

Inferring multiple helper Dafny assertions with LLMs

ÁLVARO SILVA, INESC TEC, Faculty of Engineering, University of Porto, Portugal

ALEXANDRA MENDES, INESC TEC, Faculty of Engineering, University of Porto, Portugal

RUBEN MARTINS, Computer Science Department of Carnegie Mellon University, USA

The Dafny verifier provides strong correctness guarantees but often requires numerous manual helper assertions, creating a significant barrier to adoption. We investigate the use of Large Language Models (LLMs) to automatically infer missing helper assertions in Dafny programs, with a primary focus on cases involving multiple missing assertions. To support this study, we extend the DafnyBench benchmark with curated datasets where one, two, or all assertions are removed, and we introduce a taxonomy of assertion types to analyze inference difficulty. Our approach refines fault localization through a hybrid method that combines LLM predictions with error-message heuristics. We implement this approach in a new tool called DAISY (Dafny Assertion Inference SYstem). While our focus is on multiple missing assertions, we also evaluate DAISY on single-assertion cases. DAISY verifies 63.4% of programs with one missing assertion and 31.7% with multiple missing assertions. Notably, many programs can be verified with fewer assertions than originally present, highlighting that proofs often admit multiple valid repair strategies and that recovering every original assertion is unnecessary. These results demonstrate that automated assertion inference can substantially reduce proof engineering effort and represent a step toward more scalable and accessible formal verification.

CCS Concepts: • **Software and its engineering** → **Formal software verification**.

Additional Key Words and Phrases: Dafny, LLM, Proof Synthesis

ACM Reference Format:

Álvaro Silva, Alexandra Mendes, and Ruben Martins. . Inferring multiple helper Dafny assertions with LLMs. In *Proceedings of* . ACM, New York, NY, USA, 21 pages.

1 Introduction

Formal verification tools such as Dafny [19] provide strong correctness guarantees by construction, enabling developers to annotate programs with specifications. In practice, however, verification rarely succeeds without additional effort: most proofs require numerous helper assertions — assertions or lemma calls that guide the verifier without affecting program behavior. Identifying which assertions are missing and where to place them is both challenging and time-consuming. Reports from the industrial verification of cryptographic systems [11] show that *the majority of proof engineering effort is spent addressing failing proofs*, with missing assertions as a primary cause. This persistent bottleneck makes automated assertion inference a crucial step toward improving the scalability and adoption of formal methods.

Large Language Models (LLMs) have recently been applied to reduce this burden by automatically inferring missing annotations [25, 28]. Prior work shows that LLMs can generate assertions, loop invariants, program repairs, and even full specifications [23, 32, 37], substantially lowering the manual effort of proof engineering. Among these efforts, Laurel [25] introduced a two-stage pipeline of fault localization and assertion inference, combining error-message heuristics with retrieval-augmented prompting to infer single missing assertions in Dafny lemmas.

While Laurel demonstrated the promise of LLM-assisted assertion inference, it had several limitations: it primarily targeted lemmas, only supported *one missing assertion at a time*, and left

Authors' Contact Information: Álvaro Silva, alvaro.silva@fe.up.pt, INESC TEC, Faculty of Engineering, University of Porto, Porto, Portugal; Alexandra Mendes, alexandra@archimendes.com, INESC TEC, Faculty of Engineering, University of Porto, Porto, Portugal; Ruben Martins, rubenm@cs.cmu.edu, Computer Science Department of Carnegie Mellon University, Pittsburgh, USA.

room for improvement in localization accuracy. Many verification failures, however, occur in program code and require multiple assertions.

We address these challenges by extending assertion inference to handle programs with *multiple missing assertions* and by refining fault localization through a hybrid approach that combines LLM predictions with error-message heuristics. To enable systematic study, we also construct a new benchmark set of 506 programs by extending DafnyBench [21] with curated variants where one, two, or all assertions have been removed. This dataset supports controlled evaluation of both localization and inference strategies. In addition, we introduce a taxonomy of assertion types, which shows that inference difficulty varies substantially across categories, with test-like assertions being the easiest to infer and multi-line assertions the most challenging.

We implement our approach in a tool called DAISY (Dafny Assertion Inference SYstem). On 506 Dafny programs, our best configuration verifies 63.4% of failing programs when one assertion is missing, compared to 51.9% for an extended version of Laurel that supports code. When more than one assertion is missing, our best approach achieves 31.7% successful verification. Notably, even though Laurel is limited to generating a single assertion, it still verifies 22.6% of these programs, showing that many proofs can be repaired with fewer assertions than originally present.

Although our experiments center on Dafny, the methods are language-agnostic and naturally extend to other SMT-based verification tools that rely on assertions [1, 2, 12, 18, 33, 35].

Contributions: Our main contributions are:

- **Generalization to multiple assertions:** A pipeline and evaluation for handling programs with more than one missing assertion.
- **Hybrid assertion localization:** Demonstrating that combining LLM-based prediction and heuristics based on error messages outperforms either method used in isolation.
- **Robust assertion localization analysis:** A systematic evaluation framework for assessing fix positions, beyond simple line distance metrics.
- **Benchmark extension:** A curated benchmark set derived from DafnyBench with one, two, and all assertions removed, enabling systematic evaluation.
- **Assertion taxonomy:** A categorization of helper assertions and an analysis of their relative inference difficulty.
- **Reproducibility package:** The code required to reproduce the experiments is available on Github.

2 Motivating Example

We illustrate our pipeline with a concrete example. It centers around a method `FindRange(q, key)` which, given a sorted sequence `q`, returns the half-open interval `[left, right)` where all elements equal `key`. The helper predicate `Sorted(q)` is also defined.

```

1 predicate Sorted(q: seq<int>){
2     forall i,j :: 0 <= i <= j < |q| ==> q[i] <= q[j] }
3
4 method {:verify true} FindRange(q: seq<int>, key: int)
5     returns (left: nat, right: nat)
6     requires Sorted(q)
7     ensures left <= right <= |q|
8     ensures forall i :: 0 <= i < left ==> q[i] < key
9     ensures forall i :: left <= i < right ==> q[i] == key
10    ensures forall i :: right <= i < |q| ==> q[i] > key}

```

In the client method `Main()`, a call to `FindRange` correctly computes the range of the value 10. However, Dafny’s verifier cannot automatically deduce that the resulting indices are exactly 4 and

7. This failure to prove the assertion `assert i == 4 && j == 7` signifies a need for additional helper assertions to bridge the gap between the method's general postconditions and the specific expected outcome.

```

1 method Main(){
2     var q := [1,2,2,5,10,10,10,23];
3     assert Sorted(q);
4     assert 10 in q;
5     var i,j := FindRange(q, 10);
6     assert i == 4 && j == 7 by {
7         assert q[0] <= q[1] <= q[2] <= q[3] < 10;
8     }}

```

Our pipeline proceeds in two steps:

Fault Localization: Our model first identifies the locations where assertions are missing using prompts as described in Section 3.1. For this example, it outputs the positions [5, 6], indicating that new assertions should be inserted after line 5 and 6. This generates a template with placeholders:

```

1 /*<Assertion is Missing Here>*/
2 assert i == 4 && j == 7 by {
3     /*<Assertion is Missing Here>*/
4     assert q[0] <= q[1] <= q[2] <= q[3] < 10;}

```

Assertion Inference: We then prompt an LLM to generate candidate assertions for these placeholders as described in Section 3.2. The model produces a list of 10 candidate pairs. For this example, one successful candidate pair is:

- `assert 10 in q[i..j];` (for the first placeholder)
- `assert (j == |q| || q[j] > 10);` (for the second placeholder)

Inserting these assertions leads to successful verification of the program:

```

1 assert 10 in q[i..j]; /* Added */
2 assert i == 4 && j == 7 by {
3     assert (j == |q| || q[j] > 10); /* Added */
4     assert q[0] <= q[1] <= q[2] <= q[3] < 10;}

```

Commenting out either added assertion causes verification to fail again, confirming that both are necessary. This indicates that verification failures often admit multiple valid repairs that differ in both the syntactic form and the location of the required assertions. Our pipeline successfully synthesized a solution that is distinct from the original ground-truth assertions, which were:

```

1 assert i == 4 && j == 7 by {
2     assert q[0] <= q[1] <= q[2] <= q[3] < 10;
3     assert q[4] == q[5] == q[6] == 10; /* Ground Truth */
4     assert 10 < q[7]; /* Ground Truth */

```

Crucially, this demonstrates that our approach can handle the repair of programs with multiple missing assertions by identifying non-trivial, essential helper assertions. The generated assertions are functionally sufficient to complete the proof, proving that our method can find valid solutions that are different from the human-written code.

3 Methodology

As a starting point, we tried a straightforward one-shot prompting strategy: providing the complete failing program to the LLM and asking it to insert all missing assertions simultaneously. However,

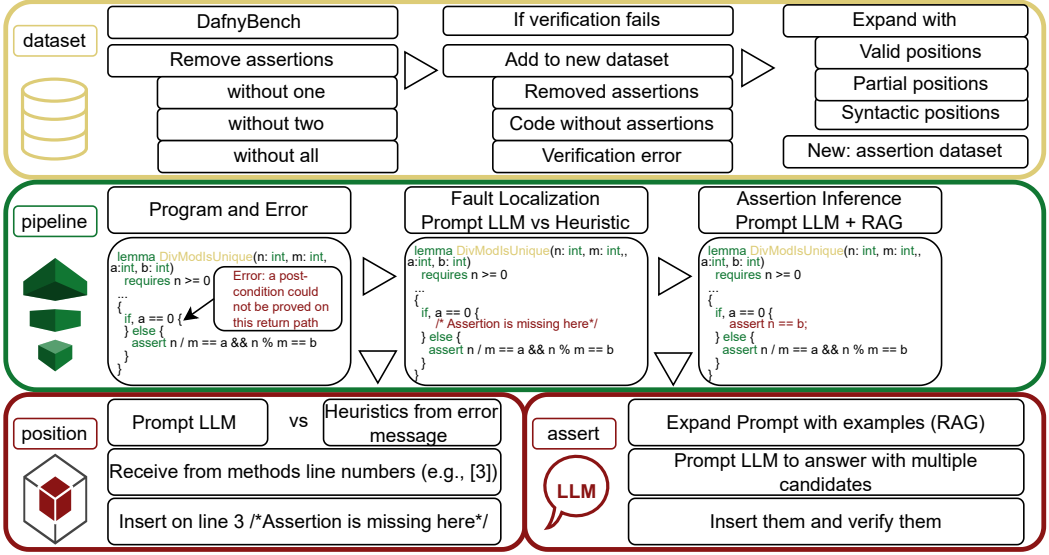


Fig. 1. Methodology Overview

this approach was unreliable, as the model sometimes altered unrelated code. To address this issue, and inspired by Laurel [25], we separated the task into two tasks: fault localization and inference.

This separation offers several advantages. It allows us to evaluate each stage explicitly and enables more efficient prompting. For example, we can request multiple assertion positions or contents in a single prompt by treating localization and inference as separate subtasks. Additionally, this method reduces costs, as each candidate requires fewer LLM calls. Figure 1 illustrates an overview of this methodology, with two main stages: *Fault Localization* and *Assertion Inference*.

3.1 Fault Localization

For fault localization, we used two main approaches: one *LLM-based* and other *heuristic-based*.

LLM-based Fault Localization: In this method, we design a prompt that presents the failing method with each line prefixed by its line number. This explicit numbering enables the model to directly reference and identify specific line candidates where it can insert assertions. Below, we show the preamble used in our LLM localization prompt.

```
You are given a Dafny method with line numbers.
Your task: return 1 or 2 line numbers after which a missing helper assertion should be
inserted to fix the program as json.
Format:
- [3] : one assertion after line 3
- [3, 4] : assertions after lines 3 and 4
Constraints:
- Only insert assertions inside the method body, i.e., between the opening { and closing }.
- Do not insert assertions in:
  - function/predicate/method signatures
  - preconditions (requires)
  - postconditions (ensures)
  - loop invariants
- Your answer must be in JSON list format: e.g., [3] or [3, 4].
Example:
0: method name(args)
```

Table 1. Error types and insertion strategies. **B**: before line l (insert at $l - 1$). **E**: end of enclosing block. We extend Laurel to support giving hints to errors of type LoopInvariants, TimeOut, SubsetConstraints and ElementNotInDomain.

Error type	Origin	Loc.
Assertion, Related, LHSValue, Calc, Constructed	Laurel	B
Postcondition, AssertBy, Forall	Laurel	E
LoopInvariants	DAISY	E
TimeOut, SubsetConstraints, ElementNotInDomain	DAISY	B

```

1:  specification
2:  {
3:    ...
4:  }
- All valid outputs: [2], [3], [2,3] (4 is outside the method)
-> If answer back [2] the new method would be
0: method name(args)
1:  specification
2:  {
3:    //Assertion Missing Here (added line)
4:    ...
5:  }
Now, decide the best line(s) do not add any commentary, give only but only the required
  answer in JSON format:
You must send at least one number in the answer!
Only give at most two lines in the answer (one or two options are the only admissible
  candidates)!
```

The prompt concludes with the target method (including line numbers) followed by the corresponding verifier error message (we called this method **Llm_{fl}**). We also experiment with an extended version that augments the prompt with retrieved examples to improve localization. For retrieving examples, we employ the best-performing strategy identified in Section 5.4, as described in Section 3.2 (we call this method **LlmEx_{fl}**). As shown in the prompt, our LLM-based fault localization can return *multiple positions*, enabling the generation of multiple assertions.

Heuristic-based Fault Localization: This approach determines insertion points for auxiliary assertions by applying heuristics derived from the verifier’s diagnostic messages. It systematically interprets the error types to suggest locations for new assertions. While our strategy builds on the techniques introduced in Laurel [25], we extend it to handle a wider range of verifier errors beyond those originally supported by Laurel. Table 1 summarizes the different error types and the corresponding insertion strategies. We refer to the base Laurel strategy as **Laurel_{fl}** and the extended version as **Laurel_{fl+}**. Note that both strategies rely on the verifier’s diagnostic messages and can return only *one position* for generating an auxiliary assertion.

In summary, for **Fault Localization Strategies** the following variants are explored:

- **Laurel_{fl}**: Laurel-like fault localization approach.
- **Laurel_{fl+}**: Improved version of Laurel-like fault localization.
- **Llm_{fl}**: LLM-based localization without augmenting the prompt with examples.
- **LlmEx_{fl}**: LLM-based localization with examples included in the prompt.
- **LlmEx_{fl}//Laurel_{fl+}**: A hybrid approach that leverages the localization results of both Laurel_{fl+} and LlmEx_{fl} in parallel.

3.2 Assertion Inference

The assertion inference strategy is entirely LLM-driven and, unlike Laurel, is generalized to handle an arbitrary number of missing assertions. The approach uses the following prompt:

```
The Dafny code below fails verification due to missing helper assertions.
Locations needing assertions are marked. For each location, return a JSON array of exactly 10
valid Dafny assertions that could fix the error at that point.
Output: a list of JSON arrays, one per location. No explanations or markdown. Escape double
quotes as \".
Examples:
If two positions:
[
  ["assert A;", "assert B;", "assert str2 != \"\"";", ...],
  ["assert C;", "assert D;", ...]
]
If one position:
[
  ["assert C;", "assert D;", ...]
]
Now generate the output do not add any commentary, give only but only the required answer in
JSON format:
```

Following this preamble, we include the whole method under analysis and additional context. The lines identified during the localization step are explicitly marked with `/*Assertion Missing Here*/` to indicate where to place the generated assertions. After invoking the LLM, a script processes the response by extracting and parsing the returned JSON.

Multi-embedding for Example Selection: To construct prompts, we retrieve relevant examples using verifier errors and program code. Each dataset entry stores precomputed embeddings of filtered error messages and full method bodies, generated with `jina-embeddings-v2-base-code` [15]. For a new failing program, we embed its error and code, compute cosine similarities with all entries, and combine them as $\alpha \cdot \text{sim}_{\text{error}} + (1 - \alpha) \cdot \text{sim}_{\text{code}}$. We then retrieve the top $k = 3$ matches, excluding examples from the same functions, methods or lemmas, to prevent data leakage. Each retrieved example provides the failing method, verifier output, and the assertions that restore verification. This approach is inspired by hybrid retrieval systems [6, 10], which integrate multiple semantic embeddings and may also incorporate keyword-based or other retrieval techniques to improve accuracy. Such systems enable more context-aware matching.

Inference Strategies: We experimented with several example retrieval methods:

- **NoEx_{in}**: Prompt without examples.
- **Random_{in}**: Prompt with randomly selected examples.
- **MulEmb_{α}_{in}**: Prompts with the examples that achieve the highest matching score against the target, using the multi-embedding method that combines code and error-message embeddings (Section 3.2). The method is parameterized by a weight $\alpha \in \{0.25, 0.5, 0.75, 1\}$.
- **Embed_{in}**: Prompts with the examples that achieve the highest matching score against the target using only code embeddings from the `jina-embeddings-v2-base-code` model. This is equivalent to `MulEmb0in`.
- **Tfidf_{in}**: Prompt with the examples with the highest matching score compared with the target using TF-IDF [29] based retrieval method.

4 Assertion Inference Dataset

We built our dataset on top of DafnyBench [21], a benchmark suite of 782 Dafny programs that successfully compile and verify. DafnyBench is currently the most comprehensive benchmark set available for Dafny. We systematically remove assertions from this corpus to create modified

programs that fail verification, thereby producing instances where our method must recover the missing assertions. We apply this procedure to all *functions*, *methods*, and *lemmas*, generating three types of instances for each:

- w/o-1** (*without one assertion*): We removed each assertion individually, one at a time. If removal caused verification to fail, we added both the missing assertion and the corresponding program to the **w/o-1** dataset.
- w/o-2** (*without two assertions*): We removed two assertions simultaneously, considering all unique pairs. If a pair contained one assertion already covered in **w/o-1** and another that was not, we excluded it, as solving it would be equivalent in difficulty to the corresponding **w/o-1** case. In contrast, we retained pairs where both assertions were in **w/o-1**, as well as pairs where neither was included. The motivation is twofold: when both assertions appear in **w/o-1**, they are each necessary for verification; when neither does, it indicates that even with two assertions removed, the program may still be verified by generating only one.
- w/o-all** (*without all assertions*): We remove all assertions, retaining only cases with at least three missing assertions to ensure no overlap with **w/o-1** or **w/o-2**. If verification fails, we add the missing assertions and the program to the **w/o-all** dataset.

After applying this procedure, we obtained 241 **w/o-1**, 235 **w/o-2**, and 30 **w/o-all** instances.

The **w/o-all** dataset is small, with only 30 cases, which is expected for several reasons. First, a substantial fraction of assertions (approximately 25%) serve primarily as lightweight testing aids, so removing them eliminates the corresponding checks, causing the verification to pass simply because there is nothing left to verify. Second, the combinatorial imbalance reduces the number of examples: for a method with 20 assertions, there are 20 distinct single-assertion removals but only one all-assertion removal. Finally, requiring at least three removed assertions further restricts this category to ensure its distinction from the **w/o-1** and **w/o-2** datasets.

Assertion Taxonomy: Because not all assertions may be equally difficult to infer, we classified single missing assertions into four categories to better understand which types present greater challenges for automated inference:

- **INDEX:** Assertions about sequences, sets, or arrays index properties (e.g., $s = s[0] + s[1..]$).
- **TEST:** Assertions resembling unit tests, often in `Main`.
- **MULTI:** Assertions spanning multiple lines, often using `by`.
- **OTHER:** Other remaining cases.

Table 2 presents the distribution of assertion types in the **w/o-1** dataset. Although the OTHER category contains many assertions, a substantial number also fall into more specific types, particularly INDEX. We expect that multi-line assertions (MULTI) are harder to infer, whereas TEST assertions should be easier. In Section 5.5, we analyze verification success rates by type, providing evidence in support of these expectations.

5 Evaluation

We designed our evaluation to address four research questions:

Table 2. Distribution of 241 assertions in the **w/o-1** dataset by type.

Type	INDEX	TEST	MULTI	OTHER
Values	69	33	12	127
%	28.6%	13.7%	5.0%	52.7%

- RQ1. How effective is Dafny assertion inference using LLMs?** We show how multiple approaches that combine fault localization and assertion generation perform on benchmarks **w/o-1**, **w/o-2**, and **w/o-all** — showcasing that LLM-based and heuristic-based approaches are complementary for fault localization.
- RQ2. How well do the different fault localization methods perform?** We compare the fault localization position with the ground truth for the different approaches.
- RQ3. How well do different methods for extracting examples improve assertion inference?** We compare different methods to extract examples based on retrieval-augmented generation and TF-IDF using verifier errors and program code.
- RQ4. Which types of assertions are harder to infer?** We split assertions into four categories and analyzed the success rate for each category.

5.1 Experimental Setup

We use the 506 Dafny programs (241 **w/o-1**, 235 **w/o-2**, and 30 **w/o-all**) for assertion inference generated as described in Section 4. These programs fail to verify unless additional assertions are introduced. To ensure reproducibility, we fixed the toolchain to use specific versions of Dafny [19] (fork of v4.10) and Z3 [8] (v4.15.2) and set a verification time limit of 300 seconds, as in Laurel.

Implementation and Cost: Each approach combines a fault localization method with an example retrieval strategy (e.g., **Llm_{fl}-Embed_{in}** uses LLM-based fault localization and embedding-based retrieval). All experiments were conducted with GPT-4.1, chosen both for its use in prior work on Dafny assertion inference [25] and for its cost-effective performance. At the time, OpenAI’s pricing was \$2 per 1M input tokens and \$8 per 1M output tokens. Our prompting strategy generates 10 candidate assertions per prompt, leading to an estimated cost of about \$70 for fully replicating all experiments (around 20M input tokens and 4M output tokens).

Running our best configuration, **LlmEx_{fl}-MulEmb0.5_{in}**, on the full benchmark (506 examples × 10 assertions) requires 3.6M input tokens and 0.30M output tokens, costing approximately \$8.84. This equals 1.33¢ per example or 0.133¢ per assertion candidate. Although this method uses more prompts than simpler alternatives (as it uses one more for localization), the overall cost remains low because of the small output size.

Fault Localization Evaluation: We classify predicted insertion positions into three categories based on whether inserting the ground truth assertions there succeeds or fails the verification: (i) **invalid**: insertion causes a syntax error; (ii) **partial**: insertion is syntactically valid, but verification fails; (iii) **valid**: insertion is syntactically valid and verification succeeds. To support this evaluation, we precompute validity during dataset generation by checking all single-assertion cases.

This evaluation goes beyond measuring line distance from the ground truth position. Success depends on the logical context of the insertion point, not merely its proximity. For instance, our dataset contains cases where the same assertion can be validly placed in many different locations, sometimes dozens, each restoring verification. To analyze this scenario, we considered the 241 **w/o-1** programs and attempted to reinsert the removed assertions at different program points. We then measured how many valid positions this produced, that is, how many distinct ways a program could be verified by placing the same assertion in different locations. Figure 2 shows assertions with more than 25 valid insertion points, while Figure 3 demonstrates that valid positions can be spread over 40 or more lines. These findings underscore that line-distance to the ground truth is an unreliable metric, whereas our validity-based evaluation offers a more accurate measure of success.

Finally, although this metric provides a solid evaluation framework, it is not perfect. In some cases, adding a different assertion or even multiple assertions at locations where the ground truth assertion would fail can still result in successful verification, a case we analyze in Section 5.3.

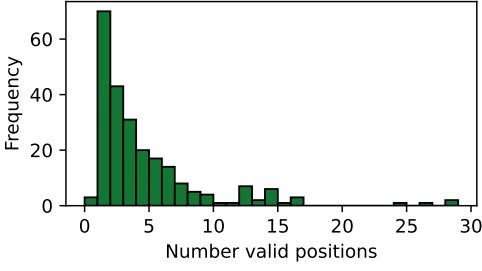


Fig. 2. Distribution of the number of valid insertion points for the ground truth assertion.

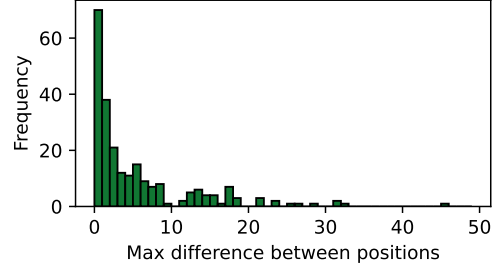


Fig. 3. Maximum line distance between valid insertion points for the same ground truth assertion.

Table 3. Verification success rate of each approach across benchmark categories, using the best retrieval strategy MulEmb0.5_{in}.

Approach	Benchmarks			
	w/o-1(241)	w/o-2(235)	w/o-all(30)	Combined(506)
LlmEx _{fl}	124(51.5%)	70(29.8%)	7(23.3%)	201(39.7%)
Laurel _{fl+}	125(51.9%)	54(23.0%)	6(20.0%)	185(36.6%)
LlmEx _{fl} //Laurel _{fl+}	153(63.4%)	76(32.3%)	8(26.7%)	237(46.8%)
GrTru _{fl}	160(66.4%)	85(36.2%)	10(33.3%)	255(50.4%)

5.2 RQ1: How effective is Dafny assertion inference using LLMs?

To demonstrate the effectiveness of the best approaches for inferring assertions, we combined the best-performing fault localization method with the top-performing example retrieval strategy (**MulEmb0.50_{in}**) and measured their overall performance. We also examined the effect of combining the two best position methods. Specifically, we evaluated the following approaches (**LlmEx_{fl}**, **Laurel_{fl+}**, **LlmEx_{fl}//Laurel_{fl+}** described in section 3.1). For comparison, we include **GrTru_{fl}**, a method that uses ground-truth assertion positions to approximate an empirical upper bound on fault localization performance.

Table 3 summarizes the verification success of the best approaches. Overall, **LlmEx_{fl}** achieved the highest accuracy, correctly predicting 39.7% of assertions, compared with 36.6% for the **Laurel_{fl+}** method. Note that for the **w/o-1** dataset, the performance of both approaches is similar with 51.5% and 51.9%, respectively. The most significant difference arose in the **w/o-2** dataset (29.8% vs. 23.0%). In the **w/o-all** dataset, **LlmEx_{fl}** also performed better. This outcome is expected: only the LLM-based fault localization can identify multiple positions, enabling the inference of multiple assertions and explaining its advantage on the **w/o-2** dataset. Notably, although **Laurel_{fl+}** can generate only *one* assertion, this is sufficient to verify 54 programs in the **w/o-2** dataset and 7 in the **w/o-all** dataset. This indicates that there are multiple ways to verify the programs and that successful verification can often be achieved with fewer assertions than in the original program.

Multiple Missing Assertions: As expected, the overall performance on **w/o-2** is lower than on **w/o-1**. We attribute this difference to several factors:

- (1) Most entries in **w/o-2** correspond to cases where the ground truth contained two assertions from **w/o-1**. In these programs, verification required inferring two assertions. Consequently, methods capable of generating more than one assertion had a clear advantage.

Table 4. Success rates for **w/o-2**, subdivided by assertion pair type using retrieval strategy MulEmb0.5_{in}.

Approach	Benchmarks w/o-2	
	both w/o-1 (172 cases)	none w/o-1 (63 cases)
LlmEx _{fl}	17 (9.9%)	53 (84.1%)
Laurel _{fl+}	8 (4.7%)	46 (73.0%)
LlmEx _{fl} //Laurel _{fl+}	20 (11.7%)	56 (88.9%)
GrTru _{fl}	33 (19.1%)	52 (82.5%)

- (2) We further separate the success analysis of **w/o-2** into two categories: (i) cases where both assertions were of type **w/o-1** (172 cases), and (ii) cases where none of the assertions were of type **w/o-1** (63 cases). As shown in Table 4, for the first category (*both w/o-1*) the success rates for **LlmEx_{fl}** and **Laurel_{fl+}**, were 9.9% and 4.7%, respectively. For the second category (*none w/o-1*), the success rates were much higher for **LlmEx_{fl}** and **Laurel_{fl+}**, with 84.1% and 73% respectively. This highlights that **LlmEx_{fl}** consistently outperformed **Laurel_{fl+}**, achieving more than twice the success rate in the more challenging category. Although, **LlmEx_{fl}** inferred only one assertion for 86 of the 172 benchmarks it still managed to fix 8 of them. For the rest of the examples were it correctly predicted that two assertions were missing it fixed 9.

Both approaches performed significantly worse on the **w/o-all** dataset than on the **w/o-1** dataset. The **w/o-all** dataset contains programs with 3 to 10 assertions removed, averaging 5 removals. **Laurel_{fl+}** generates only a single assertion, which was sufficient to verify 6 programs. In contrast, **LlmEx_{fl}** can generate multiple assertions, but we used the same prompt as in **w/o-2** and **w/o-1**, limiting output to at most 2 assertions. Under this setting, 7 programs were verified: 4 with one assertion and 3 with two. We also tested prompts without this restriction, but results were similar, with a maximum of 4 generated assertions. Since we observed that verification often requires only adding a few additional assertions and the number of missing assertions is unknown a priori, we use by default the version that caps generation at 2 assertions.

The main reason for the poor performance in the **w/o-all** dataset is that, unlike the other settings, no assertions remained in the program to provide local contextual clues for the model. Consequently, the model had to rely almost entirely on examples retrieved from similar programs, which limited its ability to generate relevant candidates. These instances are also inherently more complex, as they require inferring multiple assertions across different positions. Given the absence of in-program guidance and the added complexity, weaker results on **w/o-all** dataset were expected.

Combining Multiple Fault Localization Approaches: A gap remains between **LlmEx_{fl}** (39.7%) and **Laurel_{fl+}** (36.6%) compared to **GrTru_{fl}** (50.4%), highlighting room for improving fault localization. Our analysis shows that **LlmEx_{fl}** and **Laurel_{fl+}** are often orthogonal, and running them in parallel would substantially improve performance, bringing it much closer to **GrTru_{fl}**. Table 5 shows that a significant portion of cases exists where **LlmEx_{fl}** and **Laurel_{fl+}** do not overlap. In particular, in fault localization, **Laurel_{fl+}** found a valid position in 40 cases where **LlmEx_{fl}** did not, while the reverse occurred in 33 cases (a position is considered valid if the ground-truth assertion would verify there). When combined, overall performance achieves 63.4% for **w/o-1** and 31.7% for programs with more than 1 missing assertion. Overall, the performance is increased from 39.7% for the best individual model to 46.8% for the ensemble, representing a substantial improvement compared to either model alone. When **Laurel_{fl+}** fails, **LlmEx_{fl}** often succeeds, and vice versa. However, notice that this effect is mostly seen in the **w/o-1** dataset (for the **w/o-2** and **w/o-all** datasets, the improvements are more modest). Consider the following example:

Table 5. Performance and Position Overlap between LlmEx_{fl} and Laurel_{fl+} Methods

Metric	Benchmark	Only Laurel_{fl+}	Only LlmEx_{fl}	Both	Union
Successful Verif.	Combined	36	52	149	237
Successful Verif.	w/o-1	29	28	96	153
Valid Position	w/o-1	40	33	136	209

```

1 method testRawsort() {
2   var a : array<T> := new T[] [3, 5, 1];
3   /*<Assertion is Missing Here>*/ LlmEx_fl
4   rawsort(a);
5   /*<Assertion is Missing Here>*/ Laurel_fl+
6   assert a[..] == [1, 3, 5];}

```

Here, the Laurel_{fl+} heuristic fails because it assumes the fixing assertion should be in the line before the failing assertion. In contrast, LlmEx_{fl} correctly places the assertion before the call to `rawsort(a)`, capturing subtler inference opportunities that traditional heuristics miss.

Considering the combined performance of 46.8% versus the GrTru_{fl} 50.4%, jointly using both methods effectively addresses the challenge of inferring accurate assertion positions, achieving results close to the GrTru_{fl} . This is particularly true for **w/o-1**, which had the greatest increase; for **w/o-2**, the increase was not so pronounced, highlighting that the greatest contribution comes from LlmEx_{fl} in these scenarios. Note that many assertions, when inserted at positions different from the ground truth, can also lead to successful verification. As a result, it is possible to outperform the perfect fault localization provided by the ground truth. This occurs for the *none* subset of **w/o-1** within the **w/o-2** dataset, as shown in Table 4.

RQ1: Our results show that LLM-based inference (LlmEx_{fl} -MulEmb0.5_{in}) achieves verification success of 51.5% when one assertion is missing, and 39.7% across all benchmarks, outperforming Laurel_{fl+} . Performance varies across categories. While LLMs provide strong results, accuracy remains limited for **w/o-all** and for the hardest cases of **w/o-2** benchmarks. Notably, combining LlmEx_{fl} with Laurel_{fl+} raises success to 63.4% when one assertion is missing and 46.8% overall, nearly matching the oracle and showing the methods are complementary, representing the best found configuration for DAISY.

5.3 RQ2: How well do the different fault localization methods perform?

Fault Localization: To assess how effectively each method infers assertion positions, we first study whether placing the removed assertion at the predicted position would lead to successful verification. To keep the analysis trackable, we restrict our evaluation to the **w/o-1** dataset, where exactly one assertion was removed. Figure 4 reports the results using the categorization introduced in Section 5.1. Specifically, predicted positions are classified as: *valid* (verification succeeds), *partial* (the position is valid but verification fails), *invalid* (the position is not valid and causes a syntax error), or *no pos* when the method does not return a prediction.

The original Laurel_{fl} system left approximately 20% of cases unsupported, primarily due to missing loop-invariant handling, an issue explicitly addressed in our improvements. Our enhanced Laurel_{fl+} method reduced unsupported cases to roughly 7%. For **w/o-1**, Laurel_{fl+} achieved the highest accuracy (73%), followed closely by LlmEx_{fl} (70%). Note that even GrTru_{fl} exhibits 3.8%

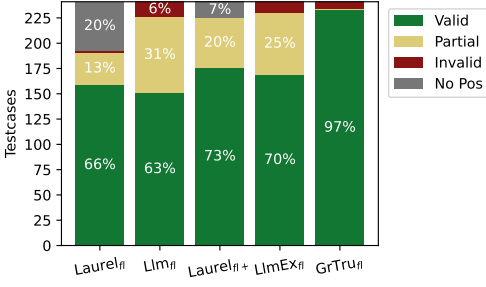


Fig. 4. Fix Position Categorization by LLM for w/o-1

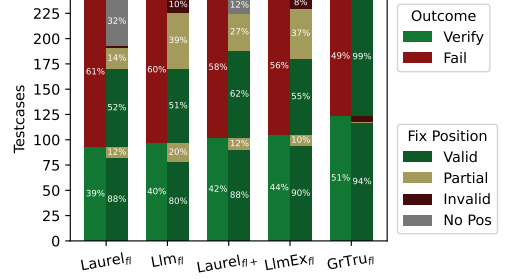


Fig. 5. Success Rate and localization distribution for w/o-1

Table 6. Verification success rate for each approach for each category of benchmarks for the position retrieval strategy without examples (NoEx_{in}).

Approach	Benchmarks			
	w/o-1(241)	w/o-2(235)	w/o-all(30)	Combined (506)
LlmEx _{fl}	105(43.6%)	68(28.9%)	4(13.3%)	177(35.0%)
Llm _{fl}	97(40.2%)	59(25.1%)	4(13.3%)	160(31.6%)
Laurel _{fl} +	102(42.3%)	56(23.8%)	5(16.7%)	163(32.2%)
Laurel _{fl}	93(38.6%)	57(24.4%)	2(6.7%)	152(30.0%)
GrTru _{fl}	124(51.5%)	79(33.6%)	5(16.7%)	208(41.1%)

invalid placements, because for position evaluation it mimics the pipeline by inserting assertions at the start of a line to test validity, although it uses the true location when generating fixes.

Success Rate: To reduce the influence of example retrieval in the overall performance, we test the success rate of fault localization leading to verified programs without using any strategy for example retrieval (NoEx_{in}). Table 6 shows that GPT-4.1 with LLM-based position inference achieves a 31.6% success rate, slightly outperforming Laurel’s default position inference at 30.0%. Our enhanced Laurel_{fl}++ approach performs even better, reaching 32.2%. The best overall performance comes from the LlmEx_{fl} strategy, with a 35.0% success rate. Results for inferring two assertions simultaneously follow the same trend observed in Section 5.2. For the **w/o-all** dataset, success rates are more modest, with Laurel_{fl}++ holding a small advantage. However, this difference is not significant, as also noted in Section 5.2, where the order was reversed due to a single program. Overall, when compared to the oracle GrTru_{fl} positions, a performance gap in position inference remains.

Success Rate vs. Fault Localization: Figure 5 show, for each method, the overall success/failure rate (left bar) and the distribution of inferred positions (right bar) for the **w/o-1** dataset. 88% of Laurel_{fl}++’s fixes fall in valid positions and 12% in partial ones; for Llm_{fl} these values are 80% and 20%, and for LlmEx_{fl} 90% and 10%. Conditioning on position reveals a clear pattern: Laurel_{fl}++ succeeds in 51% of valid cases but only 24% of partial ones; Llm_{fl} reaches 51% vs. 25%; and LlmEx_{fl} 56% vs. 18%. Across methods, a valid classification nearly doubles the likelihood of success relative to a partial one, indicating that position quality is a strong predictor of verification outcomes.

RQ2: Our results show that LLMs reliably infer candidate positions, performing on par with error-message-based heuristics for the **w/o-1** dataset and outperforming them on the **w/o-2** dataset. Extending the prompt with examples (LlmEx_{fl}) provides a slight additional advantage. Evaluating predicted positions also proves useful: valid positions are more than twice as likely as partial ones to lead to successful verification, making this a strong metric for assessing the quality of position inference algorithms.

5.4 RQ3: How well do different methods for extracting examples improve assertion inference?

To isolate the effect of fault localization when assessing the impact of example extraction, we evaluated several retrieval methods under perfect fault localization using the GrTru_{fl} position assertions. In addition, we propose a new hybrid retrieval strategy that combines code and error-message embeddings, using a cosine similarity metric with different values of α ($\text{GrTru}_{fl}\text{-MulEmb}\alpha_{in}$).

Table 7 compares different retrieval methods. The NoEx_{in} and Random_{in} baselines perform almost identically, indicating that irrelevant examples provide no benefit. This underscores the importance of retrieving contextually meaningful examples. All other methods outperform these baselines, confirming the clear value of retrieval-based inference.

The best-performing strategies were Embed_{in} , and our hybrid MulEmb0.50_{in} and MulEmb0.75_{in} . The overall results of MulEmb0.50_{in} and Embed_{in} (255 vs. 254), suggests that our hybrid approach is viable but not significantly better than the simpler embedding-only option. Since it is slightly better, we proceeded with our hybrid strategy (MulEmb0.50_{in}) for subsequent experiments. This choice reflects both the competitiveness of the method and our interest in exploring more flexible embedding combinations for future improvements.

We also observed that providing good examples substantially boosts performance for **w/o-1**: accuracy improves from 51.5% with NoEx_{in} to around 68% with the best retrieval method for that dataset, an increase of more than 15 percentage points. This confirms that relevant examples are critical for assertion inference. Interestingly, for **w/o-2** the gains were much more modest (around 3%). A plausible explanation is that the **w/o-2** dataset typically contains longer code snippets with at least two assertions (often many more), so the presence of other assertions within the prompt already provides considerable contextual information. Finally, in the **w/o-all** dataset, examples again play a major role: since no assertions are present in the code itself, examples become the only source of assertion-related context in the prompt.

As additional context, previous work on Laurel [25] compared its retrieval strategy against Tfidf_{in} and Embed_{in} , finding Laurel's method to be marginally better. However, the performance gap was not large, and implementing Laurel's retrieval strategy for our dataset would require substantial adaptation effort, similar to the significant modifications we had to make for their fault localization method. For this reason, and given that Tfidf_{in} and Embed_{in} already serve as strong and widely adopted baselines, we restricted our comparison to these methods, ensuring a fair and reproducible evaluation while keeping the focus on our proposed hybrid approach.

RQ3: Our results show that Retrieval-Augmented Generation (RAG) effectively retrieves contextually relevant examples, outperforming random or no-example baselines and improving results by more than 10%. The top-performing strategies were the embedding-based method and our hybrid approach ($\alpha = 0.5$). The identical results suggest the two methods are comparable in effectiveness, as the difference is not statistically significant.

Table 7. Verification success rate for each approach for each category of benchmarks for the example retrieval strategy on the ground truth positions (GrTru_{fl}).

Approach	Benchmarks			
	w/o-1(241)	w/o-2(235)	w/o-all(30)	Combined (506)
NoEx _{in}	124(51.5%)	79(33.6%)	5(16.7%)	208(41.1%)
Random _{in}	124(51.5%)	75(31.9%)	5(16.7%)	204(40.3%)
Tfidf _{in}	158(65.6%)	79(33.6%)	10(33.3%)	247(48.8%)
Embed _{in}	164(68.0%)	82(34.9%)	8(26.7%)	254(50.2%)
MulEmb0.25 _{in}	151(62.7%)	75(31.9%)	8(26.7%)	234(46.2%)
MulEmb0.50 _{in}	160(66.4%)	85(36.2%)	10(33.3%)	255(50.4%)
MulEmb0.75 _{in}	157(65.1%)	86(36.6%)	09(30.0%)	252(49.8%)
MulEmb1.00 _{in}	150(62.2%)	83(35.3%)	08(26.7%)	241(47.6%)

5.5 RQ4: Which types of assertions are harder to infer?

To evaluate which types of assertions are harder to infer, we use the classification presented in Section 4, where we categorize each assertion as being INDEX, MULTI, TEST, or OTHER for the **w/o-1** dataset where only one assertion is missing. Table 8 shows that TEST assertions are generally easier to infer, while MULTI assertions are the most challenging. Though the difference is modest, INDEX assertions are slightly easier than OTHER assertions, suggesting that targeted prompts could improve performance on INDEX types.

These findings highlight the relevance of separating TEST-like assertions from the rest, as they are not only easier to solve but also provide a clear pathway for enhancing AI-assisted verification pipelines. The high success rate for TEST assertions suggests a promising, two-stage **specification-first** workflow:

- (1) **Specification Drafting:** An LLM first generates candidate executable specifications (as TEST-like assertions) based on high-level requirements.
- (2) **Proof Repair:** The Dafny verifier is invoked, and its failure provides a precise target for a second LLM agent. This agent, leveraging methods from this work, then infers the necessary helper assertions to bridge the gap between the code and the specification.

This approach effectively uses the verifier as a ground-truth oracle to guide iterative proof synthesis. It reframes the problem from “infer an arbitrary missing assertion” to the more manageable task of “infer an assertion that satisfies this specific, automatically generated verification condition.”

In the era of AI and generative models, many works aim to infer complete code and specifications automatically [23, 32]. In these scenarios, automatically checking the quality of the solution is crucial. Our results provide empirical support for a pipeline where generating tests in Dafny is a reliable first step; when these tests cannot be verified, generating missing assertions of the TEST type can be extremely helpful. This connects our contributions directly to the broader trend of co-evolutionary synthesis, where code, specification, and proof are generated together.

RQ4: TEST assertions are generally easier to infer, while MULTI assertions remain the most challenging. INDEX assertions are marginally easier than OTHER assertions, suggesting that targeted prompts for INDEX cases could improve results. This hierarchy suggests TEST assertions could be a reliable first step in automated proof-synthesis pipelines.

Table 8. Success rates per assertion type for the **w/o-1** dataset using the best overall models, all using MulEmb0.5_{in} for example retrieval

Assertion Type	LlmEx _{fl}	GrTru _{fl}	Laurel _{fl+}
INDEX	38 / 69 (55%)	52 / 69 (75%)	41 / 69 (59%)
MULTI	3 / 12 (25%)	1 / 12 (8%)	2 / 12 (17%)
OTHER	60 / 127 (47%)	79 / 127 (62%)	61 / 127 (48%)
TEST	23 / 33 (70%)	28 / 33 (85%)	21 / 33 (64%)

6 Failure Analysis

Our best approach **LlmEx_{fl}**/**Laurel_{fl+}** is unable to solve 36.6% of the cases for the **w/o-1** dataset, and even if we had perfect fault localization, we would still have 33.6% unsolved cases for this dataset, indicating that some assertions are inherently challenging to infer. We manually analyzed all **w/o-1** assertions that none of the methods could solve (around 80 cases in total) and identified categories that suggest directions for improvement.

Complex multi-line assertions: There were 9 assertions that proved too difficult for our approach, mostly because they required multi-step assertions. The models rarely produced multi-step “by” assertions when a single assertion was missing. For this class of problems, approaches that attempt to generate complete auxiliary lemmas may prove more reliable.

Indexing and sequences: Many failures involved reasoning about sequence slicing or concatenation, which is common in Dafny proofs. Typical missing assertions include: “assert rest == rest[0..1] + rest[1..];”, “assert r == r[..k] + r[k..];”. Instead of producing these direct equalities, the model often attempted over-complicated multiset reasoning or type-incorrect variations. Improvements could come from incorporating explicit pattern templates for sequence equalities or augmenting the prompt with more examples of this type. We counted 13 cases, many of which appear solvable with better awareness of these common algebraic forms.

Insufficient context: Since we did not provide definitions of auxiliary functions, lemmas, or types in the prompt, some **GrTru_{fl}** assertions could not be inferred. In these 9 cases, the model could not guess the missing assertions due to missing background knowledge. Augmenting the prompt with more context could be an alternative.

Limited use of by-clauses: In some examples, the LLM correctly identified the required assertion but failed to provide a by-justification. For instance, in: assert c <= x by reveal A; the model inferred assert c <= x, but did not recognize the need to reveal the named clause A explicitly. This incapacity to reveal named assumptions accounted for 4 cases. A stronger integration of named assumptions and lemma references could likely solve such errors.

Confusion in Dafny proof idioms: We observed several heuristic gaps in how the model handled common Dafny proof flows:

- In 4 cases, the model attempted to assert the entire array equality (e.g., assert q == [1, 2, 2];) instead of asserting properties of each element individually (e.g., q[0]==1, q[1]==2, . . .), which is more useful in proof contexts.
- In 3 cases, the model failed to insert standard assertions converting between sequences, arrays, and multisets.
- In another 3 cases, the solution corresponded to an assertion already present elsewhere in the code, which the model did not attempt to reuse.

These issues could likely be mitigated with assertion templating, improved prompt design, or the reuse of existing assertions in the proof context. In some situations, the failures may be explained by the large size of the proof context, which may have caused the model to overlook simple repetitions.

Near misses: Finally, we observed several cases where the model’s output was nearly correct but not exact. For example, in the following lemma:

```

1 lemma perfectCube_Lemma (x:int)
2   ensures exists z :: (x*x*x == 3*z || x*x*x == 3*z + 1 || x*x*x == 3*z - 1);{
3   if x%
4   else if x%
5   else {
6     var k := (x-2)/3;
7     assert x*x*x == (3*k+2)*(3*k+2)*(3*k+2) == 27*k*k*k + 54*k*k + 36*k + 8;
8     /*<Assertion is Missing Here>*/}}

```

The correct assertion was: `assert x*x*x == 3*(9*k*k*k + 18*k*k + 12*k + 3) - 1;`

The model’s first attempt produced an assertion that was `assert x*x*x == 3*(9*k*k*k + 18*k*k + 12*k + 2) - 1;`, an almost correct expression, but with a small algebraic mistake. These “near misses” suggest that a lightweight algebraic simplifier or verifier-aided rewriting step could help resolve these near failures.

7 Threats to Validity

LLM Data Leakage: A primary threat to the validity of our results is the possibility of data leakage in the training of large language models (LLMs). Models such as GPT-4.1 may have been exposed to codebases or verification examples similar to those in our benchmarks during pretraining. This could artificially inflate performance by enabling memorization rather than genuine inference of assertion placements. However, our experiments show that improvements persist even when varying prompting strategies and testing new scenarios, suggesting that our observed gains reflect more than data leakage.

Benchmark Representativeness: Our study relies on a benchmark of verification tasks that may not fully capture the breadth of real-world programs or assertion requirements. Nevertheless, we employed the most extensive known dataset of Dafny code to date, offering a comprehensive evaluation within the scope of available formal verification examples.

Position Simplifications: To determine position validity, we inserted assertions consistently at the start of lines. While this approach facilitates automated checking, it can occasionally render programs unsolvable due to syntactic or semantic constraints. In practice, this limitation only significantly impacted about 4% of cases (manifesting as invalid positions in the ground truth position evaluation). Addressing this would require more sophisticated insertion mechanisms beyond our current setup.

Comparison of Multiple Assertions: When multiple assertion positions are inferred (e.g., position 1: A0, position 2: A1), we evaluate not only the complete pair (A0, A1) but also the partial combinations (A0, blank) and (blank, A1). This approach more accurately reflects realistic usage scenarios of our framework, which aims to maximize overall performance while also considering the validity of individual assertions. Testing only the complete pair (A0, A1) could result in the entire combination being marked invalid due to a single incorrect position. This design decision can influence comparisons between problems with different numbers of inferred assertions.

8 Related Work

Complete Program Synthesis: Recent work has begun integrating Large Language Models (LLMs) into formal verification workflows in Dafny, exploring complementary directions for combining specification, code generation, and proof. Clover [32] reframes correctness verification as a consistency-checking task across code, documentation, and formal annotations, leveraging GPT-4 to automatically generate code and specifications from natural language and then verify their mutual consistency. This triangulated approach significantly boosts confidence in correctness, achieving acceptance rates of up to 87% on verified instances. In parallel, Mirchev et al. [23] introduce a co-evolutionary methodology that jointly refines a tuple of artifacts (program, logical specification, and test cases) synthesized from the same prompt. Their program-proof co-evolution engine iteratively repairs and aligns these artifacts. Complementing these approaches, Misu [24] investigates the direct synthesis of verified Dafny programs, evaluating GPT-4 and PaLM-2 on MBPP problems with varied prompting strategies. GPT-4, particularly under retrieval-augmented chain-of-thought prompting, produces formally verified solutions for 58% of tasks, ultimately contributing 153 verified Dafny implementations. Also, Bradfronberner [3] introduces an algorithm that synthesizes verified Dafny code from around twenty challenging specifications by incrementally generating and checking code, which constrains program decoding and reduces the tokens required for verification [3].

In other languages, Yao et al. use GPT-4 with static analysis to generate invariants, conditions, and proofs for Rust (Verus), cutting proof code by 80% [38]. Wu et al.'s combines LLM-proposed invariants with solver validation, improving results on Java benchmarks and SV-COMP benchmarks [36]. In more traditional language, LLMs have also shown success in full program synthesis [5, 17].

Invariant Inference: Invariant generation is a key subproblem in program synthesis for deductive verification, as invariants are essential for reasoning about loops [13]. Compared to full program synthesis, it is simpler and more tractable, and research in this area has achieved significant progress. For example, Pascoal et al. [26] created a tool focused on inferring invariants for Dafny, which achieved success on the first attempt in 92% of cases, and within five attempts in 95%.

Several studies explore using LLMs and neural networks for invariant inference and ranking [16, 27]. For example, Chakraborty et al.'s iRank [4] combines GPT-generated candidates with a learned ranker to improve accuracy across loops in C programs. In contrast, classic methods rely on symbolic analysis [7], while recent work uses machine learning to synthesize invariants validated with SMT-based verifiers [14, 30].

Assertion Inference: The most comprehensive study to date on assertion inference for Dafny is Laurel [25], proposed by Mugnier et al., which combines static analysis with LLM-based generation. Laurel identifies potential fix sites by analyzing error messages from the Dafny verifier, applies heuristics to select promising locations, and retrieves similar examples to guide LLM prompting. This hybrid approach achieves a 52.4% success rate in inferring missing assertions. The gap in performance between invariant inference and assertion inference highlights the latter's higher complexity. Our work differ from Laurel in several ways: (1) it evaluates a pipeline for multi-assertion inference and explores methods that directly generate multiple assertion candidates; (2) it leverages LLMs not only for generation but also for predicting fix positions, achieving slightly better accuracy than Laurel's inferred positions; (3) it shows that LLM predictions for fix positions are orthogonal to Laurel's approach, and that combining the two yields better results; (4) it introduces a framework for evaluating the quality of position inference independently of assertion generation; and (5) it provides a fine-grained evaluation across different assertion types, offering insights into which categories of assertions are more tractable for LLM-based inference.

In other domains, retrieval-augmented methods have been explored for property generation. For instance, PropertyGPT [20] targets smart-contract verification: it builds a vector database of existing verified properties and, given new code, retrieves related examples to include in the LLM prompt. This RAG-inspired pipeline significantly improves the quality of generated function pre/postconditions and invariants. Non-LLM approaches like GoldMine [34], automatically generate hardware assertions by combining dynamic trace mining with formal checks.

Proof Repair: Silva et al. [31] investigated GPT-4’s ability to support lemma discovery and proof sketching in Dafny. Although generating syntactically valid code remains challenging, their study showed that even imperfect suggestions can meaningfully aid proof completion, particularly in lemma inference. Poesia et al. [28] explored the joint inference of loop invariants and helper assertions, achieving a 50.6% success rate with a fine-tuned model. Their approach, however, relied on exhaustively testing each candidate fix at all valid insertion points, which, while effective in benchmarks, becomes impractical for real-world developer tools due to the significant verification overhead. Loughridge et al. [21] took a different route, applying LLMs to directly infill missing loop invariants and assertions in the DafnyBench dataset, reaching a success rate of 68%. We initially experimented with prompt-based infilling, but, as explained, this approach proved unreliable. Instead, our methodology adopts a more cost-effective prompting strategy, generating multiple candidate solutions (up to ten) within a single query. The discrepancy in performance, even when using DafnyBench, can be explained by differences in dataset construction. Whereas Loughridge et al. included all cases, we restricted our evaluation to cases where removing an assertion produced a genuine verification failure, which excluded easier cases containing only loop invariants or trivial assertions that do not aid the verification, resulting in a more challenging benchmark.

In the broader theorem-proving context, Lu et al. [22] analyze GPT-3.5 on Coq proofs and find it often captures the high-level proof structure but fails on the details. They propose PALM, a “generate-then-repair” pipeline: the LLM generates an initial proof, then symbolic repair routines fix low-level errors. PALM dramatically outperforms prior tactics-based systems, proving 76–180% more Coq theorems than baselines. These efforts contrast with traditional ITP assistance [9] (tactics, SMT-based sledgehammer, etc.).

9 Conclusion

We presented a comprehensive study on LLM-based inference of helper assertions in Dafny, implemented in a tool called DAISY, addressing both the localization of missing assertion positions and the generation of their content. Our methodology enabled evaluation across programs of varying difficulty, including cases with multiple missing assertions. We also introduced a taxonomy of assertion types. The results highlight three key insights: (1) LLMs can effectively infer assertion positions, performing on par with or better than error message-based heuristics, and the two approaches are complementary, with their combination achieving performance close to that of a ground-truth oracle; (2) the proposed methods can handle multiple missing assertions, although performance degrades substantially when all assertions are removed, and inference difficulty varies widely across categories, with test-like assertions being relatively easy and multi-step assertions the most challenging; and (3) retrieval-augmented generation improves inference by incorporating relevant examples into prompts. While challenges remain, particularly for complex multi-line assertions, indexing expressions, and context-dependent proofs, our analysis points to promising directions for future work, including integrating assertion templates, enriching prompts with auxiliary definitions, and leveraging ensemble strategies. Overall, this work advances the state of the art in automated assistance for formal verification and demonstrates the potential of LLMs to meaningfully reduce developer effort in proof engineering.

10 Data Availability

The code required to reproduce the experiments is available on Github

Acknowledgment

This work was financed by *Fundação para a Ciência e a Tecnologia* (Portuguese Foundation for Science and Technology) within the project *VeriFixer*, with DOI 10.54499/2023.15557.PEX, and co-financed through the Carnegie Mellon Portugal Program under the fellowship reference PRT/BD/155045/2024.

References

- [1] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H Schmitt, and Mattias Ulbrich. 2016. *Deductive software verification-the key book*. Springer.
- [2] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J Summers. 2019. Leveraging Rust types for modular specification and verification. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30.
- [3] David Brandfonbrener, Simon Henniger, Sibi Raja, Tarun Prasad, Chloe Loughridge, Federico Cassano, Sabrina Ruixin Hu, Jianang Yang, William E. Byrd, Robert Zinkov, and Nada Amin. 2024. VerMCTS: Synthesizing Multi-Step Programs using a Verifier, a Large Language Model, and Tree Search. arXiv:2402.08147 [cs.SE] <https://arxiv.org/abs/2402.08147>
- [4] Saikat Chakraborty, Shuvendu K. Lahiri, Sarah Fakhoury, Akash Lal, Madanlal Musuvathi, Aseem Rastogi, Aditya Senthilnathan, Rahul Sharma, and Nikhil Swamy. 2023. Ranking LLM-Generated Loop Invariants for Program Verification. In *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6–10, 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, 9164–9175. <https://doi.org/10.18653/V1/2023.FINDINGS-EMNLP.614>
- [5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] <https://arxiv.org/abs/2107.03374>
- [6] Shiting Chen, Zijian Zhao, and Jinsong Chen. 2025. Each to Their Own: Exploring the Optimal Embedding in RAG. arXiv:2507.17442 [cs.CL] <https://arxiv.org/abs/2507.17442>
- [7] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, New York, NY, USA, 238–252.
- [8] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.
- [9] Martin Desharnais, Petar Vukmirović, Jasmin Blanchette, and Makarius Wenzel. 2022. Seventeen provers under the hammer. In *13th International Conference on Interactive Theorem Proving-ITP 2022 (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 237)*, June Andronick and Ugo de Liguoro (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 11:1–11:18. <https://doi.org/10.4230/LIPIcs.ITP.2022.11>
- [10] Nguyen Nam Doan, Aki Härmä, Remzi Celebi, and Valeria Gottardo. 2024. A Hybrid Retrieval Approach for Advancing Retrieval-Augmented Generation Systems. In *Proceedings of the 7th International Conference on Natural Language and Speech Processing (ICNLSP 2024)*. 397–409.
- [11] Mike Dodds. 2022. Formally Verifying Industry Cryptography. *IEEE Security & Privacy* 20, 3 (2022), 65–70. <https://doi.org/10.1109/MSEC.2022.3153035>
- [12] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3—where programs meet provers. In *European symposium on programming*. Springer, 125–128.
- [13] Robert W Floyd. 1993. Assigning meanings to programs. In *Program Verification: Fundamental Issues in Computer Science*. Springer, 65–81.
- [14] Pranav Garg, Daniel Neider, Parthasarathy Madhusudan, and Dan Roth. 2016. Learning invariants using decision trees and implication counterexamples. *ACM Sigplan Notices* 51, 1 (2016), 499–512.
- [15] Michael Günther, Jackmin Ong, Isabelle Mohr, Alaeddine Abdesslem, Tanguy Abel, Mohammad Kalim Akram, Susana Guzman, Georgios Mastrapas, Saba Sturua, Bo Wang, Maximilian Werk, Nan Wang, and Han Xiao. 2024.

- Jina Embeddings 2: 8192-Token General-Purpose Text Embeddings for Long Documents. arXiv:2310.19923 [cs.CL] <https://arxiv.org/abs/2310.19923>
- [16] 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 [cs.PL] <https://arxiv.org/abs/2311.07948>
 - [17] Anirudh Khatri, Joyce Cahoon, Jordan Henkel, Shaleen Deep, Venkatesh Emani, Avriella Floratou, Sumit Gulwani, Vu Le, Mohammad Raza, Sherry Shi, Mukul Singh, and Ashish Tiwari. 2023. From Words to Code: Harnessing Data for Program Synthesis from Natural Language. arXiv:2305.01598 [cs.DB] <https://arxiv.org/abs/2305.01598>
 - [18] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying rust programs using linear ghost types. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 286–315.
 - [19] K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*, Vol. 6355. Springer, Springer, Berlin, Heidelberg, 348–370.
 - [20] Ye Liu, Yue Xue, Daoyuan Wu, Yuqiang Sun, Yi Li, Miaolei Shi, and Yang Liu. 2025. PropertyGPT: LLM-driven Formal Verification of Smart Contracts through Retrieval-Augmented Property Generation. In *Proceedings 2025 Network and Distributed System Security Symposium (NDSS 2025)*. Internet Society, San Diego, CA, USA, 1–15. <https://doi.org/10.14722/ndss.2025.241357>
 - [21] 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 [cs.SE] <https://arxiv.org/abs/2406.08467>
 - [22] Minghai Lu, Benjamin Delaware, and Tianyi Zhang. 2024. Proof automation with large language models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. ACM, Luxembourg City, Luxembourg, 1509–1520.
 - [23] Martin Mirchev, Andreea Costea, Abhishek Kr Singh, and Abhik Roychoudhury. 2024. Assured Automatic Programming via Large Language Models. arXiv:2410.18494 [cs.SE] <https://arxiv.org/abs/2410.18494>
 - [24] 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.
 - [25] Eric Mugnier, Emmanuel Anaya Gonzalez, Nadia Polikarpova, Ranjit Jhala, and Zhou Yuanyuan. 2025. Laurel: Unblocking Automated Verification with Large Language Models. *Proceedings of the ACM on Programming Languages* 9, OOPSLA1 (2025), 1519–1545.
 - [26] João Pascoal Faria, Emanuel Trigo, and Rui Abreu. 2025. Automatic Generation of Loop Invariants in Dafny with Large Language Models. In *International Conference on Fundamentals of Software Engineering*. Springer, Springer, Berlin, Heidelberg, 138–154.
 - [27] Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2023. Can large language models reason about program invariants?. In *International Conference on Machine Learning*. PMLR, 27496–27520.
 - [28] Gabriel Poesia, Chloe Loughridge, and Nada Amin. 2024. dafny-annotator: AI-Assisted Verification of Dafny Programs. arXiv:2411.15143 [cs.SE] <https://arxiv.org/abs/2411.15143>
 - [29] Juan Ramos et al. 2003. Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, Vol. 242. New Jersey, USA, 29–48.
 - [30] Rahul Sharma and Alex Aiken. 2016. From invariant checking to invariant inference using randomized search. *Formal Methods in System Design* 48, 3 (2016), 235–256.
 - [31] 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 (FormalISE)*. Association for Computing Machinery, New York, NY, USA, 138–142. <https://doi.org/10.1145/3644033>
 - [32] Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. 2024. Clover: Closed-Loop Verifiable Code Generation. In *International Symposium on AI Verification*. Springer, Springer, Berlin, Heidelberg, 134–155.
 - [33] Nikhil Swamy, Cătălin Hrițcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, et al. 2016. Dependent types and multi-monadic effects in F. In *Proceedings of the 43rd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 256–270.
 - [34] Shobha Vasudevan, David Sheridan, Sanjay Patel, David Tcheng, Bill Tuohy, and Daniel Johnson. 2010. Goldmine: Automatic assertion generation using data mining and static analysis. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. IEEE, IEEE, Dresden, Germany, 626–629.
 - [35] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. 2014. LiquidHaskell: experience with refinement types in the real world. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell (Gothenburg, Sweden) (Haskell ’14)*. Association for Computing Machinery, New York, NY, USA, 39–51. <https://doi.org/10.1145/2633357.2633366>

- [36] Haoze Wu, Clark Barrett, and Nina Narodytska. 2024. Lemur: Integrating Large Language Models in Automated Program Verification. *arXiv:2310.04870 [cs.FL]* <https://arxiv.org/abs/2310.04870>
- [37] Valentina Wu, Alexandra Mendes, and Alexandre Abreu. 2025. Specification-Guided Repair of Arithmetic Errors in Dafny Programs using LLMs. In *23rd International Conference on Software Engineering and Formal Methods (SEFM 2025)*.
- [38] Jianan Yao, Ziqiao Zhou, Weiteng Chen, and Weidong Cui. 2023. Leveraging Large Language Models for Automated Proof Synthesis in Rust. *arXiv:2311.03739 [cs.FL]* <https://arxiv.org/abs/2311.03739>