Seeking Specifications: The Case for Neuro-Symbolic Specification Synthesis

Citation for the original published paper (version of record):
Granberry, G., Ahrendt, W., Johansson, M. (2026). Seeking Specifications: The Case for Neuro-Symbolic Specification Synthesis. Formal Aspects of Computing, 38(2).
http://dx.doi.org/10.1145/3785411

(article starts on next page)

Seeking Specifications: The Case for Neuro-Symbolic Specification Synthesis

GEORGE GRANBERRY, Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden
WOLFGANG AHRENDT, Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden
MOA JOHANSSON, Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden

This work is concerned with the generation of formal specifications from code, using Large Language Models (LLMs) in combination with symbolic methods. Concretely, in our study, the programming language is C, the specification language is ACSL, and the LLM is Deepseek-R1. In this context, we address two research directions, namely the specification of intent vs. implementation on the one hand, and the combination of symbolic analyses with LLMs on the other hand. For the first, we investigate how the absence or presence of bugs in the code impacts the generated specifications, as well as whether and how a user can direct the LLM to specify intent or implementation, respectively. For the second, we investigate the impact of results from symbolic analyses on the specifications generated by the LLM. The LLM prompts are augmented with outputs from two formal methods tools in the Frama-C ecosystem, Pathcrawler and EVA. We demonstrate how the addition of symbolic analysis to the workflow impacts the quality of annotations.

CCS Concepts: • **Computing methodologies** → **Artificial intelligence**; • **Software and its engineering** → *Software notations and tools*; • **Theory of computation** → **Program reasoning;**

Additional Key Words and Phrases: Formal methods, LLMs, symbolic execution, static analysis, reasoning

ACM Reference Format:
George Granberry, Wolfgang Ahrendt, and Moa Johansson. 2026. Seeking Specifications: The Case for Neuro-Symbolic Specification Synthesis. *Form. Asp. Comput.* 38, 2, Article 16 (June 2026), 36 pages. https://doi.org/10.1145/3785411

1 Introduction

The field of specification synthesis offers a possible solution to the inherent complexities involved in creating and maintaining specifications for software verification. Creating useful specifications demands a deep understanding of both the specification language and the verification process,

Associate Editor: Nikolai Kosmatov and Laura Kovács

This work was supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP), funded by the Knut and Alice Wallenberg Foundation.
Authors' Contact Information: George Granberry (corresponding author), Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden; e-mail: georgegr@chalmers.se; Wolfgang Ahrendt, Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden; e-mail: ahrendt@chalmers.se; Moa Johansson, Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden; e-mail: moa.johansson@chalmers.se.

ACM 1433-299X/2026/06-ART16
https://doi.org/10.1145/3785411

which can often be as intricate, if not more so, than the software to be verified. This complexity poses a significant barrier [14, 60], especially in dynamic environments where frequent updates and refactoring are the norm. Maintaining an accurate alignment between ever-evolving code and its specifications can become a cumbersome and error-prone process.

Specification synthesis potentially alleviates these concerns by automating the generation and adaptation of specifications. Instead of requiring developers to manually write detailed specifications – a task that can be both time-consuming and susceptible to human error – specification synthesis aims at inferring and editing specifications directly from the codebase and associated context. The goal is to transform specifications into convenient guardrails that provide valuable insights and guidance to programmers, rather than chores performed at the end of the software pipeline.

Approaches to generating specifications typically employ a range of symbolic techniques, encompassing static as well as dynamic analyses [35]. For instance, Daikon [15], a widely recognised tool in dynamic analysis, infers properties by observing program behaviour at runtime. On the other hand, static analysers deduce properties based on the program's structure without executing it. Despite their precision, the primary limitation of these methods is their rigidity. Symbolic techniques are constrained by a limited range of expressible properties and typically specialise in specific types of analyses which restricts their flexibility in adapting to diverse verification needs.

On the other side of specification synthesis techniques are machine-learning-based **Natural Language Processing** (**NLP**) [6] and **Large Language Models** (**LLMs**) [8]. NLP tools specialise in understanding human language, such as comments, while LLMs stand out for their flexibility and creativity when dealing with arbitrary inputs. These models can theoretically generate any specification that can be articulated in their associated language, provided that they are appropriately trained and given the right prompts.

However, this strength also introduces a significant challenge: the large range of potential specifications LLMs can produce often includes outputs that may not be practically useful or even plainly wrong. While an LLM can generate a wide array of specifications, the lack of inherent direction means that there is no guarantee that the generated specifications will be relevant or valuable for specific verification tasks. This challenge has led users of LLM-based synthesis to utilize *prompt engineering* [65] techniques in order to increase the likelihood of the LLM to produce outputs that align with their objectives. Despite the seeming effectiveness of this practice across general LLM usage, little research has been done to figure how these prompting techniques are utilized by LLMs when it comes to specification synthesis.

A fundamental conceptual issue in the context of specification synthesis is that specifications synthesized directly from code may reflect the implemented behavior rather than the intended behavior. Depending on the purpose of the specification, this can be problematic in contexts where correctness cannot simply be assumed, and synthesized specifications risk formalizing unintended behavior, potentially obscuring bugs instead of revealing them.

In this article, we demonstrate that specification synthesis is a viable and effective practice using current-day tools. First, we explore how LLMs handle the conceptual challenge of distinguishing between implemented and intended behavior when synthesizing specifications. Afterward, we propose a complementary integration of existing symbolic analysis tools with LLM-based specification synthesis, combining the precision of symbolic methods with the flexibility and expressive power of LLMs.

In our experiments, we observed that LLMs - particularly those with advanced reasoning capabilities - were resilient against the implementation-versus-intent problem. These models were almost always able to identify bugs and target **intended** behavior rather than the implemented behavior when provided with incorrect or buggy code. Furthermore, their reasoning capabilities enabled effective bug detection **even with no context provided**. We also found that prompts can be adjusted to explicitly acknowledge the possibility of bugs, thereby further guiding the LLM toward producing specifications that reflect the developer's intent rather than the implemented behavior.

In addition, we find that incorporating results from symbolic analysis into prompts significantly impacts both the quantity and content of generated specifications. To evaluate this effect, we focus on tooling from the Frama-C ecosystem, which provides formal analysis tools for C programs. We express specifications using the ACSL specification language and leverage two tools: PathCrawler, a structural test case generator that produces input/output examples, and EVA, a static analyzer that performs value analysis. In general, augmenting the prompts with output symbolic tools led to fewer annotations overall, but increased the relevance and focus of the resulting specifications. Symbolic input influenced the type of annotations as well: PathCrawler encouraged the generation of more structured and abstract postconditions, while EVA led to an increase in preconditions, directly reflecting its emphasis on identifying runtime errors. Together, these results highlight the adaptability of LLMs to integrate structured, symbolic information into their specification synthesis process.

Research Questions

This article investigates the following research questions:

(1) How does LLM-based specification synthesis from code handle buggy code?
(2) How can existing symbolic methods be used to influence and improve LLM-based specification synthesis methods?

This article extends and modifies the conference paper [19] in several ways. First, we replace OpenAI's GPT-4 with Deepseek-R1 and re-run all experiments using this newer model (more can be read about this in the Appendix). Second, we significantly expand the section on implementation versus intent by introducing a larger and more diverse dataset designed to test that distinction. Finally, we incorporate Deepseek-R1's more detailed reasoning output into our qualitative analysis, allowing for a more comprehensive understanding of how the model arrives at its specifications.

The structure of the article is as follows: First, we introduce the experimental setup, covering the tools, datasets, and prompt strategies used. Second, we present an investigation into the implementation-versus-intent distinction, including a dataset constructed to test how LLMs handle buggy or misleading implementations. Third, we perform neuro-symbolic prompting experiments, qualitatively analyzing the effect of adding the output of symbolic analyses to the prompts when generating specifications from code with Deepseek-R1. Finally, we perform some quantitative evaluation of the generated specifications using symbolic tools.

2 Tools and Languages

2.1 Frama-C and ACSL

The Frama-C ecosystem is an open-source suite of tools designed for the analysis of the source code of software written in C [31, 32]. It integrates various static and dynamic analysis techniques to evaluate the correctness, safety, and security of C programs. It also supports the specification language ACSL [3, 55], which is used to formulate *contracts* consisting of, among others, preconditions – assumptions on the input and prestate of a function – and postconditions – requirements on the output and poststate of a function. These contracts, examples of which can be seen both in Sections 3 and 4, provide a clear and formal framework for understanding and verifying a function's behaviour. Other ACSL annotations commonly used are *assertions* - stating a condition that needs

```
input_l,input_table[0],input_table[1],output,verdict
2,0,0,,success
2,0,73,73,0,success
```

Fig. 1. Example PathCrawler CSV output which describes inputs, outputs, and user provided oracle verdicts for BubbleSort.

to be true at some point in execution - and *loop invariants* which specify conditions that need to be maintained by every iteration of a loop.

2.2 Automated Test Generation: Pathcrawler

The PathCrawler tool is designed for the automated testing of C programs [66]. Its primary function is to generate and execute test inputs for C code, with a particular focus on achieving high code coverage. Employing a technique known as concolic testing [52] – using a combination of concrete and symbolic execution – Pathcrawler efficiently explores different execution paths in the program. First, it generates test inputs and then executes them, providing valuable information from the execution results across a broad spectrum of program paths. In addition, PathCrawler allows users to incorporate test oracles, classifying the outcome of every test case of some function. However, we want to highlight that we did not make such oracle implementations available to the LLM when asking it to generate specifications. Generic oracles can be seen as executable specifications, and would have diluted the significance of our experiments. Instead, we only include input/output pairs, with the non-generic verdict. An example of output from PathCrawler can be seen in Figure 1.

2.3 Value Analysis: EVA

The EVA static analyzer uses abstract interpretation to approximate a set of possible values that program variables can take to avoid certain runtime errors [7]. By doing so, it can identify a range of potential issues, such as division by zero, buffer overflows, null pointer dereferences, and arithmetic overflows. EVA's analysis helps in ensuring that the code behaves correctly across all possible execution paths and input values. EVA is designed to respect, and work with, ACSL annotations when they are present. For an example of an report ouput from EVA refer to Figure 2.

2.4 LLM and Prompts

We have chosen to use Deepseek-R1[1] (deepseek-reasoner version 2025-01-20) as our LLM for generating specifications. We ran this experiment using GPT-4 in our previous work [19] but chose to switch to Deepseek-R1, not only for its competitive pricing, but also for its more extensive reasoning output. For a more comprehensive write-up about the differences between Deepseek-R1 and GPT-4 in our experiments refer to Appendix A.

We prompt Deepseek-R1 with a C program, providing instructions for how to generate ACSL annotations in a step-by-step manner. We also include a few examples of valid ACSL annotations in the prompts, leveraging a form of "few-shot learning" [8] to guide the model. Unlike the previous version of this article, we did not include "Chain-of-thought" [62] prompting techniques in the prompt as reasoning models such a Deepseek-R1 perform this by default. Our "baseline" prompt, which is used to generate specifications without the aid of symbolic analysis, can be seen in Figure 3 on Page 6.

Expanding upon the baseline prompt, in Section 4, we augment it with outputs from the Pathcrawler and EVA tools to further guide the model. These two prompts can be found in Figures 4 and 5 on pages 7 and page 8, respectively.

[1]https://platform.deepseek.com

```
...
[eva:alarm] temp_files/tmphpdyn83w/eva_temp.c:8: Warning:
  signed overflow. assert -2147483648 ≤ x * 2;
[eva:alarm] temp_files/tmphpdyn83w/eva_temp.c:8: Warning:
  signed overflow. assert x * 2 ≤ 2147483647;
[eva:alarm] temp_files/tmphpdyn83w/eva_temp.c:9: Warning:
  signed overflow. assert -2147483648 ≤ v - y;
[eva:alarm] temp_files/tmphpdyn83w/eva_temp.c:9: Warning:
  signed overflow. assert v - y ≤ 2147483647;
[eva:alarm] temp_files/tmphpdyn83w/eva_temp.c:11: Warning:
  out of bounds write. assert \valid(tab + 2);
[kernel] temp_files/tmphpdyn83w/eva_temp.c:11: Warning:
  all target addresses were invalid. This path is assumed to be dead.
[eva] done for function testme
[eva] temp_files/tmphpdyn83w/eva_temp.c:11:
  assertion 'Eva,mem_access' got final status invalid.
[eva] ====== VALUES COMPUTED ======
[eva:final-states] Values at end of function testme:
  NON TERMINATING FUNCTION
[eva:summary] ====== ANALYSIS SUMMARY ======
  ---------------------------------------------------------------------------
  1 function analyzed (out of 1): 100% coverage.
  In this function, 4 statements reached (out of 14): 28% coverage.
  ---------------------------------------------------------------------------
  Some errors and warnings have been raised during the analysis:
    by the EVA analyzer:       0 errors    0 warnings
    by the Frama-C kernel:     0 errors    1 warning
  ---------------------------------------------------------------------------
  5 alarms generated by the analysis:
       4 integer overflows
       1 invalid memory access
  1 of them is a sure alarm (invalid status).
```

Fig. 2. Example EVA report.

3 Implementation vs. Intent

A core criticism of specification synthesis is that it risks producing incorrect specifications when the source code contains bugs. One might argue that this issue is not critical since users should be able to identify faulty specifications just as they would identify faulty code and revise accordingly. However, modern specification languages such as ACSL are expressive programming languages in their own right. As such, users may be just as prone to overlooking subtle errors in specifications as they are to overlooking bugs in code. This creates a potential for overconfidence in specifications that merely formalize flawed behavior, rather than revealing it. For this reason, the distinction between implementation and intent becomes a central concern. If LLM-based synthesis tools cannot recover the intended intent of a program from buggy code, then the synthesized specification may reinforce errors rather than assist in their detection.

LLMs have opened up new pathways for addressing the challenges of specification synthesis. In this section, we perform a qualitative analysis of the reasoning processes generated by Deepseek-R1 to explore how these capabilities manifest in practice. By examining how the model responds to mismatches between implementation and intent, we aim at understanding whether and how LLMs can identify discrepancies, infer programmer intent, and produce specifications that go beyond simply restating buggy behavior.

Specifically, we aim at answering the following questions:

(1) How capable are LLMs at identifying the intent of a program?
(2) How reliant are they on cues such as function names and comments?
(3) Is an LLM able to recognize subtle bugs in programs?

```
You are a LLM that takes the following inputs and returns a C program annotated with ACSL
    annotations.

Inputs:
1. A C program with no ACSL annotations

GOALS:
1. Describe any abstract properties that could be represented as ACSL annotations
2. Generate ACSL annotations that describe the functional behavior of the program based on your
    analysis of the program
3. Returning a program with no annotation is not a valid solution
4. Do not edit the C code, only add annotations
5. Do not add any annotations inside of the function body
6. Do not generate loop invariants
7. Do not skip any code in the returned solution to make it shorter.

ANNOTATION EXAMPLES:

Examples 1 (single annotation):
/*@ requires low >= 0 && high <= 9; */

Example 2 (block annotation style):
//Only use this style for function headers. Do not use blocks for multiple annoations in the
    function body
/*@
  @ requires low >= 0 && high <= 9;
  @ requires elem >= 0 && elem <= 9;
*/

FORMAT INSTRUCTIONS:

First describe your reasoning behind the added annotations

Return the annotated c code wrapped in markdown
```c
...
```

START OF INPUT:

```c
{program_str}
```
```

Fig. 3. Prompt used for generating ACSL annotations.

(4) If a bug is recognized, will the LLM generate a specification that mirrors the buggy implementation or one that follows the intent of the program?

3.1 Dataset

The dataset used in this study is publicly available at https://github.com/ggranberry/intent_dataset. We collected programs for a test suite called *intent_tests*, composed of 50 C programs, designed to investigate how LLMs manage conflicts between implementation and intention. This suite is structured into four sections:

The first section, *Basic*, contains 20 programs commonly taught in undergraduate computer science curricula, including binary search, duplicate identification, queue insertion, linked-list append, string reversal, and palindrome checking. These serve as a benchmark on standard, widely recognized algorithms.

```
You are a LLM that takes the following inputs and returns a C program annotated with ACSL
     annotations.

Inputs:
1. A C program with no ACSL annotations
2. A CSV file that represents input/output pairs that result from running the Frama-C Pathcrawler
     tool

GOALS:
1. Descrbe any abstract or functional properties that you can reason about from the pathcrawler
     output
2. Describe any abstract or functional properties based on the code
3. Generate ACSL annotations that describe the functional behavior of the program based on both
     your analysis of of the program as well as the pathcrawler output
4. Returning a program with no annotation is not a valid solution
5. Do not edit the C code, only add annotations
6. Do not add any annotations inside of the function body
7. Do not generate loop invariants
8. Do not skip any code in the returned solution to make it shorter.

ANNOTATION EXAMPLES:

Examples 1 (single annotation):
/*@ requires low >= 0 && high <= 9; */

Example 2 (block annotation style):
//Only use this style for function headers. Do not use blocks for multiple annoations in the
     function body
/*@
  @ requires low >= 0 && high <= 9;
  @ requires elem >= 0 && elem <= 9;
*/

FORMAT INSTRUCTIONS:
First describe your reasoning behind the added annotations
Return the annotated c code wrapped in markdown
```c
...
```
START OF INPUT:
```c
{program_str}
```
```csv
{pathcrawler_str}
```
```

Fig. 4. Prompt used for generating ACSL annotations with PathCrawler as additional context.

The second section, named *Famous*, includes 10 more complex but still well-known programs, such as Duff's Device and Kahan's Summation Algorithm. These programs present a more challenging scenario for an LLM in which it is familiar with the provided program but might have trouble generating an appropriate specification.

In the third section, *Mirror*, we include 10 intentionally brief programs, mostly consisting of just one or two lines of code. These are specifically designed so their implementations can be straightforwardly and precisely mirrored into ACSL specifications. The purpose of this section is to test how tightly coupled the specifications are to the implementation.

The fourth section, *Unique*, comprises 10 programs that we authored ourselves. They are designed to be straightforward, but the handwritten nature makes them less likely to be included in an LLM's

```
You are a LLM that takes the following inputs and returns a C program annotated with ACSL
    annotations.

Inputs:
1. A C program with no ACSL annotations
2. A report generated by the Frama-C EVA static analysis tool

GOALS:
1. Descrbe any abstract or functional properties that you can reason about from the EVA report
2. Describe any abstract or functional properties based on the code
3. Generate ACSL annotations that describe the functional behavior of the program based on both
    your analysis of of the program as well as the EVA report
4. Returning a program with no annotation is not a valid solution
5. Do not edit the C code, only add annotations
6. Do not add any annotations inside of the function body
7. Do not generate loop invariants
8. Do not skip any code in the returned solution to make it shorter.

ANNOTATION EXAMPLES:

Examples 1 (single annotation):
/*@ requires low >= 0 && high <= 9; */

Example 2 (block annotation style):
//Only use this style for function headers. Do not use blocks for multiple annoations in the
    function body
/*@
  @ requires low >= 0 && high <= 9;
  @ requires elem >= 0 && elem <= 9;
*/

FORMAT INSTRUCTIONS:

First describe your reasoning behind the added annotations

Return the annotated c code wrapped in markdown
```c
...
```

START OF INPUT:
```c
{program_str}
```

```
{eva_str}
```
```

Fig. 5. Prompt used for generating ACSL annotations with EVA as additional context.

training data. With these tests we hoped to see if Deepseek R1's reasoning abilities extended to programs that it has never encountered.

3.2 Generating Annotations

The variability of LLMs like Deepseek-R1 can be adjusted via its "temperature" setting which controls the level of determinism during generation. As we are interested in exploring what the *average* specification generated by a given prompt is, we choose to generate three distinct specifications for each program (and prompt) within our test suite, repeating the steps above with a temperature setting of 0.7. This approach allows us to capture a spectrum of possible specifications, while not being too economically costly.

Given that we are using the `deepseek − reasoner` model, each generated response consists of two parts: the ACSL specification and an accompanying chain-of-thought reasoning provided by the model prior to generating the specification. Having direct access to the LLM's reasoning process is particularly beneficial, as it provides insight into the internal logic employed by the model. Specifically, it allows us to pinpoint nuance such as determining whether the LLM recognizes familiar algorithms, discerns the intended functionality of the provided programs, or identifies the subtle bugs we deliberately introduced.

Loop invariants are key for functioning specifications. However, generating loop invariants is not the focus of this article. The generation of loop invariants with LLMs calls for sophisticated prompts dedicated to this task [18, 30]. Rather, we focus on function contracts – particularly preconditions, postconditions, and assigns clauses – and our prompts are designed to generate these while leaving out loop invariants.

3.3 Evaluation

Evaluating specifications is challenging due to the absence of a definitive specification for any given program. Different users often have varying priorities and perspectives on which properties are worth verifying, making the notion of a definitive specification subjective. Similarly, a specification might be logically correct, but more or less trivial with respect to the program at hand, in which case it provides little value.

In light of these challenges, our evaluation methodology does not attempt to benchmark the generated specifications against any predefined gold standard, nor does it aim at determining the optimal approach to creating specifications. Instead, our focus is on identifying the behaviours and patterns that emerge when generating specifications with controlled prompting and program edits.

While the generated specifications themselves are interesting, we found that the reasoning process that lead to the specification to be much more informative. To this point, we primarily focused on analysis of the **reasoning output** generated from the different variations of the *Intent Tests* suite. In these tests, we treat the programs themselves as controlled variables, systematically introducing buggy variations to assess how capable the LLM is at pinpointing and specifying user intent in code.

We generated specifications for four variations of the *Intent Tests* suite:

(1) **Baseline Set**: The original, unmodified set of programs, which we consider correct to the best of our knowledge. This serves as our baseline for comparing the effects of our subsequent modifications.
(2) **Buggy Set**: We introduced subtle bugs into each of the programs. These errors were intentionally designed to be syntactically minimal at the same time as they significantly impact the semantic correctness of the programs.
(3) **Anonymized Set**: We anonymized the function names in the baseline set, replacing meaningful identifiers with generic placeholders to obscure the original intent from the LLM.
(4) **Buggy Anonymized Set**: This variation combined both anonymization of function names and the introduction of the same subtle bugs used in the Buggy Set.

3.4 Identifying Intent

Deepseek-R1 was overwhelmingly successful at identifying the intent behind the programs provided. Across nearly 600 specification generations, the model correctly inferred the intended functionality for nearly every program, including those with injected bugs and anonymized function names.

The 3 exceptions in which the intent was not able to be identified occurred in programs that were both buggy and anonymized. In these few cases, the injected bugs altered the program semantics significantly enough that the program more resembled another coherent program.

Table 1. Number of Bugs Noticed by the LLM Grouped by Test Suite Category

Test Suite	Bugs Noticed
Basic	47/60
Famous	9/30
Mirror	28/30
Unique	26/30
Anonymized Basic	50/60
Anonymized Famous	14/30
Anonymized Mirror	26/30
Anonymized Unique	27/40

3.5 Reliance on Identifiers

In the unmodified version of the *Intent Tests*, programs were presented with descriptive and meaningful function names. This setup naturally provides strong cues that can make the task of identifying program intent relatively trivial for an LLM. However, in our experiments, anonymizing these function names had surprisingly little effect on the model's performance. In fact, Deepseek-R1 seemed to be slightly better at identifying bugs with anonymized function names.

In the reasoning outputs, we consistently observed that the LLM was either able to immediately recognize the program or experienced an "Aha!" moment partway through the reasoning process. This indicates that the model's recognition capabilities are driven more by its understanding of the program itself rather than simple reliance on identifier names.

```
Wait, this looks familiar. Oh right! This is the fast inverse square root algorithm, but modified.
```

3.6 Identifying Bugs

Deepseek-R1 was also effective at identifying subtle bugs in code, even when not explicitly prompted to search for them. As shown in Table 1, the model was able to detect bugs in the majority of the programs provided.

However, the model was noticeably less effective in detecting bugs in the *Famous* category of the test suite. One possible interpretation for this is that the programs in this group are more complex than those in the other categories, making them more difficult to reason about. Another insight, revealed through examination of the reasoning traces, is that the model often struggled to identify bugs in these cases because it had difficulty constructing meaningful specifications.

For instance, in the case of Kahan's Summation Algorithm, which is designed to reduce precision loss in floating-point summation, the model immediately identified the purpose of the program. However, it then spent most of its reasoning phase attempting to determine which properties could be usefully specified for such a program. This effort left little time or attention for analyzing the structure or behavior of the code itself, thereby reducing its ability to detect the bug. This pattern was repeated in other complex examples, where the effort to determine an appropriate specification overshadowed reasoning about program correctness.

3.7 Is Implementation or Intent Specified?

While Deepseek-R1 demonstrates the ability to detect bugs, an equally important point is how it reacts when a bug is found. In every reasoning trace we examined, the model included a statement indicating that its specification would be based on the provided implementation rather than any inferred intent. With example phrasing like

```
Wait, strlen returns the length, which is the number of characters excluding the null terminator.
    So for a string ``aba'', strlen is 3. So right starts at 3. But the valid indices are 0,1,2.
    So str[right] in the first iteration would be str[3], which is the null terminator. That's a
    problem. Oh, the original code is incorrect here. But the user's code is given, and we have
    to annotate it as is. So the function as written is wrong because it's comparing the null
    terminator. But the user's code may have a bug, but we have to add annotations based on the
    code as given.
```

Based on this reasoning, Deepseek-R1 went on to create specifications that didn't match the intent of the user even though it was able to identify the intent of the code. One such specification can be seen in Figure 6

Although this default behavior may not align with the expectations of users hoping to generate specifications for intended behavior, it is easily addressed. By modifying the prompt to acknowledge the possibility of buggy code and to explicitly instruct the model to generate specifications for the intended behavior rather than the actual implementation, we were able to shift the model's output accordingly. Specifications generated under these revised prompts more closely aligned with the intended functionality, even when the implementation contained subtle errors.

```
We are given a function isPalindrome that is intended to check if a string is a palindrome.
However, the implementation is flawed. The current code checks if every consecutive pair of
    characters are equal (which would mean all characters are the same) and returns 0 if any
    consecutive pair is not equal. This is not the correct way to check for a palindrome.

The intended behavior:
A palindrome is a string that reads the same forwards and backwards.
So, we should compare the first character with the last, the second with the second last, and so
    on.

Since we are not allowed to change the code, we must annotate based on the intended behavior.

Steps for annotation:

   The function takes a const char* str, meaning it does not modify the string.

   We must note that the string must be valid (i.e., null-terminated) so that strlen can be used.

   The intended algorithm:

       Set two indices, left starting at 0 and right at the last character (strlen(str)-1).

       Then, while left < right, compare str[left] and str[right]. If they are not equal, return
    0.

       If all comparisons are equal, return 1.

However, the code does not do that. The code in the loop is incorrect.

Since we are to annotate what the program is intended to do, we will write specifications
    accordingly.

We will add:

   A requires clause: The string must be valid (i.e., str is a valid pointer to a null-terminated
     string).

   An ensures clause: The result is 1 if and only if the string is a palindrome, and 0 otherwise
```

Figure 7 shows the specification generated for our buggy palindrome program once the prompt is instructed to prioritize intent.

```
/*@
  requires \exists integer i; i >= 0 && str[i] == '\0' && (\forall integer j; 0 <= j <= i ==>
      \valid_read(str + j));
  ensures \result == (\strlen(str) == 0);
  assigns \nothing;
*/
int isPalindrome(const char* str) {
    int left = 0, right = strlen(str);
    while (left < right) {
        if (str[left] != str[right]) {
            return 0; // Not a palindrome
        }
        left++;
        right--;
    }
    return 1; // Palindrome
}
```

Fig. 6. Specification generated for buggy palindrome.c when implementation is prioritized.

```
/*@
  requires \valid_read(str);
  ensures \result == 1 <==> (\forall integer i; 0 <= i < strlen(str) ==> str[i] == str[strlen
     (str)-1-i]);
  assigns \nothing;
  terminates;
*/
int isPalindrome(const char* str) {
    int left = 0, right = strlen(str) - 1;
    for (int i = 0; i < strlen(str); i++){
        if (str[i] != str[i+1]) {
            return 0; // Not a palindrome
        }
    }
    return 1; // Palindrome
}
```

Fig. 7. Specification generated for buggy palindome.c when intent is prioritized.

3.8 Summary: Implementation vs. Intent

Our qualitative analysis shows that Deepseek-R1, equipped with reasoning capabilities, is able to infer the intent behind programs even when presented with unfamiliar code, anonymized identifiers, and no documentation. It consistently demonstrated the ability to detect purposely subtle bugs-errors designed to evade casual inspection and reflected meaningfully on how to respond to such discrepancies. In every instance where a bug was identified, the model referred back to the prompt to determine whether to align its specification with the observed implementation or the inferred intent. While this LLM-based solution should be treated as the function **approximation** that is, LLMs such as Deepseek-R1 are a promising tool for specification synthesis, especially when guided by well-formed prompts and further model tuning.

4 Augmenting LLM-based Specification Synthesis with Symbolic Analysis

Two common criticisms of LLM-based specification synthesis are: (1) the space of possible specifications is effectively unbounded, possibly leading to specifications that are trivially true (e.g., `requirestrue`) or simply incorrect; and (2) LLMs lack the precision provided by domain-specialized symbolic tools.

However, there is no inherent reason why LLM-based generation and symbolic analysis could not be combined. In this work, we run symbolic tools on a given program and add their output to the LLM prompt. This method preserves the flexibility of the LLM while injecting the precision and focus of symbolic methods.

While many symbolic tools exist, we focus on two from the Frama-C verification framework: PathCrawler, which performs test generation through path exploration, and EVA, which performs static value analysis.

In this section, we aim at answering the following questions:

- How do LLMs such as Deepseek-R1 interact with the output of symbolic tools?
- Do the specifications generated by LLMs change when symbolic output is included in the prompt?
- What are some plausible directions for future integrations of symbolic methods and LLM-based synthesis?

4.1 Dataset

For our study, we have chosen to utilise the 55 programs from the Pathcrawler test suite, which we will refer to as **pathcrawler_tests**. Note that the test suite is not available online; it was provided to us by the Pathcrawler developers. Thereby, we can assume that this test suite was not directly used in the LLM's training (although it might have seen similar ones). This suite includes a variety of program types, balancing well-known algorithms like Binary Search with more niche programs such as a Soup Heater controller. It also contains small, specially crafted programs designed to test specific capabilities of Pathcrawler, adding another layer of diversity to our tests.

According to the Pathcrawler developers, the 55 programs are supposedly correct, in the sense that they are believed to correspond to their respective intention, and have no known bugs. Consequently, we examine with this suite to which extent our method produces accurate annotations for supposedly correct programs.

4.2 Generating Annotations for Pathcrawler Tests

In addition to all of the processes described in Section 3.2, we generate additional sets of annotations in for pathcrawler_tests. For each program, we generate three sets of ACSL annotations:

(1) **baselines_set**: Specifications generated using just the program in the prompt
(2) **pathcrawler_set**: Specifications generated by including a compact representation of test-cases generated by Patchcrawler in the prompt.
(3) **eva_set**: Specifications generated by running EVA on the program and including its report on potential value errors in the prompt.

4.3 Evaluation

4.3.1 Annotation Counts. First we quantify the annotations generated across all programs for our three sets: **baseline_set**, **pathcrawler_set**, and **eva_set**. While this approach does not say anything about the quality or semantics of any annotations, it does provide us with a macro-level view of which kinds of annotations are being generated and at what frequency. This systematic approach allows us to zoom out and capture the influence of different symbolic contexts on the annotation generation process.

4.3.2 Qualitative Analysis. Similar to our analysis in Section 3.3, this section focuses on a qualitative evaluation of the specifications generated by the LLM, as well as the reasoning that precedes them. Our goal is not to assign a quantitative score to specification quality, but rather to understand the model's behavior, especially in response to symbolic analyses added to the prompt.

In this phase, we place particular emphasis on comparing specifications produced in the `baseline_set` with those in the `pc_set` and `eva_set`, which include symbolic pre-analysis generated by tools such as PathCrawler and EVA. These comparisons allow us to evaluate how the presence of symbolic analysis affects the content of generated specifications.

Additionally, we analyze the model's reasoning steps prior to specification generation. This provides insight into how the symbolic analysis outputs are being interpreted and incorporated into the reasoning process that proceeds generation. This helps us to understand whether the analysis results are being used merely as contextual information or as direct cues.

4.3.3 Quantitative Analysis of EVA. Lastly, we perform a quantitative analysis of the annotations generated when EVA reports are included in the prompt. While formally proving the correctness of annotations is neither straightforward nor informative, due to the lack of loop invariants generated by our prompt, it is still possible to evaluate them using other measurable criteria. We also considered using the WP and PathCrawler plugins for quantitative analysis, but deductive verification with WP was out of scope without loop invariants, and the lack of integration between ACSL and PathCrawler prevented its use. In this article, we therefore focus on one metric: the number of runtime alarms produced by the EVA value analysis. This allows us to assess the impact of the generated annotations without providing loop invariants

4.4 Counting Annotations

In this section, we count the number of ACSL annotations produced in the specifications across the three datasets described in Section 4.2. Specifically, we count occurrences of three core annotation types: `requires` clauses (preconditions), `ensures` clauses (postconditions), and `assigns` clauses (which describe memory mutations). Each clause is counted per-keyword. For example, the annotations `@ensuresfoo`, `@ensuresbar&baz`, and `@requiresboo|car` would be counted as two `ensures` clauses and one `requires` clause. These counts are presented in Figure 8 on page 15.

While a small number of other annotation types such as `logic` predicates or `ghost` variables were occasionally generated, they were too infrequent be worth including in our analysis. Our goal is to determine whether the inclusion of symbolic tools like PathCrawler leads to any noticeable differences in the number or type of annotations produced by the model.

4.5 Annotation Counts

4.5.1 More is Not Better. An important point to consider when interpreting the annotation counts presented in Figure 8 is that a larger number of annotations does not necessarily indicate a better or more useful specification. For instance, a trivial postcondition such as `ensurestrue;` is always redundant.

We encountered the issue of frivolous ACSL clauses in the previous iteration of this experiment, where GPT-4 often generated `assigns` clauses within the function body for every variable assignment. While these annotations adhered to ACSL syntax, they were semantically vacuous and did not aid in program verification or understanding.

When re-running our experiments with Deepseek, we observed that specifications generated without symbolic assistance tended to contain the highest number of annotations overall. However, unlike in our earlier studies, Deepseek did not produce clearly frivolous specifications. Instead, we observed a modest increase of approximately 20% in annotation counts.

Although the reasoning traces did not provide a definitive explanation for this behavior, one plausible interpretation is that, in the absence of additional guidance from symbolic tools, the model focused more on easily identifiable elements of the code. This may have led to an overproduction of annotations reflecting basic implementation details.

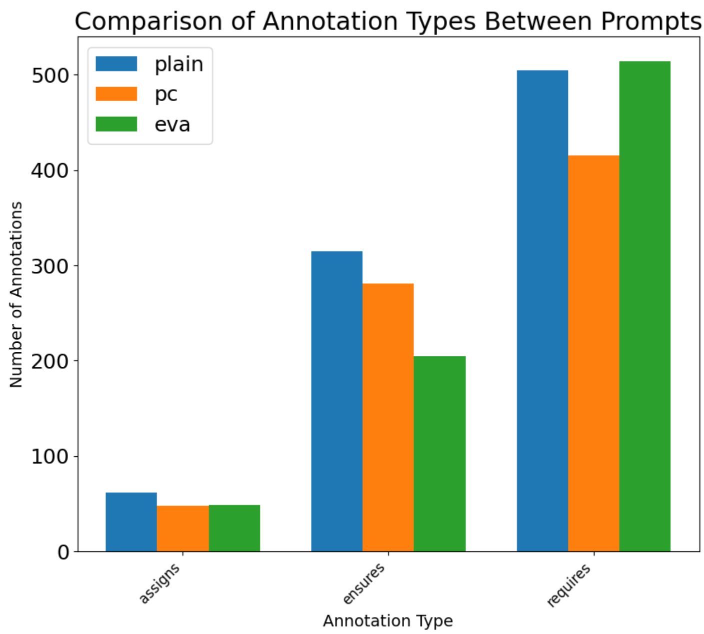

Fig. 8. Annotation-type counts for each prompt.

4.5.2 EVA and Preconditions. The only annotation type for which we observed a clear increase was the number of preconditions generated when EVA reports were included in the prompt. This finding aligns with results previous iterations of this experiments where GPT-4 converted EVA alarms into preconditions to remedy possible runtime errors. We explore this behavior in more detail in the dedicated section on EVA that follows.

4.6 PathCrawler Qualitative Analysis

4.6.1 Usage of Examples. When asked to generate specifications, Deepseek-R1 consistently reasons about code using examples whether or not they are explicitly provided in the prompt. As shown in Figure 9, in the absence of examples the model invents its own to support its understanding of program behavior. However, when concrete examples are included in the prompt, as is the case when using outputs from PathCrawler, the provided examples are directly used during the reasoning process. This can be seen in the reasoning process shown in Figure 10.

This behavior is desirable. Unlike ad-hoc examples generated by the LLM, the input/output pairs produced by PathCrawler are selected based on structural analysis of the program, and the criteria for their generation can be configured and understood by the user interacting with PathCrawler. While the broader problem of structural test case generation remains an open research area, it is also mature enough to offer practical tools and workflows. Integrating such examples into the LLM's input gives users a controllable way to guide the specification generation process with known and analyzable test scenarios.

4.6.2 Non-trivial Example Reasoning. The reasoning traces generated by Deepseek-R1 reveal that its engagement with input examples is non-trivial. Rather than merely accepting or describing

```
The CSV input has some test cases. For example, input_t[0] and input_t[1] are -37 and 0, and
     valeur 0. The output is 0. Wait, in the function 'testme', the output is 'retour', which
     depends on the return value of 'g'. Let's trace that.

In the first test case: n=2, valeur=0, t is [-37,0]. So 'g' checks if 0 < t[0] (-37)? No. Then
     check if 0 >= t[1] (0), yes. So r is set to -1. Then 'testme' gets i=-1, so retour is 0.
     Which matches the output 0.

Another test case: input_n=2, input_valeur=-91, t[0]=0, t[1]=62. So in 'g', valeur is -91. -91 < t
     [0] (0), so r is -2. Then 'testme' sets retour to 0. Output is 0. Correct.

Third test case: n=2, valeur=0, t is [0,12]. In 'g', 0 is not less than t[0], which is 0. Then
     check if 0 >= t[1] (12), no. Then loop over i=0 (since n=2, loop runs once). Check if 0 >= t
     [0] (0) and < t[1] (12). Yes. So r=0. So 'testme' returns 1. Which matches the output 1.

So from these test cases, the 'g' function seems to be looking for the index where 'valeur' is in
     the interval [t[i], t[i+1}), but only if the array is sorted in ascending order. Wait, in the
      third test case, t is [0,12], sorted. But what if the array isn't sorted? Let's say t is
     [5,3]. If valeur is 4, then t[0] is 5, which is larger than 4, so r stays -2. Then check if 4
      >= t[1] (3), yes, so r is -1. The loop would check i=0, 4 >=5? No. So r remains -1. So 'g'
     returns -1, and 'testme' returns 0. So 'g' might expect that the array is sorted. But the
     code itself doesn't enforce that. So maybe the function 'g' is intended to work on a sorted
     array. So maybe the preconditions require that the array is sorted in ascending order.
```

Fig. 9. Snippet of reasoning done for ADPCM.c (Adaptive differential pulse-code modulation) when provided with PathCrawler output.

```
But perhaps we can model the behavior more precisely. Let's think about the possible cases for '
     testme':

- If g returns -1 or -2, retour is 0.
- If g returns between 0 and n-1, retour is 1.
- Else, retour is -1. But according to 'g's code, this case can't happen. So maybe there's an
     error here. But the code as written includes the else, so in ACSL, we need to cover all
     possibilities.
```

Fig. 10. Snippet of reasoning done for ADPCM.c (Adaptive differential pulse-code modulation) when no input/output examples are provided.

examples, the model actively traces the inputs through the program logic and compares the outputs with its internal model of how the program works. This kind of step-by-step evaluation is distinctive to reasoning-style LLMs and was not observed in our previous experiments with GPT-4's chain-of-thought outputs.

Take, for example, Figure 11 and its accompanying reasoning in Figure 12. Deepseek does not stop at tracing examples; it also generalizes from them. In this case, the model traces multiple concrete input values through the bitwise logic of the program, including large negative integers. It performs bitwise shifts and masks, demonstrating an ability to simulate low-level operations "non-standard" numbers to verify behavior.

More notably, the model then abstracts these examples into general rules that relate specific input ranges to program outputs. This behavior is mirrored in both its reasoning trace and in the unprompted comments that it embedded in the specification. These annotations describe how different intervals of the input variable a influence the return value, reflecting a higher-level understanding of the program's semantics beyond what is visible in the code alone.

Comparing this to the specification generated without PathCrawler, shown in Figure 13, the specification generated with PathCrawler is both more desirable and semantically equivalent. While the PathCrawler-assisted version decomposes the specification into clear logical cases based

```
/*@
  // Case when input is -1 gives exactly 5
  ensures a == -1 ==> \result == 5;
  // Values where right-shifted a is < -4 or >4 yield 2 (excluding a=-1)
  ensures (a != -1) && ((a >> 1) < -4 || (a >> 1) > 4) ==> \result == 2;
  // Right-shifted a in [-4,-2] or [1,4] gives 3 (excluding a=-1)
  ensures (a != -1) && ((a >> 1) >= -4 && (a >> 1) <= -2) ==> \result == 3;
  ensures (a != -1) && ((a >> 1) >= 1 && (a >> 1) <= 4) ==> \result == 3;
  // Right-shifted a in [-1,0] gives maximum non-special value 4
  ensures (a != -1) && ((a >> 1) >= -1 && (a >> 1) <= 0) ==> \result == 4;
*/
int testme(int a){
  int result = 0;
  if ((a >> 1) >= -4)
    result++;
  if ((a >> 1) <= 4)
    result++;
  if ((a >> 1) > -2)
    result++;
  if ((a >> 1) <= 0)
    result++;
  if (a == -1)
    result++;
  return result;
}
```

Fig. 11. Specification generated for TestShiftRT.

on observed mappings of input values ranges to distinct outputs, the version generated without PathCrawler performs a more literal translation of the implementation into ACSL.

This direct translation is not technically incorrect, but it offers little interpretive value. In effect, it re-states the code in ACSL without offering additional insight. For a user trying to understand the behavior or verify the correctness of a program, they are essentially provided a guarantee that the program "does what it says"; they might as well read the code itself.

An observation, and maybe part of the true explanation for this behavior, across our examples is that reasoning without provided inputs - such as those provided by PathCrawler - leads the LLM to invest more effort into inventing examples on its own. This effort reduces the time and attention available for structural and logical generalization. In contrast, when supplied with example inputs, the model shifts more quickly to analyzing patterns and deriving more meaningful, generalized specifications.

Input Pattern Recognition. Another behavior we observed was Deepseek's ability to identify and leverage patterns in input data during its reasoning process. The LLM appeared to actively search for consistent properties in the input such as ordering, bounds, or structure in order to infer preconditions or better understand the underlying logic of the program.

For example, in the PathCrawler output and accompanying reasoning showed in Figure 14, the model quickly picked up on the fact that all the provided inputs were sorted. This recognition was not only mentioned in its reasoning but also incorporated into the preconditions of the generated specification.

A more subtle instance of this Deepseek's ability to pick up on abstract patterns is illustrated in the PathCrawler output and accompanying reasoning showed in Figure 15. There, Deepseek used a single example early in its reasoning trace to hypothesize a high-level intent for the program. This could factor into our observation that programs with PathCrawleroutput provided seemed to produce specifications with more abstract properties.

```
Let's analyze each case:
- For a=0:
  - (0 >> 1) is 0. So all four shift-related conditions: 0 >=-4 (yes), 0 <=4 (yes), 0 >-2 (yes), 0
      <=0 (yes). So result is 4. Then the fifth condition a==-1 is false. So total 4, which
     matches the CSV.
- For a=-1:
  - (-1 >> 1) in C is implementation-defined, but typically arithmetic shift, so for signed
     integers, right shift of a negative number fills with 1s. So -1 in binary is all 1s. Shifting
      right by 1 gives -1 again. So (a >>1) is -1.
  - Check conditions:
    1. -1 >=-4 -> yes -> +1
    2. -1 <=4 -> yes -> +1
    3. -1 >-2 -> yes -> +1 (since -1 is greater than -2)
    4. -1 <=0 -> yes -> +1. So four from shifts. Then a==-1 -> +1. Total 5, which matches CSV.
... much more reasoning
1. (a>>1) >=-4 -> -830 million >=-4 -> yes. So +1.
2. (a>>1) <=4 -> -830 million <=4 -> yes. So +1. Now result is 2.
3. (a>>1) >-2 -> -830 million >-2 -> yes? Wait, no. -830 million is less than -2. So this
     condition is false. So no increment.
4. (a>>1) <=0 -> yes, since it's negative. So +1. Now result is 3?
... more reasoning
So from the code and data, we can see that:

- When (a>>1) is in [-4,4], but depending on the other conditions.

But the conditions are not all overlapping. For example:

Condition1: (a>>1) >=-4 -> possible values from -4 to MAX (for signed int)

Condition2: (a>>1) <=4 -> possible values from MIN to 4.

So the intersection of condition1 and condition2 is [-4,4].
```

Fig. 12. Interesting snippets from the reasoning process being the specification generated for TestShiftRT with PathCrawler output in the prompt.

```
/*@
  @ ensures 0 <= \result <= 5;
  @ ensures \result == ((a >> 1 >= -4) + (a >> 1 <= 4) + (a >> 1 > -2) + (a >> 1 <= 0) + (a
     == -1));
*/
int testme(int a){
  int result = 0;
  if ((a >> 1) >= -4)
    result++;
  if ((a >> 1) <= 4)
    result++;
  if ((a >> 1) > -2)
    result++;
  if ((a >> 1) <= 0)
    result++;
  if (a == -1)
    result++;
  return result;
}
```

Fig. 13. Specification generated for TestShiftRT with no PathCrawler output in the prompt.

Small Numbers of Test Cases. One immediately noticeable trend in our experiments was that programs with only a small number of test cases, typically those with a single reachable branch, gained little benefit from PathCrawler output. In these cases, the specification generated by Deepseek was essentially the same as that generated without symbolic assistance.

```
input_elem,input_A[0],input_A[1],input_A[2],input_A[3],input_A[4],input_A[5],input_A[6],input_A[7],input_A[8],
    ↪ input_A[9],output,verdict
94,0,0,0,0,0,0,0,0,0,0,0,success
7,0,0,0,0,0,0,0,0,0,7,1,success
0,0,0,0,0,0,0,0,0,0,0,1,success
```

```
For example, the third test case has all zeros, which is trivially sorted. The second test case's
    array has zeros except the last element 7, which is sorted. The first test case's array is
    all zeros. So, perhaps the function expects the array to be sorted. But the code doesn't
    enforce this. However, for the binary search to work correctly, the array must be sorted.
    Therefore, a requires clause in ACSL should state that the array is sorted
```

Fig. 14. PathCrawler output for Bsearch.c.

```
input_len,input_buffer[0],input_buffer[1],input_buffer[2],input_buffer[3],input_msg[0],input_msg[1],output,verdict
2,0,0,0,0,0,0,0,0,unknown
3,0,0,0,0,10,46,0,0,46,46,unknown
2,0,0,0,0,10,46,10,46,46,unknown
2,0,0,0,0,10,0,10,0,unknown
```

```
Next, looking at the CSV data. The columns are input_len, input_buffer (positions 0-3), input_msg
    (positions 0-1), output, verdict. The output seems to show the buffer after processing. For
    example, in the third row, input_msg is 10 and 46 (ASCII for '\n' and '.'), and the output
    becomes 10, 46, 46. This suggests that when "\n." is encountered, the code adds an extra '.'
    to the buffer, hence three characters: '\n', '.', '.'.
```

Fig. 15. PathCrawler output and reasoning for BugKPath.c.

```
Additionally, from the CSV data, when the input message starts with "\n.", the buffer contains "\n
    ..". An ensures clause could capture this, but in ACSL, it's challenging to relate the input
    msg to the output buffer for all cases. However, based on the example, we can add an ensures
    clause that if the first two characters are "\n.", then the buffer's first three characters
    are "\n..". But this is very specific. For example:

ensures (len >= 2 && msg[0] == '\n' && msg[1] == '.') ==>
        (buffer[0] == '\n' && buffer[1] == '.' && buffer[2] == '.');

But this is just one case. It's unclear if this is desired. However, the user's examples show that
     the code adds an extra '.', so the ensures clause could mention this.
```

Fig. 16. PathCrawler output and reasoning for BugKPath.c.

This meets our expectations, as PathCrawler provided only minimal new information to the model.

Even when there were multiple paths for PathCrawler to generate inputs for, having only one test case per control path occasionally led Deepseek to incorrect conclusions. The lack of variation in the data left the model with insufficient information to accurately generalize behavior. Going back to our previous example, in the specification generated for the program `BugKPath`, shown in Figure 17 along with its reasoning in Figure 16, Deepseek inferred that a transformation only occurred at the beginning of a string. In reality, `BugKPath` was intended to replace every occurrence of "\n." in a buffer with "\n..". The model's misunderstanding stemmed from the fact that the only test case it was given featured the target pattern at the start of the input. Without additional examples, the model generalized prematurely, producing a specification that reflected the limited input rather than the actual behavior.

```
/*@
  requires len >= 0;
  requires (len > 0) ==> \valid_read(msg + (0..len-1));
  requires \valid(buffer + (0..BUFSZ-1));
  assigns buffer[0..BUFSZ-1];
  ensures (len >= 2 && \old(msg[0]) == '\n' && \old(msg[1]) == '.') ==>
          (buffer[0] == '\n' && buffer[1] == '.' && buffer[2] == '.');
*/
void testme (char *msg, int len, char *buffer)
{
  int i;
  int j;
  //char buffer[BUFSZ];
  int limit = BUFSZ - 1;

  for (i = 0; i < len; ) {
    for (j = 0; i < len && j < limit; ){

      if (i + 1 < len
          && msg[i] == '\n'
          && msg[i+1]== '.') {

        buffer[j] = msg[i]; /* Suppose j == limit - 1 */
        j++;
        i++;

        buffer[j] = msg[i]; /* Now j == limit */
        j++;
        i++;
        /* BAD */

        buffer[j] = '.';    /* Now j == limit + 1 = sizeof(buffer) */
        j++;
      } else {
        buffer[j] = msg[i];
        j++;
        i++;
      }
    }
  }
}
```

Fig. 17. PathCrawler output and reasoning for BugKPath.c.

4.6.3 PathCrawler Summary. Including PathCrawler output into the specification synthesis process had a subtle but noticeable effect on the quality of the generated specifications. In particular, specifications generated with PathCrawler displayed more abstract reasoning and better structural organization when "high quality" input/output examples were provided.

This improvement appeared to result from Deepseek's reasoning process being freed from the need to generate examples itself, allowing it to concentrate more on generalizing program behavior. High quality examples were defined as those that covered all execution paths with multiple examples per path. Without full path coverage, the control flow of the program cannot be accurately captured. Without sufficient examples per path, the model risks producing specifications tailored to a single example.

PathCrawler, as a tool, was designed to produce the minimal number of test cases required to achieve full path coverage. While this goal aligns well with test generation, it is not necessarily optimal for specification synthesis. However, there is no reason why PathCrawler or a similar tool could not be extended or adapted to better suit the needs of specification synthesis. More broadly, providing LLMs with curated input/output examples formatted for both human readability and alignment with the underlying code appears to be a promising direction for guiding specification synthesis in practice.

4.7 EVA Qualitative Analysis

EVA, a static analysis tool, specialises in conducting a comprehensive value analysis of C programs. The report that it generates includes the detailed outcomes of the value analysis but also lists alarms that signify possible runtime errors linked to these value states. Such alarms are indicators of conditions under which the program might fail or behave unexpectedly, essentially flagging areas of the code that are prone to errors due to specific input values or execution paths.

4.7.1 Avoiding Runtime Errors. In our analysis of the **eva_set**, a distinct characteristic emerges: the prevalence of precondition annotations that serve to help the function avoid runtime errors. These preconditions appear to be a direct result of the runtime alarms included the EVA reports. The alarms detail problematic input ranges, such as values that could cause the program to behave unpredictably or fail – common issues highlighted include index-out-of-bound errors, divide-by-zero errors, and integer overflows.

An example of these value domains can be seen in the specification generated for the program **Alias5**, seen in Figure 18. The specification clearly aligns with the EVA report included in its synthesis prompt.

```
...
[eva:alarm] temp_files/tmphpdyn83w/eva_temp.c:8:Warning:
  signed overflow. assert -2147483648 <= x * 2;
[eva:alarm] temp_files/tmphpdyn83w/eva_temp.c:8:Warning:
  signed overflow. assert x * 2 <= 2147483647;
[eva:alarm] temp_files/tmphpdyn83w/eva_temp.c:9:Warning:
  signed overflow. assert -2147483648 <= v - y;
[eva:alarm] temp_files/tmphpdyn83w/eva_temp.c:9:Warning:
  signed overflow. assert v - y <= 2147483647;
[eva:alarm] temp_files/tmphpdyn83w/eva_temp.c:11:Warning:
  out of bounds write. assert \valid(tab + 2);
[kernel] temp_files/tmphpdyn83w/eva_temp.c:11:Warning:
  all target addresses were invalid. This path is assumed to be dead.
```

Notice how the alarms are utilised by the LLM in generating preconditions that ensure that the program avoids both integer overflow and index-out-of-bounds errors. These precise runtime-aware annotations are largely unique to annotations generated with EVA as the reports provide a level of direction and precision absent in the baseline prompt.

4.7.2 EVA Tunnel-vision. In examining the impact of incorporating EVA reports into the specification generation process, it becomes apparent that the LLM prioritises the analysis of the EVA report over the direct reasoning about the program's implementation. This observation emerges distinctly when analysing the specifications and reasoning CoT produced by the LLM. For example, compare the specifications seen in Figures 19 and 20 where the former represents a specification generated with an EVA report in the prompt and the latter without. The specification generated in Figure 19 completely ignores all of the structure and logic that was focused on normally and instead completely focuses on eliminating runtime errors. This is also reflected in the reasoning about the EVA report displayed in Figure 21 on page 23.

In this case, we notice that while the specification in **eva_set** was able to infer that this program was a triangle classifier, its focus was clearly taken by the provided EVA report.

```
/*@
  requires -1073741824 <= x <= 1073741823; // Prevent x*2 overflow
  requires -2147483648 <= (x*2 - y) <= 2147483647; // Prevent v-y overflow
  requires \valid(tab + (0..2));            // Ensure tab[2] is accessible
  requires \valid(tab + (y +4));            // Ensure tab[y+4] access is valid
  requires \valid(tab + (2 + (x*2 - y)));   // Ensure *(pt+v) write is valid
*/
int testme(int x, int y, int tab[]) {
  int v;
  int *pt;
  int **ppt;
  v = x * 2;
  v = v - y;         /* v = (x * 2) - y */
  pt = &tab[2];
  tab[2] = x;
  ppt = &pt;
  pt = pt + 1;       /* pt = &tab[2] + 1 */
  *ppt = &tab[2];    /* pt = &tab[2] */
  *(pt + v) = y;     /* tab[2 + (x * 2) - y] = y */
  if (tab[y + 4] > 5)
    return 1;
  return 0;
}
```

Fig. 18. Alias5 specification generated with an EVA report as context.

```
/*@
  requires INT_MIN <= i + j <= INT_MAX;
  requires INT_MIN <= i + k <= INT_MAX;
  requires INT_MIN <= j + k <= INT_MAX;
  ensures \result == 1 || \result == 2 || \result == 3 || \result == 4;
*/
int testme(int i, int j, int k){
  int type_code;
  if ((i == 0) || (j == 0) || (k == 0)) type_code = 4;      /* line 13 */
  else {
    type_code = 0;
    if (i == j) type_code = type_code + 1;                  /* line 16 */
    if (i == k) type_code = type_code + 2;                  /* line 17 */
    if (j == k) type_code = type_code + 3;                  /* line 18 */
    if (type_code == 0){                                    /* line 19 */
      if ((i+j <= k) || (j+k <= i) || (i+k <= j))           /* line 20 */
  type_code = 4;
      else
  type_code = 1;
      }
    else if (type_code > 3) type_code = 3;                  /* line 25 */
    else if ((type_code == 1) && (i+j > k)) type_code = 2;  /* line 26 */
    else if ((type_code == 2) && (i+k > j)) type_code = 2;  /* line 27 */
    else if ((type_code == 3) && (j+k > i)) type_code = 2;  /* line 28 */
    else type_code = 4;
    }
  return type_code;
}
```

Fig. 19. Tritype specification generated with an EVA report as context.

This observation raises concerns about LLM-based specification synthesizers' ability to balance the input from static analysis tools like EVA that provide clear direction for the LLM. While the safety and domain boundaries are well-captured, the essence of what the program is designed to do can sometimes be overshadowed by the focus on avoiding runtime errors and handling edge cases as dictated by the EVA report.

```
/*@
  ensures 1 <= \result <= 4;
  ensures \result == 4 <==> (i == 0 || j == 0 || k == 0 || (i + j <= k) || (j + k <= i) || (i
      + k <= j));
  ensures \result == 3 <==> (i != 0 && j != 0 && k != 0 && i == j && j == k);
  ensures \result == 2 <==> (i != 0 && j != 0 && k != 0 &&
    ((i == j && i + j > k) || (i == k && i + k > j) || (j == k && j + k > i)));
  ensures \result == 1 <==> (i != 0 && j != 0 && k != 0 &&
    i != j && i != k && j != k && (i + j > k) && (j + k > i) && (i + k > j));
*/
int testme(int i, int j, int k){
  int type_code;
  if ((i == 0) || (j == 0) || (k == 0)) type_code = 4;      /* line 13 */
  else {
    type_code = 0;
    if (i == j) type_code = type_code + 1;                 /* line 16 */
    if (i == k) type_code = type_code + 2;                 /* line 17 */
    if (j == k) type_code = type_code + 3;                 /* line 18 */
    if (type_code == 0){                                   /* line 19 */
      if ((i+j <= k) || (j+k <= i) || (i+k <= j))          /* line 20 */
  type_code = 4;
      else
  type_code = 1;
      }
    else if (type_code > 3) type_code = 3;                 /* line 25 */
    else if ((type_code == 1) && (i+j > k)) type_code = 2; /* line 26 */
    else if ((type_code == 2) && (i+k > j)) type_code = 2; /* line 27 */
    else if ((type_code == 3) && (j+k > i)) type_code = 2; /* line 28 */
    else type_code = 4;
    }
  return type_code;
}
```

Fig. 20. Tritype specification generated with the baseline prompt.

```
The provided C program is a function `testme` that takes three integer inputs `Side1`, `Side2`,
     and `Side3` and uses these inputs to calculate and return a value `triOut`.

From the EVA report, we can see several warnings about potential signed overflow in the program.
     This occurs when the sum of two sides of the triangle is close to the maximum value an
     integer can hold (`2147483647`), leading to a possible overfl
```

Fig. 21. Reasoning performed about an EVA report run on Tritype.c.

4.7.3 EVA Summary. The effects of adding EVA reports to our prompts were more clear than those of PathCrawler. EVA's analysis produced a direct mapping from its runtime alarms to ACSL preconditions, which the LLM translated consistently and reliably.

This raises a question: do we even need an LLM for this task? Since EVA's alarms can be mechanically interpreted as input restrictions, one could imagine writing a dedicated tool to convert them directly into `requires` clauses. To some extent, this is a valid point when the symbolic output aligns cleanly with the target specification language.

However, this assumes the existence of a purpose-built translation tool for every symbolic analyzer and every specification formalism used in practice. Such tools are rarely developed comprehensively and are unlikely to exist across the full combinatorial space of symbolic tools and specification languages.

Instead, what we demonstrate with EVA is how easily LLMs can flexibly incorporate the output of symbolic analysis without needing custom integration. In our case, we simply appended the raw EVA report to the LLM prompt, and the model responded with annotations that captured the relevant constraints effectively. This showcases the adaptability of LLM-based specification synthesis in heterogeneous toolchains.

4.8 Quantitative Analysis with EVA

Ideally, we would like large language models to generate deductive verification specifications that can be formally proved with tools such as Frama-C's WP [13], since this represents a key step toward the broader goal of specification synthesis. Achieving this, however, requires either that suitable loop invariants are provided alongside the specifications, or that one considerably limits the expressivity of specifications.

At the same time, formally proving specifications is not the only way to assess whether or not they are suitable for the target program. In this article, our focus is not primarily on the accuracy of the specifications themselves, but rather on the intent behind the generated specifications and how that intent can be influenced through symbolic analysis. To this end, we side-step the challenge of proving specifications with deductive verifiers by using another tool that is able to utilize ACSL - EVA.

4.8.1 Experiment Setup. The goal of this experiment is to measure whether adding symbolic context to a prompt reduces the number of runtime alarms reported by EVA. The key mechanism that makes this experiment effective is that EVA interacts with provided ACSL specifications. To test this, we take the ACSL annotations generated in our three sets - **baseline_set**, **pathcrawler_set**, and **eva_set** - and re-run EVA on each of them. EVA uses provided **requires** clauses to constrain the input values considered during its Value Analysis, and it attempts to prove provided specifications using this analysis, providing an alternative methodology to deductive verification. However, in this quantitative analysis, we only take advantage of the input constraining feature. Differences in the number and nature (e.g., integer overflow) of alarms can serve as an indicator of how the added symbolic context influences the resulting specifications.

4.8.2 Results. Based on our qualitative analysis of **eva_set** we expected a substantial reduction in the number of alarms generated by EVA during the second run. In the analysis, Deepseek-R1 primarily focused on eliminating runtime alarms by constraining input ranges. However, our results, shown in Figure 22 were more nuanced than anticipated. While there was a significant reduction in alarms - approximately 30% - this was less than expected given how thoroughly Deepseek-R1 appeared to address alarms, particularly those related to integer overflow and invalid memory accesses. Although specifications in **eva_set** performed best at reducing alarms, they were only slightly more effective than those in **pc_set**.

4.8.3 Explaining the Discrepancy: Integer Overflows. Explaining this discrepancy requires some understanding of how EVA performs abstract interpretation. At its core, the issue arises from the way Deepseek-R1 expresses preconditions. A simple constraint such as the interval function in Figure 23 is non-relational: each variable is constrained independently, and this can be easily represented by interval domains used by EVA. By contrast, a relational constraint involves multiple variables in the same condition. Relational constraints cannot be represented by EVA's default non-relational domains, so they are effectively ignored during state reduction. (While relational domains like octagons are also supported in principle, EVA does not currently use them to reduce states based on ACSL contracts.) For more details, we refer the reader to [9, 10].

Examining the annotations generated in **eva_set**, we found that most of the contracts written by Deepseek-R1 to prevent integer overflows were of relational form. This explains the discrepancy between the expected and observed alarm reduction: EVA was unable to use the generated annotations.

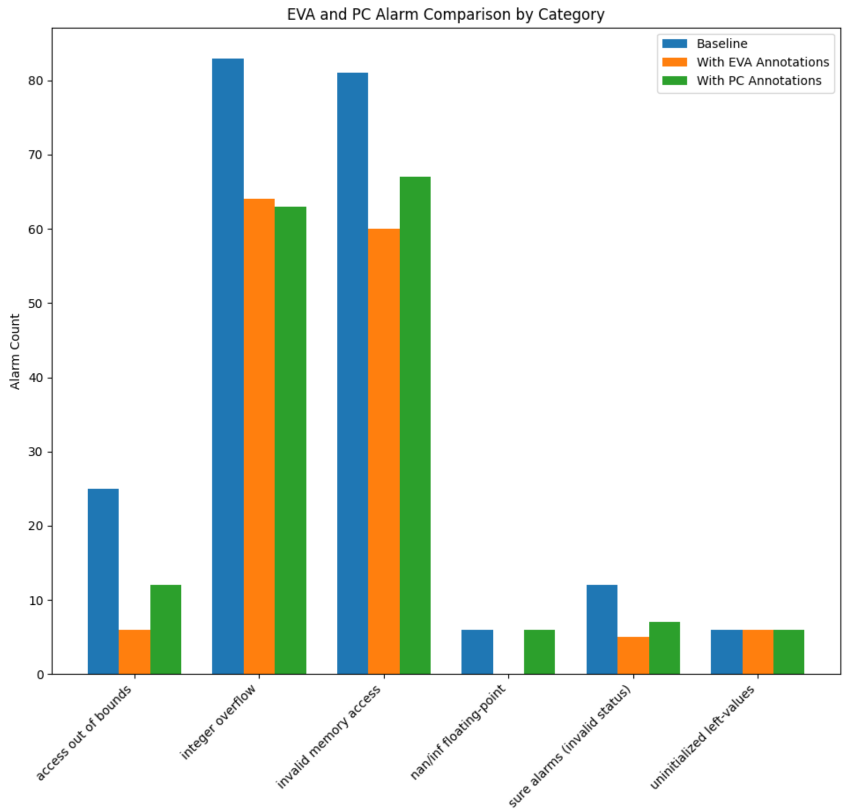

Fig. 22. EVA alarms after annotations were added.

```
/*@ requires 0 < x < 10; */
int interval(int x){
  return x + x;
}

/*@ requires 0 < x + y < 10; */
int relational(int x, int y){
  return x + y;
}
```

Fig. 23. ACSL Contracts with a **interval** and **relational** constraints.

4.8.4 Explaining the Discrepancy: Invalid Memory Accesses. Similar to the case with integer overflows, the reason why Deepseek-R1's annotations did not eliminate as many invalid memory access alarms as we expected requires knowledge of the limitations and theoretical foundations of EVA's analysis . At first glance, EVA gives the impression of being a standalone static analyzer that can run directly on C code without ACSL support. While EVA can indeed compute useful value analyses without annotations, its effectiveness often depends on additional guidance and integration with the larger Frama-C ecosystem. Much like in deductive verification, successful elimination of alarms frequently requires help. In the cases where invalid memory access alarms persisted despite annotations, the primary cause was insufficient information about loop behavior. Without loop invariants or other annotations that constrain pointer ranges, EVA was unable to conclude that array accesses were always within bounds.

4.8.5 Quantitative Analysis Conclusions. Using EVA as additional context for Deepseek-R1 to generate ACSL specifications was both a success and a reminder of the limitations of LLMs without human guidance. On the positive side, Deepseek-R1 was able to utilize EVA reports to express a more specific *intent* in generated specifications. This was the initial goal of these experiments.

However, it was clear from the quantitative analysis that Deepseek-R1 appears to have only a surface-level understanding of working with a technical tool like EVA. While it could interpret alarms and adjust specifications in ways that are intuitive to human eyes, it lacked the deeper procedural knowledge required to address more complex issues, such as abstract domain knowledge and loop behavior. Interestingly, this aspect closely mirrors what is difficult also for less experienced *human* users.

There is no fundamental reason an LLM could not be fine-tuned to handle the intricacies of using EVA or other Frama-C plugins effectively. But as they stand, general-purpose models such as Deepseek-R1 are far from experts. For now, LLMs can serve as helpful assistants within the Frama-C ecosystem, offering suggestions and reducing some manual effort. They are not, however, an alternative to learning to use and understand the underlying principles of these tools.

5 Related Work

There is an ever-growing body of work exploring the opportunities of combining LLMs with various tools for formal-methods and theorem proving.

5.1 Theorem Provers

In the domain of proof-assistants, a few works have explored the task of synthesising properties or lemmas using neural methods [2, 29, 50, 61] with varying success. More focus has been on training models for generating proofs, with applications to most mainstream proof assistants like Isabelle/HOL, Coq, and Lean [5, 17, 25, 28, 69]. Recent developments of proof co-pilots for Lean and Coq are ongoing, aiming at making next-tactic suggestions for the user while creating proof scripts [11, 33, 57, 63].

5.2 Specifications from Natural Language

Significant research has focused on translating informal **natural language** (**NL**), such as human-written descriptions or documentation, into formal specifications for verification. This is a natural area of research, as specification authors often lack expertise in formal specification languages. The process is relatively straightforward, as the intent of the specification is already expressed in NL, requiring only translation into a formal representation. Studies have explored mapping NL to various specification domains [16, 23, 37–39, 41, 45, 51, 68, 71]. Murphy et al. [47] propose a notable approach, guiding the LLM by preprocessing NL before generating formal specifications.

5.3 Specifications from Code

Specification synthesis from code is the area most closely related to our work, as we similarly aim at generating formal specifications from source code, augmented with information produced by symbolic analysis tools. Much of the prior research in this domain has concentrated on the synthesis of loop invariants (e.g., [27, 30, 42, 44, 48, 67]), typically under the assumption that preconditions and postconditions are already provided. This emphasis is in part due to the inherent subjectivity of contracts, and partly due to the difficulty in manually writing loop invariants. Other research areas similar to generating loop invariants involve the insertion of intermediate assertions to assist automated tools such as Dafny [46, 49, 56]. A common theme among all of these works is that **the contract is provided** and the part of the specification that is synthesized is meant only to further the verification.

The research most closely aligned ours is that which focuses on the automated generation of specification **contracts** instead of invariants or insertions, which rely on pre-existing contracts. Greiner et al. [20] focus on generating **Java Modeling Language** (**JML**) contracts for Java programs by fine-tuning models on a dataset of Java methods annotated with JML contracts. The large number of JML examples enables the models to infer contracts from code structure alone. Wen et al. [64], similar to our research, also generate contracts with the help of static analysis. However, instead of attempting to guide the *intent* of the LLM, they leverage this information to decompose large or complex functions into smaller, logically coherent components. Our research, instead, uses context from symbolic methods to change the *types of contracts* generated by LLMs.

5.4 LLM Reasoning

This article qualitatively analyzes how DeepSeek R1, a reasoning-focused LLM, interprets program semantics using human-in-the-loop evaluation. Beyond qualitative approaches, benchmarks like FormalBench [36] and DafnyBench [43] and CLEVER [59] assess LLMs' abilities in formal methods. CRUXEval [22], takes a more general approach and measures an LLM's ability to reason about the execution semantics of Python programs and is extended by the "The Counterfeit Conundrum" [21] which takes buggy code into account. Finally, Shojaee et al. [54] challenge whether "reasoning" LLMs truly reason or rely on pattern matching. These works frame our investigation into LLM reasoning about programs and specifications.

6 Limitations

6.1 LLMs

LLMs remain a relatively new technology, and obtaining reproducible results from them is still a somewhat open topic, particularly when accessed through hosted services such as Deepseek or OpenAI. These services often operate behind proprietary APIs, and it is difficult to know what internal updates or changes may be introduced over time. This introduces a potential source of variability in results that cannot be easily controlled or accounted for.

In this work, we employed elevated temperature settings to encourage more diverse and creative outputs from the model. While this approach improves specification coverage and variety, it also increases randomness and reduces reproducibility. There is a tradeoff here: the traits that make LLMs effective for synthesis, creativity and flexibility, also make them more difficult to evaluate.

Despite these limitations, we were generally able to reproduce, using Deepseek-R1, the overall findings of our earlier experiments conducted with GPT-4 [19] (for as far as the experiments of this work overlap with the experiments in [19]). However, it is important to note that our qualitative analysis could be subject to confirmation bias. Since we had prior expectations from earlier results, there is an inherent risk that we interpreted or evaluated outcomes through the lens of those expectations during this second round of analysis.

6.2 Datasets

While the Intent Dataset used in this work was fully handcrafted and tailored to our experimental objectives, the PathCrawler dataset was not. The suite of programs included in the PathCrawler Set was originally developed to exercise the capabilities and edge cases of the PathCrawler tool itself. As such, a significant portion of these programs were not designed with specification synthesis in mind and do not always lend themselves to interesting specifications.

The advantage of using the PathCrawler Set was that it came with infrastructure for symbolic execution already in place. Specifically, the programs included input constraint files and configuration settings that enabled PathCrawler to explore paths effectively and generate inputs within meaningful bounds. This allowed us to test the integration of symbolic tools without first needing to develop a large number of compatible test cases from scratch. Moreover, the PathCrawler Set was closed source, which limits the danger of LLMs being contaminated with this data prior to our experiments.

Nonetheless, it is important to recognize that the total dataset used in this study consisted of approximately 50 programs, and only a subset of these were structured in a way that allowed for particularly expressive specifications. This limits the generality of our findings and highlights the need for future work on curated datasets designed specifically for evaluating specification synthesis across a broader spectrum of program behavior.

6.3 Tools and Languages

The approach presented in this article allows for experimentation with various combinations of programming languages and symbolic analysis tools. Different pairings could potentially yield equally significant and relevant results. As evidenced by prior work on specification generation for languages such as Dafny and Java discussed in the related works Section 5.3, there is work to be done with specification synthesis in those domains. However, to manage scope and allocate sufficient time for qualitative analysis, this study focused on a single domain. The C language was selected due to its extensive ecosystem of analysis tools, with Frama-C's suite being particularly well-suited for our analytical objectives.

Additional experimentation could involve integrating multiple LLMs in a more agentic system. For instance, one LLM could analyze a program and provide contextual insights to another LLM tasked with generating specifications. While this approach is valid, it risks introducing AI bias, potentially compromising the reliability of the results. To mitigate this, our study prioritizes symbolic tools, leveraging their precision and deterministic nature to ensure robust and accurate outcomes.

7 Future Work

While there are many ways to expand on this work, we highlight three especially interesting directions: models, datasets, and tooling.

Models. We have focused on general-purpose LLMs hosted by providers such as Deepseek (here) and OpenAI (in [19]). Inspired by recent efforts in the theorem proving community to fine-tune models for lemma generation or tactic prediction, we aim at experimenting with fine-tuning models specifically for contract-based deductive verification. This could involve training on large dataset of programs annotated with ACSL specifications or similar formal annotations, with the goal of improving the model's ability to generate and reason about ACSL specifications. If this is successful, the method is likely to carry over to other specification languages, e.g., JML for Java.

Datasets. In early experiments, we attempted to incorporate verification as part of our evaluation. However, earlier LLMs struggled with syntactic correctness and often with generating necessary loop invariants. These limitations led to largely uninteresting results. Moving forward, we hope to design a dataset that includes a larger set of programs that are realistically verifiable, possibly focusing on real-world, loop-free examples to start with. Such a dataset would allow us to revisit the question of whether specification synthesis from LLMs can be effectively integrated into verification pipelines.

Tooling. Our current work integrates symbolic outputs using two tools from the Frama-C ecosystem: PathCrawler and EVA. However, many other tools exist in the formal methods community that could provide useful results for specification synthesis. Whether through prompt-level integration, preprocessing, or post-processing, we see potential in exploring how the output of these tools can be used to enhance LLM-based synthesis.

While we use EVA to detect runtime bugs, a variety of other tools focus on more specialized runtime errors, such as memory and security-based issues. For instance, tools like Memcheck [53] for C or Infer [24] for Java are plugins for identifying memory-specific issues. Similarly, security analysis tools like SpotBugs [58] for Java can identify vulnerabilities. However, these tools, while effective for bug detection, often generate false positives [24].

One promising synergy involves pairing static analysis tools with model checkers like CBMC [34] or CPAchecker [4]. By allowing users to define in-line assertions within the code structure, these model checkers can verify alerts generated by static analysis, offering a second layer of mechanical validation. Additionally, model checkers offer support for reasoning about concurrent programs. Running a tool like Helgrind [26] prior to specification synthesis could guide the creation of specifications tailored to concurrency-related properties.

8 Conclusion

This article addresses two core challenges in LLM-based specification synthesis: the high variability in generated specifications and the possibility of conflicting implementation and intent in a given program. Using Deepseek-R1, a reasoning-focused LLM, we found that the model is well-suited to reasoning about subtle discrepancies between what code does and what it is intended to do. Across a range of programs, Deepseek consistently identified syntactically subtle but semantically significant bugs and responded to prompt directives that emphasized the program's intent over its buggy implementation.

To reduce the variability in specifications, we explored whether symbolic output from formal tools could guide the model toward generating more consistent and directed specifications. We appended symbolic information from two Frama-C tools, PathCrawler and EVA, directly to the prompts and observed changes in the resulting specifications. PathCrawler, by providing input/output examples, appeared to shift the model's attention toward structural correctness and abstract logical relationships. Deepseek often used these examples as a substitute for self-generated test cases, allowing it to focus more directly on program logic. However, the utility of this approach depended heavily on the quality and quantity of the generated test cases.

In contrast, EVA's static value analysis yielded a different effect. By identifying potential runtime errors through value analysis, EVA prompted Deepseek to generate preconditions that would prevent those errors, resulting in more safety-focused specifications. While this led to stronger guarantees about program safety, it often came at the expense of postconditions.

Together, these results suggest that neurosymbolic prompting provides a viable path toward controlling both the structure and focus of LLM-generated specifications. By influencing the synthesis process with formal tool outputs, we can steer generation toward specific kinds of properties, helping address key issues of alignment and variability.

Our experiments seeking to measure the **effectiveness** of the generated ACSL were less promising. Deepseek-R1 successfully adapted its specifications based on EVA's alarms, indicating that symbolic feedback can influence intent-aware specification synthesis. However, the model's understanding of EVA was superficial; it could respond to warnings but lacked the deeper procedural knowledge needed to resolve complex cases, which required an understanding of the tool's limitations. This suggests that while LLMs can serve as useful assistants in the Frama-C ecosystem, they do not fully substitute expert understanding.

This work is aligned to the vision of trustworthy triple copiloting of implementations, tests, and specifications ("TriCo"), as co-outlined by two of the authors in [1]. However, in that vision article, we focused on the bilateral relations of the three artefacts, whereas here, when adding the

Pathcrawler output, we use implementations and tests at once when generating specifications. More generally, we see our work as a contribution to the more general aim of combining the complementary strengths of machine learning and exact analyses for effective and reliable development of trustworthy software.

References

[1] Wolfgang Ahrendt, Dilian Gurov, Moa Johansson, and Philipp Rümmer. 2022. TriCo–triple co-piloting of implementation, specification and tests. In *Proceedings of the International Symposium on Leveraging Applications of Formal Methods*. Springer, 174–187.

[2] Yousef Alhessi, Sólrún Halla Einarsdóttir, George Granberry, Emily First, Moa Johansson, Sorin Lerner, and Nicholas Smallbone. 2025. Lemmanaid: Neuro-Symbolic Lemma Conjecturing. arXiv:2504.04942. Retrieved from https://arxiv.org/abs/2504.04942

[3] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. 2008. *ACSL: ANSI/ISO C Specification Language*. CEA LIST, Saclay, France.

[4] Dirk Beyer and M. Erkan Keremoglu. 2011. CPAchecker: A tool for configurable software verification. In *Proceedings of the International Conference on Computer Aided Verification*. Springer, 33 (2011), 184–190.

[5] Lasse Blaauwbroek, Josef Urban, and Herman Geuvers. 2020. The tactician – a seamless, interactive tactic learner and prover for coq. In *Proceedings of the Intelligent Computer Mathematics*. Christoph Benzmüller and Bruce Miller (Eds.), Springer International Publishing, Cham, 271–277.

[6] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. 2018. Translating code comments to procedure specifications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 242–253.

[7] Sandrine Blazy, David Bühler, and Boris Yakobowski. 2017. Structuring abstract interpreters through state and value abstractions. In *18th International Conference on Verification Model Checking and Abstract Interpretation (Proceedings of the International Conference on Verification Model Checking and Abstract Interpretation, Vol. 10145 LNCS)*. Paris, France, 112–130. DOI: https://doi.org/10.1007/978-3-319-52234-0_7

[8] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in Neural Information Processing Systems* 33 (2020), 1877–1901.

[9] David Bühler. 2017. Structuring an Abstract Interpreter with the Eva Plug-in. Presentation at Frama-C and SPARK Day. Retrieved July 2025 from https://www.frama-c.com/download/framaCDay/FCSD17/talk/11_Buhler.pdf

[10] David Bühler, Pascal Cuoq, and Boris Yakobowski. 2024. *Eva: The Evolved Value Analysis Plug-in – User Manual (Frama-C 26.1 Iron)*. CEA, List. Retrieved from https://frama-c.com/download/eva-manual-26.1-Iron.pdf

[11] Pedro Carrott, Nuno Saavedra, Kyle Thompson, Sorin Lerner, João F Ferreira, and Emily First. 2024. CoqPyt: Proof navigation in Python in the era of LLMs. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 637–641.

[12] Codeforces. 2025. Codeforces: Programming Competitions and Contests, Programming Community. Retrieved July 23, 2025 from https://codeforces.com/

[13] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-C: A software analysis perspective. In *Proceedings of the International Conference on Software Engineering and Formal Methods*. Springer, 233–247.

[14] Jennifer A. Davis, Matthew Clark, Darren Cofer, Aaron Fifarek, Jacob Hinchman, Jonathan Hoffman, Brian Hulbert, Steven P. Miller, and Lucas Wagner. 2013. Study on the barriers to the industrial adoption of formal methods. In *Formal Methods for Industrial Critical Systems: 18th International Workshop, FMICS 2013, Madrid, Spain, September 23-24, 2013. Proceedings 18*. Springer, 63–77.

[15] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69, 1-3 (2007), 35–45.

[16] Wen Fan, Marilyn Rego, Xin Hu, Sanya Dod, Zhaorui Ni, Danning Xie, Jenna DiVincenzo, and Lin Tan. 2024. Evaluating the ability of large language models to generate verifiable specifications in verifast. arXiv:2411.02318. Retrieved from https://arxiv.org/abs/2411.02318

[17] Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: Whole-proof generation and repair with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. (San Francisco, CA, USA,). Association for Computing Machinery, New York, NY, USA, 1229–1241. DOI: https://doi.org/10.1145/3611643.3616243

[18] Cormac Flanagan, Rajeev Joshi, and K. Rustan M. Leino. 2001. Annotation inference for modular checkers. *Information Processing Letters* 77, 2-4 (2001), 97–108.

[19] George Granberry, Wolfgang Ahrendt, and Moa Johansson. 2024. Specify what? Enhancing neural specification synthesis by symbolic methods. In *Proceedings of the International Conference on Integrated Formal Methods*. Springer, 307–325.

[20] Sandra Greiner, Noah Bühlmann, Manuel Ohrndorf, Christos Tsigkanos, Oscar Nierstrasz, and Timo Kehrer. 2024. Automated generation of code contracts: Generative AI to the rescue?. In *Proceedings of the 23rd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. 1–14.

[21] Alex Gu, Wen-Ding Li, Naman Jain, Theo X Olausson, Celine Lee, Koushik Sen, and Armando Solar-Lezama. 2024. The counterfeit conundrum: Can code language models grasp the nuances of their incorrect generations? arXiv:2402.19475. Retrieved from https://arxiv.org/abs/2402.19475

[22] Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I. Wang. 2024. Cruxeval: A benchmark for code reasoning, understanding and execution. arXiv:2401.03065. Retrieved from https://arxiv.org/abs/2401.03065

[23] Christopher Hahn, Frederik Schmitt, Julia J. Tillman, Niklas Metzger, Julian Siber, and Bernd Finkbeiner. 2022. Formal specifications from natural language. arXiv:2206.01962. Retrieved from https://arxiv.org/abs/2206.01962

[24] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling static analyses at facebook. *Communications of the ACM* 62, 8 (2019), 62–70.

[25] Jilin Hu, Jianyu Zhang, Yongwang Zhao, and Talia Ringer. 2025. HybridProver: Augmenting theorem proving with LLM-driven proof synthesis and refinement. arXiv:2505.15740. Retrieved from https://arxiv.org/abs/2505.15740

[26] Ali Jannesari, Kaibin Bao, Victor Pankratius, and Walter F. Tichy. 2009. Helgrind+: An efficient dynamic race detector. In *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 1–13.

[27] Christian Janßen, Cedric Richter, and Heike Wehrheim. 2024. Can ChatGPT support software verification?. In *Proceedings of the Fundamental Approaches to Software Engineering*. Dirk Beyer and Ana Cavalcanti (Eds.), Springer Nature Switzerland, Cham, 266–279.

[28] Albert Qiaochu Jiang, Wenda Li, Szymon Tworkowski, Konrad Czechowski, Tomasz Odrzygóźdź, Piotr Miłoś, Yuhuai Wu, and Mateja Jamnik. 2022. Thor: Wielding hammers to integrate language models and automated theorem provers. In *Proceedings of the Advances in Neural Information Processing Systems*. Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (Eds.), Retrieved from https://openreview.net/forum?id=fUeOyt-2EOp

[29] Moa Johansson and Nicholas Smallbone. 2023. Exploring mathematical conjecturing with large language models. In *Proceedings of the 17th International Workshop on Neural-Symbolic Learning and Reasoning*.

[30] Adharsh Kamath, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K. Lahiri, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. 2023. Finding inductive loop invariants using large language models. arXiv:2311.07948. Retrieved from https://arxiv.org/abs/2311.07948

[31] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A software analysis perspective. *Formal Aspects of Computing* 27, 3 (2015), 573–609. DOI: https://doi.org/10.1007/s00165-014-0326-7

[32] Nikolai Kosmatov, Virgile Prevosto, and Julien Signoles. 2024. *Guide to Software Verification with Frama-C*. Springer Cham. DOI: https://doi.org/10.1007/978-3-031-55608-1

[33] Andrei Kozyrev, Gleb Solovev, Nikita Khramov, and Anton Podkopaev. 2024. CoqPilot, a plugin for LLM-based generation of proofs. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 2382–2385.

[34] Daniel Kroening and Michael Tautschnig. 2014. CBMC–C bounded model checker: (Competition contribution). In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 389–391.

[35] Sophie Lathouwers and Marieke Huisman. 2024. Survey of annotation generators for deductive verifiers. *Journal of Systems and Software* (2024), 111972.

[36] Thanh Le-Cong, Bach Le, and Toby Murray. 2025. Can LLMs reason about program semantics? a comprehensive evaluation of LLMs on formal specification inference. arXiv:2503.04779. Retrieved from https://arxiv.org/abs/2503.04779

[37] Iat Tou Leong and Raul Barbosa. 2023. Translating natural language requirements to formal specifications: A study on gpt and symbolic nlp. In *Proceedings of the 2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops*. IEEE, 259–262.

[38] Iat Tou Leong and Raul Barbosa. 2024. Translating meaning representations to behavioural interface specifications. *Journal of Systems and Software* 211 (2024), 112009.

[39] Mengming Li, Wenji Fang, Qijun Zhang, and Zhiyao Xie. 2024. Specllm: Exploring generation and review of vlsi design specification with large language model. arXiv:2401.13266. Retrieved from https://arxiv.org/abs/2401.13266

[40] Hongwei Liu, Zilong Zheng, Yuxuan Qiao, Haodong Duan, Zhiwei Fei, Fengzhe Zhou, Wenwei Zhang, Songyang Zhang, Dahua Lin, and Kai Chen. 2024. Mathbench: Evaluating the theory and application proficiency of llms with a hierarchical mathematics benchmark. arXiv:2405.12209. Retrieved from https://arxiv.org/abs/2405.12209

[41] Jason Xinyu Liu, Ziyi Yang, Benjamin Schornstein, Sam Liang, Ifrah Idrees, Stefanie Tellex, and Ankit Shah. 2022. Lang2ltl: Translating natural language commands to temporal specification with large language models. In *Proceedings of the Workshop on Language and Robotics at CoRL 2022*.

[42] Ruibang Liu, Guoqiang Li, Minyu Chen, Ling-I. Wu, and Jingyu Ke. 2024. Enhancing automated loop invariant generation for complex programs with large language models. arXiv:2412.10483. Retrieved from https://arxiv.org/abs/2412.10483

[43] Chloe Loughridge, Qinyi Sun, Seth Ahrenbach, Federico Cassano, Chuyue Sun, Ying Sheng, Anish Mudide, Md Rakib Hossain Misu, Nada Amin, and Max Tegmark. 2024. DafnyBench: A benchmark for formal software verification. arXiv:2406.08467. Retrieved from https://arxiv.org/abs/2406.08467

[44] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2024. Specgen: Automated generation of formal program specifications via large language models. arXiv:2401.08807. Retrieved from https://arxiv.org/abs/2401.08807

[45] Md Rakib Hossain Misu, Cristina V. Lopes, Iris Ma, and James Noble. 2024. Towards ai-assisted synthesis of verified dafny methods. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 812–835.

[46] Eric Mugnier, Emmanuel Anaya Gonzalez, Ranjit Jhala, Nadia Polikarpova, and Yuanyuan Zhou. 2024. Laurel: Generating Dafny Assertions Using Large Language Models. arXiv 2405.16792. Retrieved from https://arxiv.org/abs/2405.16792

[47] William Murphy, Nikolaus Holzer, Nathan Koenig, Leyi Cui, Raven Rothkopf, Feitong Qiao, and Mark Santolucito. 2024. Guiding LLM temporal logic generation with explicit separation of data and control. arXiv:2406.07400. Retrieved from https://arxiv.org/abs/2406.07400

[48] João Pascoal Faria, Emanuel Trigo, and Rui Abreu. 2025. Automatic generation of loop invariants in dafny with large language models. In *Proceedings of the International Conference on Fundamentals of Software Engineering*. Springer, 138–154.

[49] Gabriel Poesia, Chloe Loughridge, and Nada Amin. 2024. dafny-annotator: AI-assisted verification of dafny programs. arXiv:2411.15143. Retrieved from https://arxiv.org/abs/2411.15143

[50] Markus N. Rabe, Dennis Lee, Kshitij Bansal, and Christian Szegedy. 2021. Mathematical reasoning via self-supervised skip-tree training. In *Proceedings of the ICLR*.

[51] Lasse M. Reinpold, Marvin Schieseck, Lukas P. Wagner, Felix Gehlhoff, and Alexander Fay. 2024. Exploring LLMs for verifying technical system specifications against requirements. arXiv:2411.11582. Retrieved from https://arxiv.org/abs/2411.11582

[52] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A concolic unit testing engine for C. *SIGSOFT Softw. Eng. Notes* 30, 5 (2005), 263–272. DOI: https://doi.org/10.1145/1095430.1081750

[53] Julian Seward and Nicholas Nethercote. 2005. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the USENIX Annual Technical Conference, General Track*. 17–30.

[54] Parshin Shojaee, Iman Mirzadeh, Keivan Alizadeh, Maxwell Horton, Samy Bengio, and Mehrdad Farajtabar. 2025. The illusion of thinking: Understanding the strengths and limitations of reasoning models via the lens of problem complexity. arXiv:2506.06941. Retrieved from https://arxiv.org/abs/2506.06941

[55] Julien Signoles. 2011. *E-ACSL: Executable ANSI/ISO C Specification Language*. Retrieved July 2025 from http://frama-c.com/download/e-acsl/e-acsl.pdf

[56] Álvaro F. Silva, Alexandra Mendes, and João F. Ferreira. 2024. Leveraging large language models to boost Dafny's developers productivity. In *Proceedings of the 2024 IEEE/ACM 12th International Conference on Formal Methods in Software Engineering*. (, Lisbon, Portugal,). Association for Computing Machinery, New York, NY, USA, 138–142. DOI: https://doi.org/10.1145/3644033.3644374

[57] Peiyang Song, Kaiyu Yang, and Anima Anandkumar. 2024. Towards large language models as copilots for theorem proving in lean. arXiv:2404.12534. Retrieved from https://arxiv.org/abs/2404.12534

[58] SpotBugs Team. 2025. SpotBugs: Find Bugs in Java Programs. Retrieved July 2025 from https://spotbugs.github.io/

[59] Amitayush Thakur, Jasper Lee, George Tsoukalas, Meghana Sistla, Matthew Zhao, Stefan Zetzsche, Greg Durrett, Yisong Yue, and Swarat Chaudhuri. 2025. CLEVER: A curated benchmark for formally verified code generation. arXiv:2505.13938. Retrieved from https://arxiv.org/abs/2505.13938

[60] Benjamin Tyler. 2023. Formal methods adoption in industry: An experience report. In *Proceedings of the Formal Methods for an Informal World: ICTAC 2021 Summer School, Virtual Event, Astana, Kazakhstan, September 1–7, 2021, Tutorial Lectures*. Antonio Cerone (Ed.), Springer International Publishing, Cham, 152–161. DOI: https://doi.org/10.1007/978-3-031-43678-9_5

[61] Josef Urban and Jan Jakubův. 2020. First neural conjecturing datasets and experiments. In *Proceedings of the CICM*. DOI: https://doi.org/10.1007/978-3-030-53518-6_24

[62] Hongru Wang, Rui Wang, Fei Mi, Zezhong Wang, Ruifeng Xu, and Kam-Fai Wong. 2023. Chain-of-thought prompting for responding to in-depth dialogue questions with LLM. arXiv:2305.11792. Retrieved from https://arxiv.org/abs/2305.11792

[63] Sean Welleck and Rahul Saha. 2023. LLMSTEP: LLM proofstep suggestions in Lean. arXiv:2310.18457. Retrieved from https://arxiv.org/abs/2310.18457

[64] Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. 2024. Enchanting program specification synthesis by large language models using static analysis and program verification. In *Proceedings of the International Conference on Computer Aided Verification*. Springer, 302–328.

[65] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. 2023. A prompt pattern catalog to enhance prompt engineering with ChatGPT. arXiv:2302.11382. Retrieved from https://arxiv.org/abs/2302.11382

[66] Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. 2005. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In *Proceedings of the European Dependable Computing Conference*. Springer, 281–292.

[67] Guangyuan Wu, Weining Cao, Yuan Yao, Hengfeng Wei, Taolue Chen, and Xiaoxing Ma. 2024. LLM meets bounded model checking: Neuro-symbolic loop invariant inference. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 406–417.

[68] Yilongfei Xu, Jincao Feng, and Weikai Miao. 2024. Learning from failures: Translation of natural language requirements into linear temporal logic with large language models. In *Proceedings of the 2024 IEEE 24th International Conference on Software Quality, Reliability and Security*. IEEE, 204–215.

[69] Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. 2023. LeanDojo: Theorem proving with retrieval-augmented language models. In *Proceedings of the Neural Information Processing Systems*.

[70] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. 2019. Hellaswag: Can a machine really finish your sentence? arXiv:1905.07830. Retrieved from https://arxiv.org/abs/1905.07830

[71] Mengyan Zhao, Ran Tao, Yanhong Huang, Jianqi Shi, Shengchao Qin, and Yang Yang. 2024. NL2CTL: Automatic generation of formal requirements specifications via large language models. In *Proceedings of the International Conference on Formal Engineering Methods*. Springer, 1–17.

Appendices

A Prompt Changes

The prompt used for Deepseek-R1 was a streamlined version of the one we designed for GPT-4. Since Deepseek-R1 incorporates reasoning capabilities by default, the explicit chain-of-thought instructions included in the GPT-4 prompt were removed. Additionally, we removed the "emotional stimulation" component from the GPT-4 prompt, as we found it made no difference.

Following the conference version of this article [19], we identified that loop invariants generated by GPT-4 were generally insufficient for verifying the generated contracts. As function verification and loop invariants were not the primary research objective, we modified the Deepseek-R1 prompt to explicitly prohibit annotations within the function body, such as loop invariants, and focus exclusively on generating function contracts. This restriction on in-body annotations allowed us to eliminate several examples from the GPT-4 prompt that addressed common mistakes. The following prompt is the previous prompt that we used when working with GPT-4 in the context of [19].

```
You are a LLM that takes the following inputs and returns a C program annotated with ACSL
      annotations.
Inputs:
1. A C program with no ACSL annotations
GOALS:
1. Describe any abstract properties that could be represented as ACSL annotations
2. Generate ACSL annotations based on your analysis of the program
3. Returning a program with no annotation is not a valid solution
4. Do not edit the C code, only add annotations
5. Make sure to describe your thought process behind the annotations
6. Do not skip any code in the returned solution to make it shorter.
7. If you break any of these rules then my family will disown me.

ANNOTATION EXAMPLES:


Examples 1 (single annotation):
/*@ requires low >= 0 && high <= 9; */

Example 2 (block annotation style):
```

```
//Only use this style for function headers. Do not use blocks for multiple annoations in the
    function body
/*@
  @ requires low >= 0 && high <= 9;
  @ requires elem >= 0 && elem <= 9;
*/

Example 3 (loops):
/*@
  @ loop invariant low <= high;
  @ loop variant high - low;
*/
while(low <= high)

Example 4 (loop assigns) (loop assigns must be placed before loop variant):
/*@
  @ loop invariant i >= 0 && i <= 3;
  @ loop assigns fa;
  @ loop variant 3 - i;
*/
while(low <= high)

Example 5 (assigns must be in scope):
//This is VALID because x is a parameter that the function contract can see
{valid_assigns}

// this is NOT VALID because x is in the function body and can not be seen by the contract
{invalid_assigns}


FORMAT INSTRUCTIONS:

First describe your reasoning behind the added annotations

Return the annotated c code wrapped in markdown
```c
...
```

START OF INPUT:
{program}
```

B LLM Comparison

Both GPT-4, developed by OpenAI, and DeepSeek-R1, developed by DeepSeek, are general-purpose LLMs designed to perform across a wide variety of domains. However, conducting a general comparison of GPT-4 and DeepSeek-R1 is challenging due to their flexibility and the broad range of tasks they can address. Their performance can vary significantly depending on the specific use case, prompt design, and evaluation criteria, rendering broad generalizations less meaningful.

Numerous benchmarks exist to evaluate LLMs on tasks such as logic (e.g., HellaSwag) [70], mathematics (e.g., MathBench) [40], and coding (e.g., Codeforces) [12]. However, these standardized benchmarks are not directly applicable to the specific objectives of this article, as our work focuses on domain specific qualitative outcomes rather than general-purpose metrics.

Therefore, in this study, we emphasize differences observed through our qualitative analysis, leaving a more general analysis to the variety of available benchmarks.

B.1 Annotation Counts

The differences in ACSL annotations generated by GPT-4 and Deepseek-R1 cannot be solely attributed to the change in language models, as the prompt used for generation was also refined. For a detailed explanation of the prompt changes, refer to the prompt section in Appendix A.

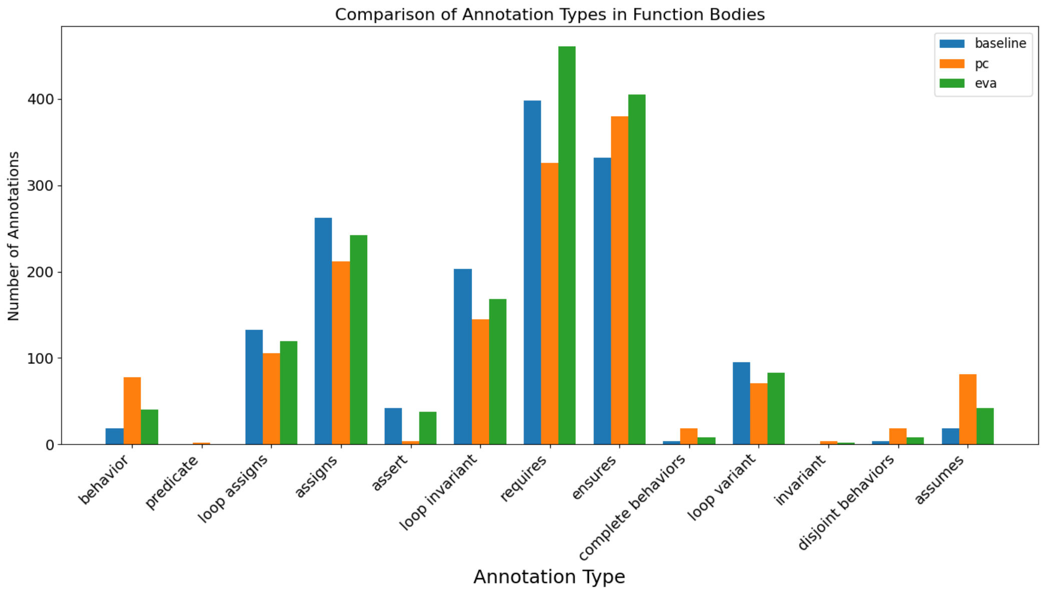

Fig. 24. Annotation-type counts from the conference paper [19].

B.1.1 Types. The most significant difference in annotation counts between the two models is the simplification of annotation types. GPT-4 produced 13 distinct types of annotations, whereas Deepseek-R1 reduced this to three core types: requires, ensures, and assigns. This reduction is partly due to the refined prompt, which explicitly forbids annotations within the function body, limiting the scope to function contracts. However, some differences, such as the absence of behavior clauses and predicates in Deepseek-R1's output, are attributable to the LLM itself.

B.1.2 Counts. The trends in the number of annotation types per prompt (baseline, Pathcrawler, EVA) remained largely consistent between GPT-4 and Deepseek-R1. Baseline prompts generally produced more annotations compared to prompts incorporating EVA or Pathcrawler, with the notable exception of preconditions in EVA-augmented prompts, which tended to be higher.

(1) **Reduction in Assigns Clauses with Deepseek-R1**: Deepseek-R1 generated significantly fewer assigns clauses compared to GPT-4. The latter often included uninformative "low-hanging fruit" assigns clauses, such as **assigns nothing** or erroneous clauses.
(2) **Shift in Focus with EVA-Augmented Prompts in Deepseek-R1**: When an EVA report was included in the prompt, Deepseek-R1 produced notably fewer ensures clauses compared to GPT-4, prioritizing requires clauses instead.

B.2 Qualitative Analysis

The qualitative analysis of ACSL annotations generated by GPT-4 and Deepseek-R1 revealed several key differences:

(1) **Syntax Errors and Erroneous Annotations**: GPT-4 produced a higher number of syntax mistakes and erroneous annotations, such as **assigns** clauses applied to variables declared within the function body, which are invalid in ACSL function contracts.

(2) **Reasoning Process**: GPT-4 typically included minimal chain-of-thought reasoning, often limited to a sentence or two. In contrast, Deepseek-R1, designed to allocate more time to reasoning before generating annotations, produced extensive reasoning outputs, often spanning hundreds of lines.
(3) **Bug Detection and Handling**: Deepseek-R1 was able to identify bugs in code and clearly explain its plan to address them. GPT-4, however, usually overlooked bugs.
(4) **Engagement with Examples**: With its more extensive reasoning process, Deepseek-R1 spent more time analyzing provided examples, particularly when Pathcrawler inputs were included. It reasoned about code execution using these inputs, leading to more contextually relevant annotations compared to GPT-4.

Received 16 April 2025; revised 11 December 2025; accepted 13 December 2025