectively inform the model. Numerous studies have focused on optimizing these few-shot examples to enhance performance and found that this approach typically performs better when in-context examples closely resemble the test examples. This is also crucial since the model's context window is generally limited.

Few-shot Prompt Example

Instruction: Determine if the sentiment of this review is positive or negative. Below are some examples:
Example 1: The food was terrible and the service was worse.
Sentiment: Negative
Example 2: I love this place! The ambiance is fantastic.
Sentiment: Positive
Input: The product exceeded my expectations and I am extremely satisfied with my purchase.

Another approach, chain-of-though (CoT) prompting refines direct prompting by encouraging the model to verbalize or articulate intermediate reasoning steps. This technique attempts to endow language models with the ability to generate a coherent series of intermediate reasoning steps that lead to the final answer for a problem. Wei et al show that this technique enhances reasoning for sufficiently large language models by enabling them

to break down multi-step problems into intermediate steps and its applicability to various tasks, including math problems and commonsense reasoning [32].

Chain-of-Thought Prompt Example

Instruction: Solve the following math problem.
Input: If a train travels at a speed of 60 km/h for 2 hours, how far will it have traveled?
Tought: First, identify the speed and the time traveled. The train travels at 60 km/h, and it travels for 2 hours. To find the distance, use the formula Distance = Speed × Time. Multiply 60 km/h by 2 hours to get 120 km.
Answer: 120 km

In the example, the chain-of-thought prompting method is demonstrated using a math problem. The model is asked to determine the distance a train travels at a speed of 60 km/h for 2 hours. By breaking down the problem into steps, the model identifies the speed and time, applies the formula for distance, and calculates the answer.

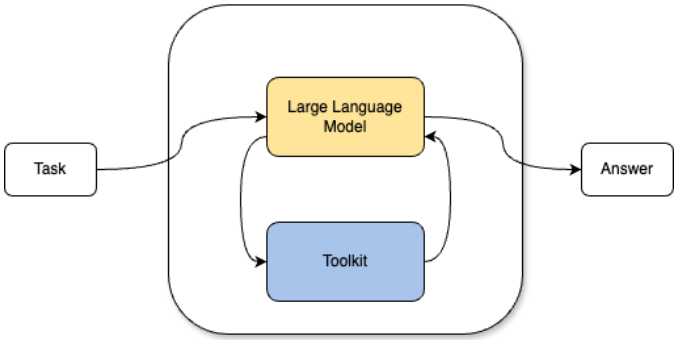

Fig. 2.1: Depiction of how LLMs can interact with external tools

More recently, LLM-applications have been extending large language models (LLMs) with the ability to interact with their environment which ends up generating feedback loops: their previous outputs influence the state of the world, which subsequently affects their future outputs. This can help improve their responses by augmenting the language model's knowledge base and enabling them to iteratively refine their answers with feedback from these external sources or environment. Some examples of this are giving the ability to call external APIs to retrieve documents from an external database or execute code.

2.3 The Modern LLM Landscape

Training large language models (LLMs) from scratch is often infeasible due to the required computational resources and time. For instance, training a model like GPT-3, which has 175 billion parameters, is estimated to cost around $4.6 million and take several months using thousands of high-end GPUs. Thus, the process becomes prohibitively expensive, even for large organizations. Instead, researchers and developers rely mostly on pre-trained versions of these models, which have already undergone training. This approach allows them to fine-tune the models for specific tasks, significantly reducing the cost and effort involved while still leveraging the capabilities of these large models.

In the modern landscape of Large Language Models, there are numerous models to choose from, both open-source and closed-source. For open-source language models, their parameters are made available to the developers or researchers who can run then perform inference on their own hardware. Some open source models, such as Meta's LLaMA, have garnered significant attention for their competitive performance relative to much larger models [29]. The models in the LLaMA series have been trained on a diverse range of data sources, enabling them to handle various natural language processing (NLP) tasks with high accuracy. Additionally, these models are designed to be highly adaptable and can be customized through fine-tuning for specific use cases, such as text generation, question-answering, or even more niche applications like code synthesis. However, it's important to note that operating the larger versions of open-source models still requires substantial computational resources, which can be a limiting factor.

Google's Gemma family of models is another noteworthy addition to the open-source landscape. The Gemma models are lightweight, state-of-the-art open models built upon the same principles used to create the Gemini models. Gemma models come in various sizes, including 2B, 7B, 9B, and 27B, and are designed for different use cases. One particular model in the Gemma family is Code-Gemma, which is specialized in code-related. Code-Gemma is optimized for code completion and generation, making it a note-worthy candidate. However, it is usually restricted to mainstream programming languages like Python, which limits its applicability to niche or less common languages.

On the other hand, closed-source models like OpenAI's GPT, Anthropic's Claude, and Google's Gemini, while often more expensive and less flexible in terms of tuning and operational customization, provide significantly better performance in downstream tasks. Despite the higher costs and reduced flexibility, the superior capabilities of these larger closed-source models make them a compelling choice for many organizations, developers and researchers. These models can be accessed through an API, enabling users access the models' powerful capabilities without needing extensive infrastructure or computational resources.

On some preliminary tests, we found the performance of smaller open-source models like LLaMA 7B and Gemma on formalization tasks to be far behind the capabilities of the other much larger models and it not being able to generate specifications that followed the Dafny language syntax.

2.4 Measuring the quality of generated LLM outputs

As we mentioned before, LLMs exhibit non-deterministic behavior which arises from the sampling process used during the generation phase, where the distribution is often dependent on the choice of prompt and context. These models are particularly prone to a phenomenon often known as hallucinations in which the generated content contains factual or logical errors. Furthermore, hallucinations in LLMs can take several forms. Recent studies have aimed to categorize these hallucinations into different types [17], [38]. These categories include, for example, input-conflicting hallucinations, when the generated output deviates from user input or task instructions. These are often a result of misunderstanding the user's intent or failing to align the response with the given task.

Context-conflicting hallucinations occur when LLMs lose track of the conversation or context, leading to inconsistencies, especially in longer, multi-turn interactions. Finally, fact-conflicting hallucinations happen when the model generates information that contradicts established world knowledge, which can mislead users and introduce factual inaccuracies.

Research by Kadavath et al. studied whether language models can evaluate the validity of their own claims and predict which questions they will be able to answer correctly[19]. They hypothesized that model hallucinations tend to be diverse. When a model is confident in its answer, it tends to give consistent responses, resulting in low entropy (less diversity). Conversely, when unsure the model "hallucinates" and produces a wide range of answers, leading to high entropy (more diversity). They found that the entropy distribution differed based on whether the model's answers were correct or incorrect, suggesting some predictive power. However, as models increase in size, they tend to solve more complex problems and provide a diverse set of correct answers thus the entropy of the token sequences as they measured might not be a robust metric.

3. FORMALIZATION

3.1 Formal Methods

In computer science and mathematics, a series of methods known as formal methods emerged to specify and verify software systems as well as assisting in the proof of mathematical theorems. These methods have their roots on the early 19th century work of mathematicians that aimed to formalize and axiomatize mathematics [3].

Formal methods rely on formal languages to define objects and specify their properties and behavior. Formal languages are essentially sets of sequences of symbols from a predefined alphabet together with a formal grammar, a set of rules that determine which sequences constitute valid expressions within a particular language [15]. In addition to a grammar or syntax rules, these languages also require the definition of semantics, which determines the meanings of the symbols and expressions. Finally, formal methods rely on the definition of inference rules, which allow the derivation of additional facts and can be used to guide the search towards the goal or property we are attempting to prove.

In computer science and software engineering ensuring the correct, secure, and safe operation of software systems is crucial. While testing the most frequently used approach to assure the quality of an implementation, it primarily aims at uncovering errors or issues and cannot guarantee their absence. In contrast, formal verification methods provide stronger quality guarantees by formally proving the correctness of a software system. These methods rely on a formal description or specification of the system or program under analysis, which essentially specifies the properties that the system must satisfy. Today formal methods are used, among other things, to specify and verify the correctness of computer programs and communication protocols, identify vulnerabilities, etc. Some formal verification methods for example use an approach known as model checking to specify the properties that the system should satisfy and then make use of a SAT or SMT solver to search for a counterexample trace, that is, an execution path that violates the specification.

3.2 The Dafny Programming Language

The Dafny programming language was developed at Microsoft Research with the purpose of integrating verification and software development into the programming language itself[20]. Dafny has been successfully used in various projects and research initiatives, particularly in areas requiring strong correctness guarantees. For instance, it has been used in a series of research projects by Microsoft Research to develop reliable and formally verifiable distributed systems protocols[14].

The Dafny programming language implements many of the constructs and data structures that can be found in most imperative programming languages today such as *if* statements, *for* and *while* loops, arrays, sets and event support for object oriented programming.

Additionally, Dafny implements a specification language to allow developers to write formal specifications and verify them using a verification framework based on Hoare logic.

Hoare logic, proposed in 1969 by Tony Hoare, is based around the concept of a Hoare triplet to describe and reason about a program's behavior. A Hoare triplet is an expression of the form $\{P\}\ S\ \{Q\}$ where P is a pre-condition, that is, a logical expression that must be true before executing the statement or sequence of instructions S. The expression Q is the post-condition, a logical expression which must hold true after the execution of S. We say that an implementation of a computer program is partially correct with respect to a specification if, assuming the precondition is true just before the function executes, then if the function terminates, the post-condition is true. The implementation is totally correct if additionally to partial correctness, the function is guaranteed to terminate and when it does, the post-condition is true.

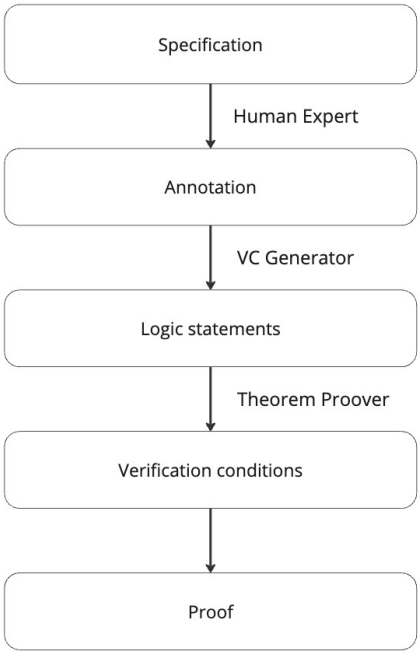

Fig. 3.1

In Dafny, these specifications are expressed as annotations within the code and are later translated into an intermediate verification language, Boogie, such that the correctness of this intermediate program implies the correctness of the original program[4]. Boogie is used to generate logical assertions or verification conditions that are passed into a satisfiability modulo theories (SMT) solver Z3 to automatically generate a proof[22]. It is worth noting that sometimes the verification might fail. Errors in Dafny can be attributed to two main reasons: either the provided specification is not consistent with the implementation or the SMT solver cannot automatically reach the required proof even though such a proof exists. Differentiating between these two is not generally trivial, however in the second scenario additional intervention by the developer is required to give additional context by

writing auxiliary conditions in the form of predicates, lemmas and functions.

```
method arraySum(a: array<int>, b: array<int>) returns (c: array<int> )
  requires a.Length=b.Length
  ensures c.Length=a.Length
  ensures forall i:: 0 ≤ i< a.Length=> a[i] + b[i]=c[i]
{
  c:= new int[a.Length];
  var i:=0;
  while i<a.Length
    invariant 0≤i≤a.Length
    invariant forall j:: 0 ≤ j< i=> a[j] + b[j]=c[j]
  {
    c[i]:=a[i]+b[i];
    i:=i+1;
  }
}
```

Listing 3.1: Example of a dafny specification

On Fig 3.1 we show an example of a Dafny program. The program defines a method, which corresponds to what might be called procedure or function in other imperative languages. In this case we define a method *arraySum* which takes two input arrays, a and b, and returns a new array c that contains the element-wise sum of the two input arrays. We can see that this Dafny method contains the following components:

1. Method Signature: The signature specifies the method name, the input and output variables along with their corresponding types. In this case the method named *arraySum* takes input variables (a and b) which are both arrays of integers *array<int>* and returns an output array c of the same type.

2. Preconditions: in Dafny preconditions are specified using the **requires** keywords. In this example the method requires that both input arrays have the same length (a.Length == b.Length), ensuring that element-wise addition is properly defined.

3. Post-conditions: in Dafny post-conditions are specified using the **ensures** keyword. In this case they signify that after execution, this method should guarantee that the length of the output array c is the same as that of the input arrays, and that for all indices i in the range from 0 to the length of the array, the sum of the elements from a and b at index i equals the element in c at the same index (c[i] == a[i] + b[i]).

4. Body: after the pre and post conditions the actual body of the method, in between braces, containes the actual implementation of the method. In this example the method initializes the output array c with the same length as the input arrays. Then a loop iterates over the indices, computing the sum for each corresponding pair of elements from a and b, storing the result in c.

5. Loop Invariants: The loop includes loop-invariants, also marked with the **invariant** keyword. These are conditions that should hold true on every iteration of the loop and are usually required to prove its termination. In this case, the first invariant asserts that the loop index i remains within bounds, while the second invariant asserts that all previously computed sums are correct (forall j:: 0 ≤ j < i ==> a[j] + b[j] == c[j]).

Dafny specifications can become much more rich and complex in practice, allowing the user to define boolean predicates, assertions and auxiliary lemmas to be used during the verification phase. It also allows to define a modifies clause to specify that given variable or field might be altered by a method during its execution.

In a similar spirit to Dafny, Coq and Lean are some of the most prominent languages used for formal verification, these are focused mostly around formal proof development. Coq is celebrated for its expressive type system, which permits the construction of mathematical proofs and interactive verification. Lean, designed for both theorem proving and programming, offers an approachable syntax while retaining powerful verification capabilities. Alternatives also exist for more popular programming languages like the Java Modeling Language (JML) for Java, which provides a way to specify program behavior but requires separate verification libraries.

It is worth noting that despite the notable progress in these verification tools achieved in recent years, the manual effort needed to write additional verification code can be considerable and difficult to master. Some studies suggest that formal specifications for the program under analysis often require as many tokens as the program itself [10].

3.3 Autoformalization

The task of auto-formalization involves transforming informal descriptions into some formally correct and automatically verifiable format [27]. By this we mean a format that adheres to a set of well-defined rules and principles, ensuring that it accurately represents the intended concepts and relationships without ambiguity. Ideally, it should be possible for this format to be evaluated by automated tools in order to verify its correctness.

In the software development process, developers often rely on documentation, typically in the form of informal natural language descriptions of a programs functionality and expected behaviour. While such informal descriptions are less precise than formal specifications and cannot be used automatically to verify the functionality of the system, they can still offer useful insights into a program's intended behavior. The recent advent of large language models has shown great promise to helping automate and bridge this gap between informal descriptions and formal specifications.

Recent studies have shown the capabilities of integrating large language models with proof assistants and formal verification tools in order to generate loop-invariants, assertions, and auxiliary lemmas for intermediate proofs. Some notable examples include Wu et al., who studied LLM-based auto-formalization on translating mathematical problems to formal specifications and proofs in Isabelle/HOL [35]. They found that LLMs could accurately translate a significant portion of a set of mathematical competition problems into formal specifications in Isabelle/HOL. Another noteworthy example is Mugnier et al., who introduced Laurel, a tool that leverages LLMs to generate helper assertions for Dafny programs to guide the SMT solver during the verification phase and showed that it could generate over 50% of the required helper assertions with only a few attempts [23].

We must note that auto-formalization remains a challenging task due to several reasons. Firstly, the system must ensure that the transformed text respects the syntax of the target language, not only this but the resulting expression must also adhere to other constraints such as the correct usage of types and dependencies used on expressions.

The more significant challenge lies in correctly capturing the meaning behind informal descriptions (the semantics) and being able to translate that meaning into a formal system. Natural language can be highly ambiguous, and the same sentence might have different meanings in different contexts. For example, when describing the behaviour of a sorting algorithm in an imperative language as "This method sorts an array of integers in ascending order", we are not specifying whether this could be modifying the original array (in-place) or creating a new copy of it (out-of-place).

Another challenge that arises in the developing such a system is domain knowledge. For example, the phrase "The function calculates the voltage drop across a resistor in a simple circuit given a voltage and resistor" implicitly assumes background knowledge in physics. To correctly formalize this statement one would need to recognize that the formula V = I * R (Ohms law), is needed.

4. DISTRIBUTION OVER THE SEMANTIC DOMAIN OF AUTO-FORMALIZATIONS

4.1 Auto-formalization with LLMs

The auto-formalization task can be thought of as the construction of a mapping between the elements of two sets. Let $\mathcal{L}$ denote the set of all natural language strings and let $\mathcal{A}$ be the set of all syntactically valid statements in a given formal language (i.e first order logic formulas or Dafny annotations). The goal of auto-formalization consists of constructing $f : \mathcal{L} \to \mathcal{A}$ mapping elements from $\mathcal{L}$ to elements in $\mathcal{A}$ [26].

The construction of such a mapping is complex due to two main challenges. Firstly, the ambiguity of natural language, that is, natural language sentences often contain ambiguities and nuances that can lead to multiple interpretations. This inherent ambiguity makes it difficult to determine a single, definitive formal representation for a given natural language string. Additionally, even when a particular meaning is clear, there may be multiple syntactically valid ways to express that meaning within the formal language. For example, in propositional logic, some simple transformations like adding double-negations or inverting the order in disjunctions leads to logically equivalent expressions.

Due to the probabilistic nature of neural language models, a mapping between the natural language and formal language domain should defined as a conditional probability distribution over the formal language domain. Given a natural language description $x \in \mathcal{L}$, a transfer model $T : \mathcal{L} \times \mathcal{A} \to \mathbf{R}$ induces a probability distribution over $\mathcal{A}$. Specifically, for any annotation $y \in \mathcal{A}$, $T(x, y)$ denotes the probability that x is transferred onto to y, defined as:

$$T(x, y) = \Pr(Y = y \mid X = x),$$

In practice, we do not have direct access to the full conditional probability distribution T for a given a natural language description x. Instead, we typically work with a model that, conditioned on x, generates outputs sequentially. This means that instead of directly sampling from a pre-defined distribution over all possible formal statements, we must rely on the model to produce outputs step by step.

For instance, in sequence-to-sequence models, the process of generating an annotation y can be seen as sampling one token at a time. At each step, the model generates the next token in the sequence based on the previously generated tokens and the original natural language description x, which serves as the conditioning context. The probability of generating the next token depends on the entire sequence of prior tokens, thus the model samples from a distribution over the next token at each step. This sequential generation process effectively approximates the process of sampling from the conditional distribution $T(x, y)$, but does not allow us to directly access the full distribution across the entire output space at once.

Both the natural language and formal language domain posses a structure between its elements that can be exploited in order to deal with the many syntactical variations of a

given statement. Specifically, an equivalence relation for the elements in $\mathcal{A}$ can be defined based on logical equivalence. That is, given two annotations $a_1, a_2 \in \mathcal{A}$ we say that these are equivalent, denoted as $a_1 \equiv a_2$, if and only if a_1 and a_2 are logically equivalent (i.e they have the same truth value for every model).

In practice this equivalence relationship between annotations can be evaluated using a theorem prover or SMT solver. For example consider the following program that takes as input an integer x and returns its absolute value y.

```
predicate is_abs_1(x: int, y: int)
{
  ( x≥0 => x=y) ∧ ( x<0 => x+y=0)
}

predicate is_abs_2(x: int, y: int)
{
  ( x≥0 => x=y) ∧ ( x<0 => y = −x)
}

lemma eq(x: int, y: int)
  ensures is_abs_1(x, y) ⟺ is_abs_2(x, y)
{
}
```

Listing 4.1: Equivalence checking using Dafny

Both predicates on 4.1 state that if x is non negative then the output y is equivalent to x. The first predicate then states that if x is negative then the sum of x and y should equal zero while the second one states that y should be equal to $-x$. The equivalence of these two predicates can be checked by writing a lemma that prooves their mutual implication.

In a similar fashion we can also define an equivalence relationship for the description domain whereby two natural descriptions are equivalent if and only if they are semantically equivalent. Unlike logical equivalence where this relationship is well defined, semantic equivalence for natural language expressions is less straightforward. This is due to the inherent complexity and variability in how meaning is conveyed and interpreted in natural language, which can depend on context and individual understanding. In practice, we can use a language model to approximate if two descriptions are semantically equivalent.

This structure in both domains $\mathcal{L}$ and $\mathcal{A}$ is crucial because it allows us to group annotations into equivalence classes based on their semantic meaning. This grouping not only simplifies the analysis by reducing redundancy but also ensures that we can focus on the meaningful semantic differences between annotations rather than their syntactic variations.

Using these equivalence relations, we can partition the domain into these classes of semantically equivalent elements. We use $e(\mathcal{A})$ to denote the set of equivalence classes in the annotation domain and use $[\ a]$ to denote the equivalence class of element $a \in \mathcal{A}$. Given a set of samples annotations from the model we can use a simple procedure to partition the domain into equivalence classes

Algorithm 1 Compute Equivalence Classes for Annotations

Input: Set of annotations $A = \{a_1, a_2, \ldots, a_n\}$, equivalence relation $E(a_i, a_j)$
Output: Set of equivalence classes $e(A)$

$e(A) \leftarrow \emptyset$ // Initialize an empty set of equivalence classes
for $a_i \in A$ **do**
  found $\leftarrow$ `false` // Flag to track if a match is found
  for each equivalence class $C_j \in e(A)$ **do**
    if $\exists a_j \in C_j$ such that $E(a_i, a_j)$ **then**
      $C_j \leftarrow C_j \cup a_i$ // If there is a match, add a_i to the existing class
      found $\leftarrow$ `true`
      break // Exit the loop, since a matching class is found
    end if
  end for
  if not found then
    $C_{new} \leftarrow \{a_i\}$ // If no matching class was found, create a new equivalence class
    $e(A) := e(A) \cup C_{new}$ // Add C_{new} to $e(A)$
  end if
end for
Return $e(A)$

On Algorithm 1 we show a procedure to compute equivalence classes in the annotation domain. It begins by initializing an empty set of equivalence classes. For each new annotation a_i, we check whether it belongs to any of existing equivalence class. For this, it compares the current candidate specification a_i against all previously formed classes using the equivalence relation $E(a_i, a_j)$. If we find that a_i is equivalent to a given a_j, a representative in an existing class C_i, then a_i is added to class C_i. If no match is found after checking all existing classes, a new equivalence class is created with a_i as the representative element. The algorithm continues this process iteratively, building up a set of equivalence classes that group equivalent annotations together. Finally, the result is a set of equivalence classes $e(A)$, where each class groups logically equivalent annotations.

4.2 Measuring accuracy and uncertainty in formalizations

There are several possible ways to measure the performance of an auto-formalization system. Most notably, a parallel can be drawn between formalization and another related task extensively studied in NLP research: machine translation. Both tasks involve transforming information from one form to another while preserving meaning and intent. Just as machine translation transforms text from one language to another, auto-formalization looks to translate informal mathematical language into formal specifications and proofs.

One performance metric, derived from the field of machine translation is the BLEU metric which essentially measures the accuracy of machine-generated translations by comparing them to human-generated references [25]. Previous studies have made use of this metric for this particular purpose, most notably Wu et al. [35] approached the translation of mathematical competition problems to formal specifications in Isabelle/HOL. The authors

used BLEU to measure the degree to which the output aligned with formal specifications crafted by a human expert.

However, we must note that BLEU as a performance evaluation metric in this context has a few problems. Firstly, the meaning in a sentence can change significantly just a few words or tokens. BLEU primarily focuses on string similarity rather than the actual meaning, which can lead to inaccuracies in evaluating the true quality of translations. This metric does not account for the various ways a sentence can be correctly translated. For example, changing the name of a quantifier or a non-free variable in a formula should not change its meaning, but this can be problematic for BLEU. Thus, BLEU might penalize such changes resulting in lower scores for valid translations that differ from the reference specification.

Alternatively, if one already has access to a reference specification $r \in \mathcal{A}$ for a given natural language description $x \in \mathcal{L}$ we can simply check whether its translation $T(x)$ is logically equivalent $T(x) \equiv r$ by using an SMT-solver to prove the statement $T(x) \iff r$. This approach would ensure that the generated specification is semantically identical to the reference, regardless of syntactic differences.

However, it is worth mentioning that SMT-solvers come with its own set of challenges and limitations. Most notably, SMT solvers can exhibit brittleness, meaning they can sometimes be sensitive to small changes in the input or problem formulation [30] . This brittleness can lead to inconsistent performance and fail to find a solution due to the complexity of some logical statements (such as those involving universal quantifiers), limitations of the solver itself or sometimes even due to seemingly irrelevant conditions like the naming of a variable. This in turn can lead to false negatives, where the solver incorrectly determines that two equivalent specifications are not equivalent. Additionally, while SMT solvers are powerful, they can struggle with very large or complex specifications, leading to longer computation times.

4.3 Confidence Measures

While accuracy measures how often a model's predictions are correct, it does not tell us how confident the model is about its predictions or how reliable those predictions might be in cases for which we don't have a ground truth specification. This is where confidence measures come into play. Confidence measures provide a numerical value that reflects the model's certainty in its predictions. These can be expressed as confidence scores, which typically range from 0 to 1, where 0 indicates no confidence and 1 indicates complete confidence. Confidence scores can be derived using various methods depending on the task and the model underlying model architecture and assumptions. For example, in classification tasks, confidence is often calculated as the probability assigned by the model to the predicted class $\hat{y}$ as $P(\hat{y}|x)$.

When dealing with sequence models, the task becomes more complex compared to binary classification because the output is a sequence of tokens rather than a single label and the probability of each token can be dependent on the previous one. For a sequence $\mathbf{y} = (y_1, y_2, \ldots, y_t)$, where T denotes the length of the sequence, the probability of generating the entire sequence according to the model is:

$$P(\mathbf{y}) = \prod_{t=1}^{T} P(y_t|y_1, y_2, \ldots, y_{t-1}) \tag{4.1}$$

In practice, log-probabilities are used since they simplify computations and help avoid numerical issues. The log-probability of the sequence is then given by summing the log-probabilities of each token conditional on the tokens that came before it:

$$\log P(\mathbf{y}) = \sum_{t=1}^{T} \log P(y_t|y_1, y_2, \ldots, y_{t-1}) \tag{4.2}$$

However, the log probability of a sequence is not always a direct reflection of confidence, especially in tasks where the output is a sequence of tokens rather than a single label. In this context, entropy can be a useful complementary measure. In information theory, entropy is a measure of uncertainty or randomness in a distribution. It quantifies the amount of unpredictability or surprise associated with a set of possible outcomes. For a discrete random variable X with possible outcomes $x_1, x_2, \ldots, x_n$ and probabilities $p(x_1), p(x_2), \ldots, p(x_n)$, the entropy $H(X)$ is defined as:

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log_2 p(x_i) \tag{4.3}$$

where $p(x_i)$ is the probability of outcome x_i, and the logarithm is typically base 2, measuring entropy in bits. When the distribution of outcomes is spread out more evenly, meaning no outcome is highly probable compared to others, the entropy is higher. This indicates greater uncertainty or unpredictability because each outcome is nearly equally likely. Alternatively, when one outcome is highly probable while others are less so, the entropy is lower. This reflects a situation where the outcome is more predictable because there's less uncertainty about which outcome will occur.

To compute entropy as a measure of uncertainty in a sequence model's predictions, we would need to generate multiple samples or predictions from the model. Kadavath et al. studied whether language models can evaluate the validity of their own claims and predict which questions they will be able to answer correctly [19]. To this aim they sampled multiple answers for a set of questions and estimated the entropy of the answer distribution as

$$H(A|Q) = \mathbb{E}_A \left[\sum_{i=1}^{N} (-\log P(a_i|Q, a_0, \ldots, a_{i-1})) \right] \tag{4.4}$$

However, they found that as models increase in size, and capable of providing more diverse set of correct answers the entropy of the token sequences as they measured might not be a robust measure. Farquhar et al recently explored an alternative approach aimed at detecting hallucinations in large language models [11]. Their method addressed this limitation of the previous entropy based approaches by computing entropy at the semantic level.

To estimate semantic entropy, first multiple samples of the sample problem must be generated. After this, the outputs generated by the model are clustered into groups of sentences that convey the same meaning. This can be done by defining an equivalence relationship $\equiv_s$ where two elements are the same if they are conveying the same idea in a particular context. In practice this can also be approximated using an language model. Then, the likelihood that a sequence produced by the model belongs to a particular class is given by summing the probabilities of all possible token sequences that can be interpreted as expressing the same meaning.

In practice this quantity is not tractable due to the fact that only a limited number of classes can be sampled from the model. This would mean that

$$SE(A|Q) = \sum_{c_i \in C} P(c_i|Q) \log P(c_i|Q) \tag{4.5}$$

where $P(c_i)$ determines the probability of observing an answer from the semantic cluster c_i given a question Q. In practice this can be approximated by $P(c_i|Q) = \frac{1}{|S|} \sum_{x \in S} 1_{x \in c_i}$ where S is the set of all outputs sampled from the model. This is particularly convenient, in our case since the log-probabilities were not available on the Gemini API (AI Studio) at the time of writing, thus we used the discrete approximation of semantic entropy were we approximate the probability of each equivalence class using the observed frequencies.

5. EXPERIMENTS AND RESULTS

In this chapter, we present an empirical evaluation of an LLM-based approach to autoformalization with a focus on addressing the research questions outlined in the introduction. The first two sections describe the general architecture and the dataset used throughout our experiments. We then evaluate 3 different strategies over a collection of programs, measuring both alignments (i.e., was the generated solution equivalent to the reference specification) and concentration (whether the largest equivalence class dominates the distribution) for the generated formalizations.

To explore what happens in cases of concentration and non-alignment, we analyze and illustrate instances of misalignment, particularly examining the most frequently occurring equivalent classes for each proposed architecture. This analysis looks to address the nature of misalignment but also serves to inform the restructuring of the prompt-based solution on later experiments.

In the final sections, we investigate the relationship between alignment and concentration through the lens of semantic entropy, aiming to analyze the if there is a relation between these two phenomenons. Additionally, we explore the performance of the inverse task (turning formal specifications into informal descriptions) to determine whether it could serve as an indicator of the overall performance of the model on a given program.

5.1 The CloverBench dataset

For our experiments we use an existing dataset of Dafny programs. The CloverBench dataset consists of 62 small hand-written example programs similar to those found in standard computer science textbooks [26]. Each program in the dataset consists of a single method and includes a Dafny implementation, annotations specifying pre-conditions and post-conditions, and a docstring that describes program.

```
0  //Find a key in a possibly empty array. Return the index of its first occurance.
1  //Ensure index is in range. If not found, return -1.
2  method Find(a: array<int>, key: int) returns (index: int)
3    ensures -1 ≤ index < a.Length
4    ensures index ≠ -1 ⇒ a[index] = key ∧ (forall i :: 0 ≤ i < index ⇒ a[i] ≠ key)
5    ensures index = -1 ⇒ (forall i::0 ≤ i < a.Length ⇒ a[i] ≠ key)
6  {
7    index := 0;
8    while index < a.Length
9      invariant 0≤index≤a.Length
10     invariant (forall i::0 ≤ i < index⇒a[i] ≠ key)
11   {
12     if a[index] = key {
13         return;
14     }
15     index := index + 1;
16   }
17   if index ≥ a.Length  {
18     index := -1;
19   }
20 }
```

Listing 5.1: Example of a dafny specification

In the example above we show one of the programs found on the CloverBench dataset. The first two lines that make up the documentation outline the goal of finding a key in an array and returning its index or -1 if the key is not present. The specification section defines the formal constraints which in this case first ensure that returned index is within bounds and then specify that if the returned index is different than -1 then the index correspond to the first occurrence of the key in the array and finally that the index returned should be -1 if the key is not present in the array. Finally there is also a body with the implementation of the method where a loop goes through all the indexes in the array until a matching value is found.

5.2 Transfer model implementation details

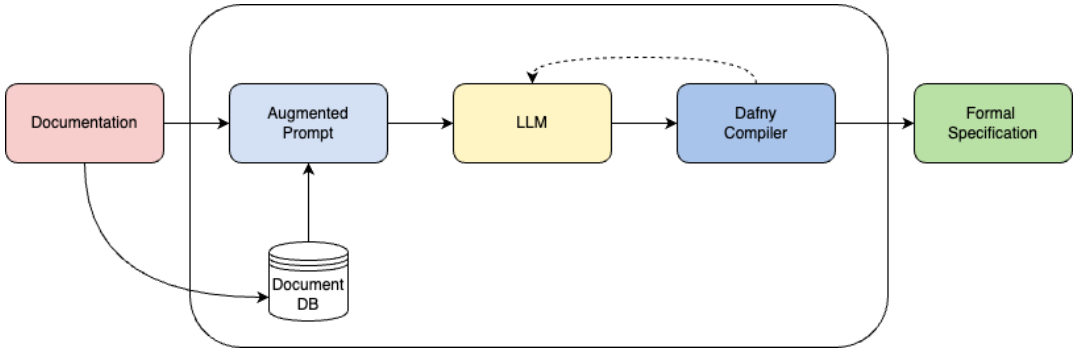

Fig. 5.1: General architecture used throughout the experiments.

The experiments in this thesis share a common architecture outlined on Figure 5.1. The initial step involves generating an initial few-shot prompt candidate from the natural language description text and method signature from the target program. This few-shot approach augments the system instruction and target documentation with other relevant examples in order to give a more comprehensive context to the LLM on the type of solutions that are expected.

The examples we use to use in the few-shot prompt are retrieved from a document database stored in memory and are selected based on their semantic similarity to the target documentation. This essentially consists of converting text into a numerical representation known as embeddings. Embeddings are dense vector representations of text, where semantically similar words and/or texts have closer representations in the target space when measured using a distance metric such as cosine similarity [21]. Identifying vectors that are close to each other in this space allows us to retrieve examples from the database that are semantically similar and contextually relevant to the target program. Throughout our experiments, we initialize the document database with the other programs in the CloverBench dataset, and we retrieve a subset $k = 10$ programs. We exclude the target program to avoid leaking information from the reference specification in our tests.

Next, after constructing the few-shot prompt we query an LLM. In this case, we use the Gemini family of models, made available through the Gemini API [13], [28]. Some of the models available in the Gemini API at the time of writing include *Gemini 1.0*, *Gemini 1.5*

Flash and *Gemini 1.5 Pro*. We opted for the *Gemini 1.5 Flash* model for our experiments, as the other models in the Gemini family were subject to rate limiting at that time. This model allows us to provide a set of parameters like temperature, the maximum number of tokens to sample and the top_p parameter for controlling the tokens used in nucleus sampling. We limited the maximum number of tokens to 1000 and used a temperature parameter of 0.7 throughout our experiments.

After receiving the response from the model, we check use the Dafny compiler to check for syntax and type errors. If any syntax or type errors are detected during compilation then a retry mechanism is used in which we query the LLM augment the initial prompt with the response and compilation error log from the compiler.

5.3 Baseline formalization

First we want to assess the consistency and accuracy of specifications generated by the LLM and the properties of the underlying distribution when looking at the set of generated equivalence classes. We want to look at how consistently the model generates specifications that fall within the same logical equivalence class. Additionally we want to determine if there is an association between concentration and alignment between the generated and reference specification. We also want to determine if there are observable biases or common patterns in the model's output for misaligned specifications.

In order to do so, for each program in the dataset, we prompted the model to generate a specification from the original documentation. We sampled 30 specifications for each program and then grouped the generated specifications into equivalence classes based on their logical equivalence.

Prompt template 1

SYSTEM INSTRUCTION
You are an expert in Dafny. Given the function's docstring and signature for a Dafny program, please extract the pre and post-conditions for the program provided. Please make the pre-condition as weak as possible and the post-conditions as strong as possible. Put one condition per line. Do not return the docstring and the function implementation. Do not use helper functions. Use abs for absolute value. Do not explain. Follow the format:

DESCRIPTION: docstring

SIGNATURE: method signature

SPECIFICATION: the program specification in Dafny

DESCRIPTION: Find a key in a possibly empty array. Return the index of its first occurrence. Ensure index is in range. If not found, return -1.

SIGNATURE: method Find(a: $array < int >$, key: int) returns (index: int)

SPECIFICATION:
$ensures -1 <= index < a.Length$
$ensures\ index \neq -1 ==> a[index] == key$
$ensures\ index \neq -1 ==> forall i :: 0 <= i < index ==> a[i] \neq key$
$ensures\ index = -1 ==> (forall i :: 0 <= i < a.Length ==> a[i] != key)$

In this case, a system instruction is provided to the language model describing the instructions for the task. Specifically, the LLM is instructed to formulate the weakest possible preconditions and strongest possible post-conditions for the method, excluding both the docstring and implementation from the response. The instructions also outline the format for the interaction, where the description is followed by the method's signature and after this, a specification following the Dafny syntax is expected. The example above demonstrates how this interaction would occur when querying the language model. In this example, we begin by providing a description and the signature for the program *find*. A possible output from the language model for this program is shown in blue below and contains the expected specification for the respective method in the Dafny syntax.

On Table 5.1, we show a breakdown of the solutions generated by the language model based on alignment and concentration. The rows indicate the type of alignment with the reference specification, categorized as *Aligned*, *Not Aligned* and *Strongly Not Aligned*. The first category, *Aligned*, corresponds to programs for which specification of the largest equivalence class coincides with the reference specification. On the second category, *Not*

Alignment/Concentration	Concentrated	Not Concentrated
Aligned	45	3
Not Aligned	3	3
Strongly Not Aligned	4	4

Tab. 5.1: Alignment and concentration of the most frequent equivalence class

Aligned, we include programs for which a specification equivalent to the reference specification was found but this one was not its majority class. On the third category, *Strongly Not Aligned* we include programs for which none of the specifications generated were aligned with the reference specification. The columns represent the concentration, with *Concentrated* indicating that a solution holds more than 50% of the weight within its logical equivalence class, signifying dominance with respect to the other generated specifications. Conversely, *Not Concentrated* solutions have less than 50% weight, indicating they are overshadowed by other incorrect or misaligned specifications.

We observed that, for 54 out of the 62 programs the LLM was able to generate a correct solution, that is, it was able to generate at least one specification which could be proven to be equivalent to the reference specification.

After grouping solutions into their respective logical equivalence classes we found that for 45 programs the equivalence class corresponding to the reference specification concentrated the majority of the weight of the distributions. For the other 9 programs, even though a solution was found, it was outweighed by another classes corresponding to mis-aligned solutions that concentrated a larger proportion of the distribution's weight.

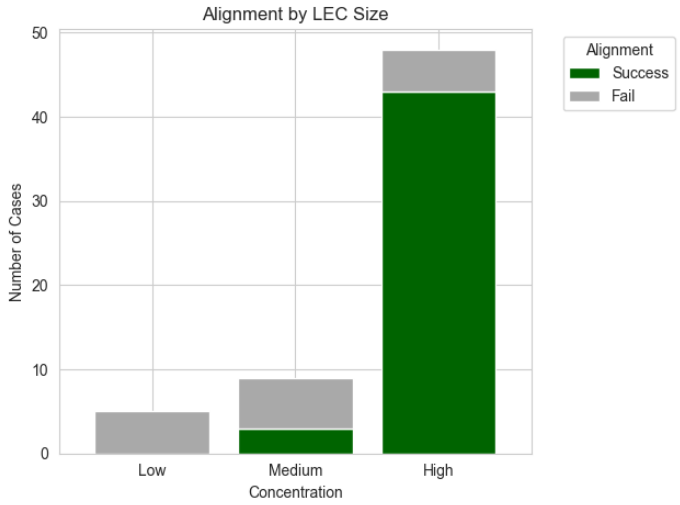

Fig. 5.2: (Left) Bar plot for the equivalence to reference specification by the frequency of the largest equivalence class.

In Fig 5.2, we show the results of this experiment after grouping the generated specifications according to their logical equivalence. The plot shows the distribution of problems

by the size of their most frequent equivalence class. The x-axis represents the number of elements within our sample that fall into the most frequent equivalence class. Additionally, the plot highlights in green and gray the proportion of programs for which the largest equivalence class was determined to be equivalent to the reference specification.

This results reveal that for most of the problems present in the dataset the model generates specifications which fall within the same logical equivalence class. Moreover, for 52 out of the 62 programs examined we see that there is a single equivalence class which captures the majority of the empirical distribution's weight. This means that when looking at the model's outputs, most of them fall into one specific logical equivalence class rather than being spread out among multiple options. We can interpret this as the model having a fairly high degree of confidence in these cases. Additionally, there appears to be a correlation between the concentration and the overall alignment. In other words, when the model's generated specifications are heavily concentrated within one equivalence class, those specifications are more likely to align with the reference specification. This would suggest that when the model is confident enough to produce specifications that cluster together, those specifications are typically a more accurate representation original intention in the documentation. We can interpret this as evidence of the model having a high degree of confidence in its responses for the majority of the problems presented here.

We also found some examples for which the output distribution is concentrated over a single incorrect equivalence class. For example, a problem in the dataset consisted on searching for an element in a possibly empty array and return the index of its first appearance. After grouping the specifications generated by the LLM we observed that these fall into two groups, with one of them concentrating over 83% of the distribution. The first one was classified as incorrect due to the missing post-condition requirement that the index returned should coincide with the first appearance.

Verification Failures	**Concentrated**	**Non Concentrated**	**Total**
Weak post-conditions	3	3	6
Incorrect Post	2	2	4
Syntax error	1	2	3
Weak pre-conditions	1	0	1
Total	7	7	14

Tab. 5.2: Nature of non-alignment for most frequent equivalence clusters.

On the right side of Fig 5.2 we present a table that details the breakdown of nature of non-alignment identified in the most frequent logical equivalence class for each problem. We observed that the most frequent non-alignment reason could be attributed to the post-conditions that were too weak compared to the reference specification. For example, on the program *LinearSearch* where the task consists of linearly searching for an element in a possibly empty array and return the index of its first appearance the generated examples consistently failed to include the post-condition that the returned index is indeed the first appearance ($\forall 0 \leq j < |a| : j < i \rightarrow a[j] \neq e$). Interestingly a similar statement did appear in the specification generated by the language model but it was conditional to the item

not being found. That is, here the model might be generating post-conditions that are syntactically plausible or commonly seen in similar tasks, but not semantically aligned with the specific program's requirements. This misalignment might happen because the model doesn't always grasp the full logical structure of the problem.

Additionally, we found programs with incorrect post-conditions among the instances of mis-aligned and concentrated solutions. For example, on one program which the task was to generate a map r by transforming the keys from an input map m using the an injective function $f : \mathcal{N} -> \mathcal{N}$. On the class that concentrated over 50% of the distribution we found the model was consistently generating specifications that contained the statement $\forall k \in \mathbf{Nat} : \neg(k \in m) \implies \neg(k \in r)$, meaning that if a key is not in the original array then it should not be present in the result/output array. The correct statement could actually be $\forall k \in \mathbf{Nat} : \neg(k \in m) => \neg(f(k) \in r)$ where if the element is not in the array then, its value after f, $f(k)$ should not be in the resulting array.

We also identified three instances where the most frequent equivalence class contained syntax errors, leading to verification failures. Finally, after manual inspection, we discovered two programs where the most frequent equivalence class was indeed correct and syntactically valid, but the verification procedure failed to verify that this specification was equivalent to the reference specification. For our analysis, we counted those specifications as correct. This is interesting since it underscores the limitations of the SAT solver approach in accurately determining equivalence, showing that these failures can occur even for this set of seemingly simple programs.

5.4 Introducing an Intermediate Verbalization Step

In the previous section, we observed that weak post-conditions were a primary source of error which suggests that the model may struggle to fully capture and evaluate the local structure and relationships between the variables involved in the program. Recent studies show that breaking down tasks into a series of intermediate reasoning steps can significantly improve the ability of large language models to perform complex reasoning [32]. A priori, we expect that this intermediate verbalization step could promote self-consistency by encouraging the model to first articulate and consider the multiple aspects of the program involved. Essentially, this approach attempts to mimic aspects of our intuition on how humans might approach this task by first decomposing a problem and considering the entities involved in it and their interactions. For this experiment, we test whether this intermediate decomposition of the source documentation into a more specific natural language description leads to an improvement in the performance of the auto-formalization capabilities of the LLM.

Prompt

DESCRIPTION:
Find a key in a possibly empty array. Return the index of its first occurrence. Ensure index is in range. If not found, return -1.

SIGNATURE:
method Find(a: array< *int* >, key: int) returns (index: int)

VARS:
a: array of integers, representing the array to be searched
key: integer, representing the key to be found
index: integer, representing the index of the first occurrence of the key in the array, or -1 if the key is not found.

THOUGHT PRE:
There are no preconditions

THOUGHT MODIFIES:
The method does not modify any of the input parameters.

THOUGHT POST:
If the key is found, the returned index should be the index of the first occurrence of the key in the array. If the key is not found, the returned index should be -1. The returned index should be within the range of the array, or -1.

To this end, we modify the system instruction to elicit a response that begins with a description of each variable involved. Following this, the response should describe in natural language the pre-conditions and post-conditions and also consider whether the method alters the input variables. Finally, we instruct the LLM to express these conditions as Dafny statements (requires, modifies, ensures). Above we show an example for a program in the dataset. In this example, the task consists of finding a key in an array and return the index of its first occurrence. We can see that the variables involved are the input variables *a* and *key* which correspond to the input array and the key to be found in the array, and the output variable *index* which corresponds to the index of the first occurrence of the key in the array.

Alignment/Concentration	Concentrated	Not Concentrated
Aligned	38	5
Not Aligned	4	5
Strongly Not Aligned	4	6

Tab. 5.3: Alignment and concentration breakdown

An analysis into the resulting distribution of the specifications generated by this variant appears to yield a degradation of both the overall accuracy and concentration. We observed a drop in the number of problems for which a solution is found to 51. Most notably, on table 5.3 we see that after grouping the specifications by their logical equivalence the number of problems for which solutions were concentrated into a class aligned with the reference specification drops from 46 to 38.

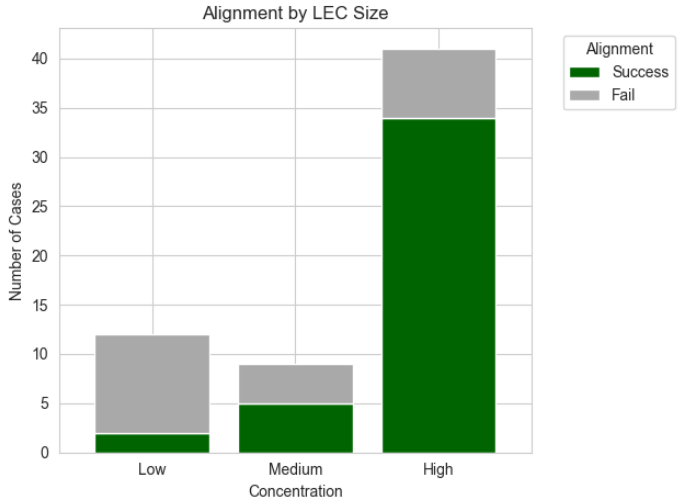

Fig. 5.3: Bar plot for the equivalence to reference specification by the frequency of the largest equivalence class.

When delving deeper at the reasons that lead to the non-aligned solutions we observe that there is a higher frequency of specifications that contain stronger than required pre-conditions. Upon manual examination we can see that most of the errors here stem from the verbalization process. Interestingly, we observed that for 6 of these programs with strong pre-conditions, their intermediate decomposition step contains a pre-condition which requires non-empty arrays on problems which do not have this explicit requirement. For example, on the program *replace* where the task consisted on replacing all elements in an array strictly greater than an input value k with -1 while elements in the array less than or equal to k remain unchanged we found that the intermediate decomposition contained the statement *The input array should not be empty* or which resulted in a pre-condition statement requiring $arr.Length > 0$. This example also highlights some of the complications that arise from what could be considered and ambiguous prompt and the model making incorrect assumptions, since the specification could indeed be verified correctly and represent a feasible implementation, the requirement for the input array to be non-empty is not mentioned anywhere in the program description.

Verification Failures	**Concentrated**	**Non Concentrated**	**Total**
Too strong preconditions	5	2	7
Weak postconditions	1	3	4
Incorrect Post	1	3	4
Weak preconditions	1	0	1
Too strong postconditions	0	1	1
Syntax error	0	2	2
Total	8	11	19

Tab. 5.4: Verification failure reasons breakdown.

We think that the higher frequency of stronger pre-conditions observed in this experiment could be caused by a bias introduced on the prompt itself. Studies have shown that LLMs

can be susceptible to a phenomenon often refereed to as prompt-bias [36]. Essentially, the way a prompt is structured or presented to the model can lead to responses that are biased towards certain types of outputs, regardless of whether they are appropriate or necessary for the task at hand. In this particular case, the explicit mention of *preconditions* in the prompt may have led the LLM to focus on including them in the generated specifications, even when they were not required or necessary on the correct solution. In essence, we think that the model might have been "primed" to include preconditions as part of its output, even if the problem itself didn't require such conditions.

5.5 Intervened Verbalization

As our results from the previous section showed, a significant number of the model errors seem to be attributable to an incorrect interpretation of the original problem descriptions. To examine this, we manually intervened the process used in the previous experiment by feeding the the same prompt from the previous section but this time with the correct natural language decomposition included in the context. We expect this manual intervention to reduce the propensity of the model to hallucinate and steer the answers towards the original intentions of the original documentation. Like before we sampled 30 specifications for each program and verified their outputs and later grouped them into their respective equivalence classes.

Alignment/Concentration	**Concentrated**	**Not Concentrated**
Aligned	56	1
Not Aligned	0	0
Strongly Not Aligned	3	2

Tab. 5.5: Alignment and concentration with pre-filled prompt

On table 5.5 we show the contingency table for alignment and concentration. We can see that for 57 out of the 62 programs the specifications generated were aligned with the reference specification. We found one program for which the correct specification was found but the distribution was not concentrated on its respective equivalence class mostly due to a around 80% of the distribution being dominated by specifications with type errors.

We found only 5 programs for which a specification aligned with the reference specification could not be found. Most notably we still found instances for which the model generated specifications that were clustered around a class not aligned with the reference specification. This included for example a case where the program consisted on modifying the value at one specified index of a 2D array of natural numbers and even the verbalization included the statement *Every element in the 2D array arr remains unchanged, except for the element at position (index1, index2). This means that any element (i, j) in the array with i != index1 and or j != index2, will retain its original value.* However, the generated specification only guaranteed this condition for the modified inner array, that is $arr[index1][j] == old(arr[index1][j])$.

In essence, we could think of the formalization task here as the LLM attempting to complete two related tasks: first, it needs to accurately interpret and decompose the problem,

and second, it must translate this understanding into a formal logical specification. This increase in performance after the intervention suggests that the source of the errors lies primarily in the first sub-task (interpreting and decomposing problem), and that by providing more refined natural language descriptions significantly reduces the likelihood of hallucinations and results in specifications that better align with the original intent of the documentation.

5.6 Relation between entropy and performance

In the previous experiments, we observed that there appears to be a relationship between concentration and performance for different transfer model instances. One particular way of quantifying the concentration for a distribution is through its semantic entropy. In this case, we believe that answers from a large language model (LLM) that exhibit lower entropy values should correspond to answers on which the model is confident which in turn could have a higher likelihood of being aligned with the original documentation. Note that it is also possible for outputs to be concentrated among incorrect or misaligned answers. In such cases, the model's confidence is misplaced on plausible but incorrect answers, leading to systematic errors that are consistently misaligned with the intention conveyed in the original documentation.

To investigate this relationship further, we measured the discrete entropy and accuracy on each strategy. We computed the accuracy on each example as the proportion of correctly generated specifications (equivalent to the reference specification) relative to the total number of specifications produced. On figure 5.4 we show both the average entropy and the accuracy for each of the different strategies that we tried.

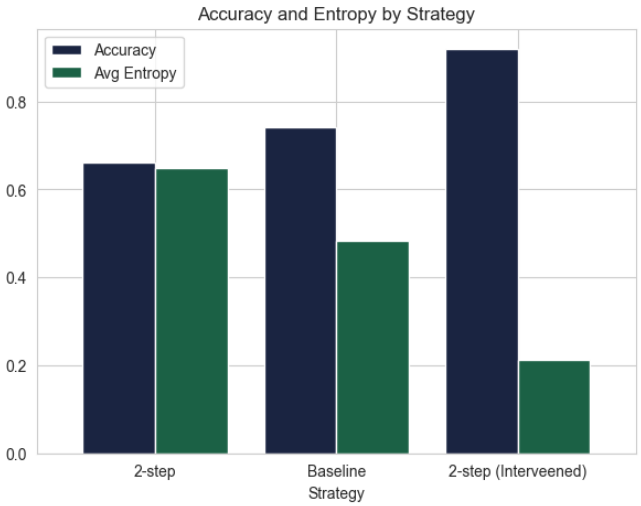

Fig. 5.4: Mean Accuracy and Entropy for each of the evaluated prompting strategies.

We can see that the verbalization instruction in this case has an accuracy of about 65%, the baseline strategy 78% and after the intervention on the verbalization strategy achieves an accuracy 92%. This would be indicative that as the overall performance for this task increases so does its average concentration and consequently the entropy of the output distribution is lower.

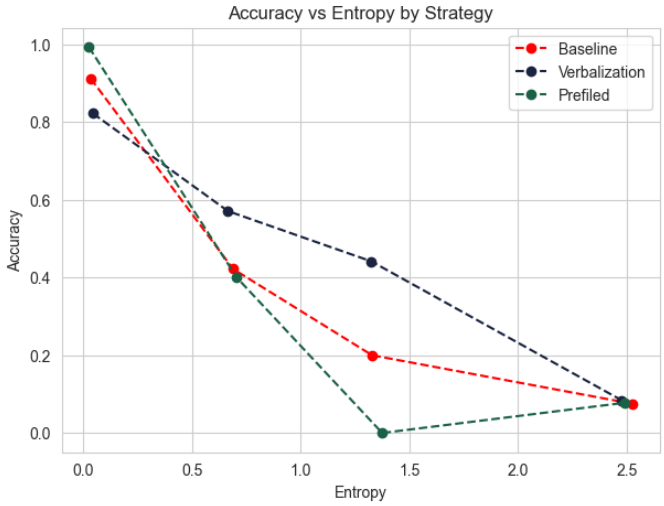

Fig. 5.5: Mean Accuracy and Entropy for each of these strategies

On figure 5.5 we show the accuracy for different entropy levels for each strategy. Here we can see that within a strategy a similar pattern emerges where lower entropy values are associated with a higher accuracy. Other studies have also shown similar relationships between the entropy of a language model and its calibration [11], [19].

5.7 Abstraction from reference specification

Next we explore the capabilities of language models to perform the inverse task. This task is also referred to as abstraction [6] and the goal consists of generating a natural language description or abstraction of a formal object (in our case the program's specification). Other research works also explore this bi-directional approach. For example, Clover [26] makes use of this method to try and verify whether generated code and its documentation are consistent with each other. Similarly Allamanis et al. [1] explore an approach on this direction by defining Round-Trip Correctness as alternative evaluation metric for code generation tasks by ask a model to make a natural language description of the code, feed that description back and check if this round-trip leads to code that is semantically equivalent to the original input.

In this case we want to know whether the performance on this task for a program could be indicative of the performance on the original formalization task for a given program. We would expect for this this metric to show us programs where either the abstraction could contain multiple possible interpretations thus carrying to different text, or additionally programs for which their complexity is high enough that the LLM fails to capture subtleties in its specification.

To this end, we provide the method signature along with its specification in the context and ask it to generate a natural language description. Then to verify whether the generated specification matches the original specification we prompt the LLM again providing both the reference and generated descriptions and ask it to asses whether this two specifications are semantically equivalent. We sampled 30 descriptions from the reference specification

of each program in our dataset following the prompt template shown below and measure its accuracy.

Abstraction Task Instruction

Given the function signature and its specifications for a Dafny program. Please return a short and concise docstring of the functional behavior implied by the specifications. Do not mention implementation details. Please only return the description. Do not explain. Below is the Dafny signature and its specifications:

SIGNATURE: method Find(a: $array < int >$, key: int) returns (index: int)
SPECIFICATION:
$ensures -1 <= index < a.Length$
$ensures\ index \neq -1 ==> a[index] == key$
$ensures\ index \neq -1 ==> foralli :: 0 <= i < index ==> a[i] \neq key$
$ensures\ index = -1 ==> (foralli :: 0 <= i < a.Length ==> a[i]! = key)$

For this task, the LLM was able to generate at least one correct description for all of the programs in our dataset, but with a varying degree of accuracy. The average accuracy sits at 92% and for about 49 of them it was able to get a correct description in all instances, that is, with an accuracy of 100%. Interestingly, there were only 3 programs for which its accuracy was below 50% and for these programs all three approaches failed to generate a correct formalization as well.

5.8 Limitations and Threats to Validity

We must note that there are some factors that may compromise the validity of our findings and its implications. One significant threat to the validity of our findings arises from the fact that the specifications used in this study are assumed to be accurate and reflective of the intended program behavior. These specifications can be subjective in nature, and their correctness and relevance may vary depending on the context in which the programs are implemented. Since specifications might not be universally objective in all cases, there is a risk that they may not fully capture the intended functional requirements, potentially introducing biases in the analysis.

Furthermore, several of the programs may be too elementary or trivial, and therefore not representative of the complexity typically encountered in real-world software programs. As a result, we are not able to extrapolate the findings to broader software engineering contexts.

Another potential threat stems from the uncertainty surrounding the training data used for the language model. It is not clear whether the model was exposed to prior specifications of similar programs written in Dafny, especially for common or well-known programs. If the model had access to such instances, it could have biased the generated specifications, possibly making the results less generalizable or skewing them toward already familiar patterns from its training corpus. This dependency and lack of transparency in the training corpus complicates the interpretation of the results, as the model's output might not solely reflect its inherent capabilities, but rather in its exposure to previously seen examples.

Finally, a fundamental limitation of this study comes from the fact that the sample size used when generating specifications is fairly small and this might limit the diversity of the generated samples. Additionally, the experiments were constrained to a single set of parameters and a single model.

6. CONCLUSIONS

We explored the use of Large Language Models (LLMs) for auto-formalization, a task that consists of turning natural language descriptions into formal specifications. To assess the effectiveness of LLM-based solutions, we made use of an existing dataset of Dafny programs, together with their documentation and formal specifications. Our analysis was focused not only on the model's overall accuracy, but also on a qualitative understanding these solutions in terms of their semantic distribution. We evaluated how the generated specifications could be grouped based on logical equivalence using Dafny's SMT solver, and analyzed the semantic entropy of the resulting distributions. This allowed us to move beyond simple syntactic comparisons and gain a deeper understanding of how the model's outputs varied and clustered semantically.

Our results revealed that the performance of the LLM-based approach varied significantly across different prompting strategies. The baseline method, which directly generated preconditions and postconditions, showed generally acceptable performance in terms of distributional concentration—that is, the model's tendency to produce similar outputs. However, we also identified areas of poor alignment, especially related to the strength of the generated preconditions. The most frequent failure in this approach was due to the model producing preconditions that were too weak or too general. This observation led us to propose a more complex strategy involving an intermediate verbalization step, which led to a performance degradation. A closer inspection of the semantic clusters of the resulting formalizations showed that the outputs tended to concentrate around specifications with stronger preconditions.

After an intervention of the transfer model with correctly decomposed natural language descriptions, breaking down the informal descriptions in a way that would help the model better capture the logical structure of the problem we saw a significant improvement both in terms of accuracy and concentration. The improved clustering of the generated formalizations around correct specifications suggested that the root cause of earlier failures lay in the model's initial "interpretation" of the natural language input. When given more precise and structured descriptions, the model was better able to generate consistent and reliable formal specifications, with significantly fewer errors or hallucinations. We believe that this qualitative analysis of performance in terms of distribution and clustering, particularly in a clustered support where outputs can be analyzed semantically, seems to provide valuable insights for understanding the challenges and limitations of LLMs in the auto-formalization task.

Additionally, our analysis of the model's semantic entropy—which reflects the semantic distribution of generated outputs—provided a valuable indicator performance. Lower entropy, corresponding to higher output concentration, was correlated with higher-quality formalizations. We also explored the inverse task of generating natural language descriptions from formal specifications and found that performance on this task was appears to be correlated with performance on the formalization task itself.

We must also note that our results seem to align with recent studies. Like Sun et al., we observed that language models generally perform well when tasked with generating formal

statements from natural language descriptions [26]. Our exploration of semantic entropy also coincides with previous investigations that suggest this metric effectively captures instances where the model's uncertainty is linked to inaccuracies. However, unlike previous studies, our work makes use of semantic clustering in the formal domain and we conduct a more in-depth analysis of the underlying causes of mis-alignment that arise during the formalization process and with instights that we believe can help guide enhancements in LLM-based solutions for the auto-formalization task.

Looking ahead, we think that several avenues warrant further investigation. One of the more pressing ones would be to expand the analysis to a more comprehensive dataset of Dafny programs. Most of the programs in this dataset are small programs that one might find in an algorithm's text book and thus might not be representative of the types of programs one modern software projects. It would also be interesting to expand this analysis beyond the formalization of single methods, for example, modeling abstract data types and their structural invariants.

We also think it would be interesting to test the effects of some handmade adversarial prompts, purposefully built to introduce ambiguity into the desired specification. This could also involve a comparison on how LLMs respond to these adversarial prompts in contrast to a human participant. Some recent studies have shown that that LLMs are likely to be misled by irrelevant information thus exploring how such information affects the models accuracy and confidence could be a valuable research direction for future research.

Furthermore, this analysis relied exclusively on the Gemini family of models. However, with an ever increasing number of language models being released to the public, and each with potentially new architectures, training data, and parameter sizes, it remains an open question whether our results carry over to these other models. More specifically, it is still unclear whether auto-formalization is an emergent capability that becomes possible only after reaching a certain model size or complexity. As we continue to see rapid advancements in model development, it will be interesting to explore how this ability manifests across different architectures and scales.

Additionally, we believe it would be worthwhile to explore the possibility of fine-tuning a pre-trained model for the autoformalization task using the set of programs descriptions and their specifications from the dataset. Furthermore, we consider that a promising avenue for future research could involve evaluating the possibility of using semantic-entropy, or other related measures of statistical concentration, as unsupervised metrics for performance evaluation during the fine-tuning process. Entropy, in particular, could serve as a powerful tool for assessing the uncertainty and diversity of a model's outputs, helping to guide model adjustments and further optimize its performance, particularly when a supervised signal might be limited to a subset of reference examples.

Finally, during our experiments we considered mainly one single specification as a ground truth, but in reality the correspondence might not be so simple. In practice, a single natural language description can be inherently ambiguous, potentially describing multiple, distinct formal interpretations. This complexity suggests that a more nuanced approach is needed to account for these ambiguities. We believe it could be valuable to explore

a more generalized metric of accuracy that could accommodate these scenarios where multiple alternative solutions could be deemed correct. Potentially, this could include the development of a distance metric over the set of possible specifications.

6.1 Related Work

There are some works that look into calibration or confidence estimation in large language models. Among these is a recent study relevant to the calibration and confidence estimation on large language models answers done by Zhang et. al [37]. They investigate this in the context of root-cause analysis for cloud infrastructure incidents and develop a system, PACE-LM, in which a large language model is queried with the instruction to explain the root cause of an incident together with a confidence score and a confidence evaluation. They proposed a way to combine this two scores into a well-calibrated confidence estimate for the reliability of the diagnostics proposed by the language model.

Finally, several recent papers explore the integration of large language models into the formal verification framework. Mugnier et. al [23] propose "Laurel", an LLM based approach to assists developers in generating helper assertions for the verification of Dafny lemmas for which the internal SMT-solver may struggle. They evaluated their approach and showed that it was able to generate over 50% of the required helper assertions given only a few attempts.

Finally, another noteworthy attempts in the generation of verifiable computer programs using Dafny is Brandfonbrener et al. who introduces a novel method for generating verifiable code in Dafny using Monte Carlo Tree Search [7]. Their approach uses a language model to generate partial versions of a target program and construct a value function on these partial programs using the feedback from the Dafny verifier which allows for the systematic exploration of possible solutions.

BIBLIOGRAPHY

[1] Miltiadis Allamanis, Sheena Panthaplackel, and Pengcheng Yin. "Unsupervised Evaluation of Code LLMs with Round-Trip Correctness". In: *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024. URL: https://openreview.net/forum?id=YnFuUX08CE.

[2] Nadia Alshahwan et al. "Automated Unit Test Improvement using Large Language Models at Meta". In: *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*. Ed. by Marcelo d'Amorim. ACM, 2024, pp. 185–196. DOI: 10.1145/3663529.3663839.

[3] Jeremy Avigad. "Opinion: The Mechanization of Mathematics". In: *Notices of the American Mathematical Society* 65.6 (June 2018), pp. 681–690. DOI: 10.1090/noti1688.

[4] Michael Barnett et al. "Boogie: A Modular Reusable Verifier for Object-Oriented Programs". In: *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*. Ed. by Frank S. de Boer et al. Vol. 4111. Lecture Notes in Computer Science. Springer, 2005, pp. 364–387. DOI: 10.1007/11804192_17.

[5] Yoshua Bengio et al. "A Neural Probabilistic Language Model". In: *J. Mach. Learn. Res.* 3 (2003), pp. 1137–1155. URL: https://jmlr.org/papers/v3/bengio03a.html.

[6] Víctor A. Braberman et al. *Tasks People Prompt: A Taxonomy of LLM Downstream Tasks in Software Verification and Falsification Approaches*. 2024. arXiv: 2404.09384 [cs.SE].

[7] David Brandfonbrener et al. *VerMCTS: Synthesizing Multi-Step Programs using a Verifier, a Large Language Model, and Tree Search*. 2024. arXiv: 2402.08147 [cs.SE].

[8] Aakanksha Chowdhery et al. "PaLM: Scaling Language Modeling with Pathways". In: *J. Mach. Learn. Res.* 24 (2023), 240:1–240:113. URL: http://jmlr.org/papers/v24/22-1144.html.

[9] Gelei Deng et al. "PentestGPT: Evaluating and Harnessing Large Language Models for Automated Penetration Testing". In: *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. Ed. by Davide Balzarotti and Wenyuan Xu. USENIX Association, 2024. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/deng.

[10] João Pascoal Faria and Rui Abreu. "Case Studies of Development of Verified Programs with Dafny for Accessibility Assessment". In: *Fundamentals of Software Engineering - 10th International Conference, FSEN 2023, Tehran, Iran, May 4-5, 2023, Revised Selected Papers*. Ed. by Hossein Hojjat and Erika Ábrahám. Vol. 14155. Lecture Notes in Computer Science. Springer, 2023, pp. 25–39. DOI: 10.1007/978-3-031-42441-0_3.

[11] Sebastian Farquhar et al. "Detecting hallucinations in large language models using semantic entropy". In: *Nature* 630.8017 (2024), pp. 625–630. DOI: 10.1038/S41586-024-07421-0.

[12] Michael Fu et al. "ChatGPT for Vulnerability Detection, Classification, and Repair: How Far Are We?" In: *30th Asia-Pacific Software Engineering Conference, APSEC 2023, Seoul, Republic of Korea, December 4-7, 2023*. IEEE, 2023, pp. 632–636. DOI: 10.1109/APSEC60848.2023.00085.

[13] Google AI. *Gemini API*. https://ai.google.dev/aistudio. Accessed: 2024-09-29. 2024.

[14] Chris Hawblitzel et al. "IronFleet: Proving Practical Distributed Systems Correct". In: *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*. Ed. by Ethan L. Miller and Steven Hand. ACM, 2015, pp. 1–17. DOI: 10.1145/2815400.2815428.

[15] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation*. 3rd. Pearson international edition. Addison-Wesley, 2007. ISBN: 978-0-321-47617-3.

[16] Jie Hu, Qian Zhang, and Heng Yin. *Augmenting Greybox Fuzzing with Generative AI*. 2023. arXiv: 2306.06782 [cs.CR].

[17] Lei Huang et al. *A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions*. 2023. arXiv: 2311.05232 [cs.CL].

[18] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models*. 3rd. Online manuscript released August 20, 2024. 2024.

[19] Saurav Kadavath et al. *Language Models (Mostly) Know What They Know*. 2022. arXiv: 2207.05221 [cs.CL].

[20] K. Rustan M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*. Ed. by Edmund M. Clarke and Andrei Voronkov. Vol. 6355. Lecture Notes in Computer Science. Springer, 2010, pp. 348–370. DOI: 10.1007/978-3-642-17511-4_20.

[21] Tomás Mikolov et al. "Efficient Estimation of Word Representations in Vector Space". In: *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2013. DOI: 10.48550/arXiv.1301.3781.

[22] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[23] Eric Mugnier et al. *Laurel: Generating Dafny Assertions Using Large Language Models*. 2024. arXiv: 2405.16792 [cs.LO].

[24] OpenAI et al. *GPT-4 Technical Report*. 2024. arXiv: 2303.08774 [cs.CL].

[25] Kishore Papineni et al. "Bleu: a Method for Automatic Evaluation of Machine Translation". In: *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA*. ACL, 2002, pp. 311–318. DOI: 10.3115/1073083.1073135. URL: https://aclanthology.org/P02-1040/.

[26] Chuyue Sun et al. "Clover: Closed-Loop Verifiable Code Generation". In: *AI Verification - First International Symposium, SAIV 2024, Montreal, QC, Canada, July 22-23, 2024, Proceedings*. Ed. by Guy Avni et al. Vol. 14846. Lecture Notes in Computer Science. Springer, 2024, pp. 134–155. DOI: 10.1007/978-3-031-65112-0_7.

[27] Christian Szegedy. "A Promising Path Towards Autoformalization and General Artificial Intelligence". In: *Intelligent Computer Mathematics - 13th International Conference, CICM 2020, Bertinoro, Italy, July 26-31, 2020, Proceedings*. Ed. by Christoph Benzmüller and Bruce R. Miller. Vol. 12236. Lecture Notes in Computer Science. Springer, 2020, pp. 3–20. DOI: 10.1007/978-3-030-53518-6_1.

[28] Gemini Team et al. *Gemini: A Family of Highly Capable Multimodal Models*. 2024. arXiv: 2312.11805 [cs.CL].

[29] Hugo Touvron et al. *LLaMA: Open and Efficient Foundation Language Models*. 2023. arXiv: 2302.13971 [cs.CL].

[30] Aaron Tomb Tristan and Jean-Baptiste. *Avoiding verification brittleness in Dafny*. Dafny Blog, https://dafny.org/blog/2023/12/01/avoiding-verification-brittleness/. Accessed: 2024-12-07. Dec. 2023.

[31] Ashish Vaswani et al. "Attention is All you Need". In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. Ed. by Isabelle Guyon et al. 2017, pp. 5998–6008. URL: https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

[32] Jason Wei et al. "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models". In: *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*. Ed. by Sanmi Koyejo et al. 2022. URL: http://papers.nips.cc/paper%5C_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html.

[33] Jason Wei et al. "Emergent Abilities of Large Language Models". In: *Trans. Mach. Learn. Res.* (2022), 209:1–209:30. URL: https://openreview.net/forum?id=yzkSU5zdwD.

[34] Haoze Wu, Clark Barrett, and Nina Narodytska. "Lemur: Integrating Large Language Models in Automated Program Verification". In: *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11 2024*. OpenReview.net, 2024. URL: https://openreview.net/forum?id=Q3YaCghZNt.

[35] Yuhuai Wu et al. "Autoformalization with Large Language Models". In: *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*. Ed. by Sanmi Koyejo et al. 2022. URL: http://papers.nips.cc/paper%5C_files/paper/2022/hash/d0c6bc641a56bebee9d985b937307367-Abstract-Conference.html.

[36] Ziyang Xu et al. "Take Care of Your Prompt Bias! Investigating and Mitigating Prompt Bias in Factual Knowledge Extraction". In: *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation, LREC/COLING 2024, 20-25 May, 2024, Torino, Italy*. Ed. by Nicoletta Calzolari et al. ELRA and ICCL, 2024, pp. 15552–15565. URL: https://aclanthology.org/2024.lrec-main.1352.

[37] Dylan Zhang et al. "LM-PACE: Confidence Estimation by Large Language Models for Effective Root Causing of Cloud Incidents". In: *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*. Ed. by Marcelo d'Amorim. ACM, 2024, pp. 388–398. DOI: 10.1145/3663529.3663858.

[38] Yue Zhang et al. *Siren's Song in the AI Ocean: A Survey on Hallucination in Large Language Models*. 2023. arXiv: 2309.01219 [cs.CL].