

000 001 002 003 004 005 006 007 008 009 010 011 012 013 014 015 016 017 018 019 020 021 022 023 024 025 026 027 028 029 030 031 032 033 034 035 036 037 038 039 040 041 042 043 044 045 046 047 048 049 050 051 052 053 VERIEQUIVBENCH: AN EQUIVALENCE SCORE FOR GROUND-TRUTH-FREE EVALUATION OF FORMALLY VERIFIABLE CODE

Anonymous authors

Paper under double-blind review

ABSTRACT

Formal verification is the next frontier for ensuring the correctness of code generated by Large Language Models (LLMs). While methods that co-generate code and formal specifications in formal languages, like Dafny, can, in principle, prove alignment with user intent, progress is bottlenecked by specification quality evaluation. Current benchmarks rely on matching against ground-truth specifications, a manual and expertise-intensive process that has limited existing datasets to a few hundred simple problems and also suffers from a reliability issue. To address this, we introduce VeriEquivBench, a new benchmark with 2,389 complex algorithmic problems that probe the limitations of current models in both code generation and formal reasoning. Our evaluation framework replaces ground-truth matching with a formally grounded metric, the equivalence score, and rigorously verifies the quality of generated specifications and code. Our results show that generating formally verifiable code remains a profound challenge for state-of-the-art LLMs. This underscores both the difficulty of the task and the need for benchmarks like VeriEquivBench to drive progress toward scalable and reliable coding agents.

1 INTRODUCTION

Large language models (LLMs) already possess substantial capacity for following natural-language instructions and executing a wide range of coding tasks (Li et al., 2022a; Jain et al., 2024; Zhao et al., 2025). At the same time, the correctness of the generated code remains a concern (Cotroneo et al., 2024; Wang et al., 2025b), where functional errors cost users extra effort to debug and also pose significant risks in the safety-critical domain (Dalrymple et al., 2024). A common solution is to evaluate generated code through unit tests (Jimenez et al., 2024; Wang et al., 2025c). However, this process offers no provable guarantee of correctness, as insufficient unit test coverage can fail to detect critical errors (Yu et al., 2025). On the contrary, a verifiable system resolves the issue by co-generating formal specifications and code to formally verify the alignment with the natural language query intention (Sun et al., 2024). Our work focuses on building an end-to-end agent for formal verification, for which we adopt Dafny (Leino, 2010). It is an ideal choice as Dafny’s automatic theorem prover (De Moura & Bjørner, 2008) eliminates the need for manual proof writing. Furthermore, its similarity to common languages like Python and C simplifies code transformation.

While several benchmarks (Ye et al., 2025b; Thakur et al., 2025) target at building a reliable reasoning system by formally ensuring the exact code generation (György et al., 2025), their progress is constrained by the reliance on manually-written, ground-truth specifications for evaluation. This formal annotation process is incredibly labour-intensive and requires deep expertise (Misu et al., 2024), which sets a barrier to scaling these benchmarks in both size and complexity. As a result, prominent Dafny benchmarks, including DafnySynthesis (Misu et al., 2024) and CloverBench (Sun et al., 2024), contain only 215 simple examples combined, insufficient for evaluating current LLMs’ advanced reasoning abilities. Moreover, the reliance on expert annotation is not only a scaling bottleneck; it also leads to a reliability issue. An analysis (Sun et al., 2024) has figured that 10% of expert-written specifications

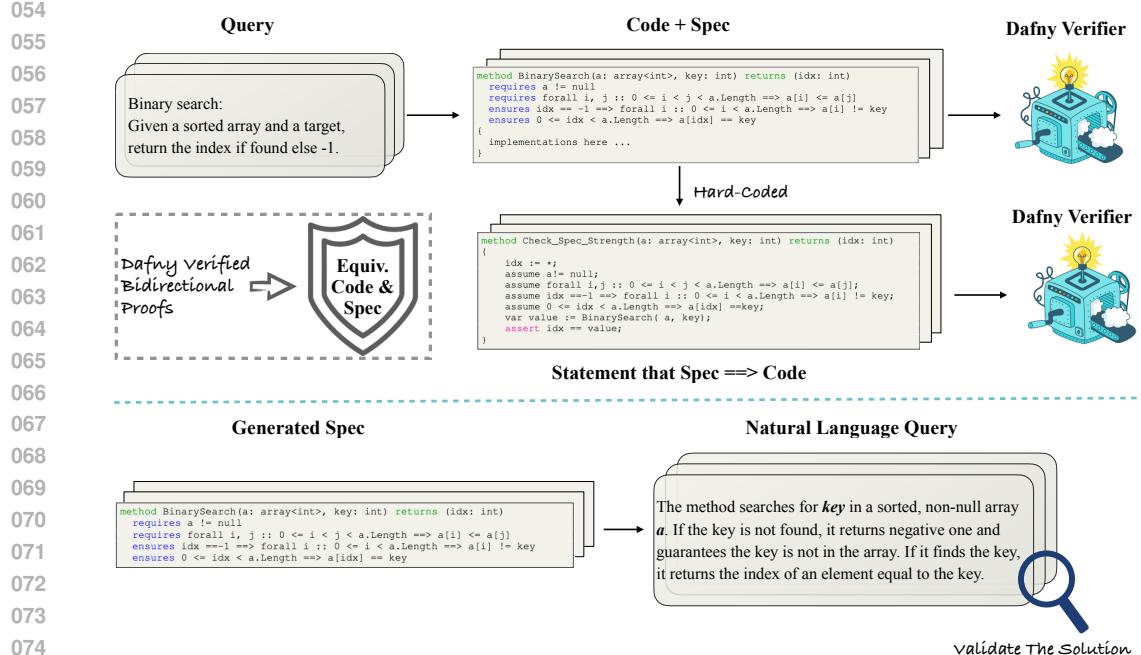


Figure 1: An end-to-end verifiable coding agent first generates code and specifications, using the Dafny verifier to prove their mutual equivalence. Here, the script to verify the implication from specification to code can be generated directly by our Python code. It then translates the complete formal specification back into natural language, allowing the user to confirm that it aligns with their original intent.

in DafnySynthesis are wrongly claimed as ground-truths, and our own review finds another 18%, containing errors or ambiguities. Such flaws undermine the validity of any benchmark that depends on a ground-truth solution. This raises a critical question: *How can we reliably evaluate specifications' quality **without** depending on the ground-truth?* To answer this question, we make the following concrete technical contributions:

Contribution 1. We propose a novel formally-grounded metric, named the *equivalence score*, that measures the mutual equivalence between generated code and its specifications. The score confirms whether a specification unambiguously describes the code's behaviour by using the Dafny verifier to check for bidirectional implication. This automated process has no false positives, ensuring that only correctly matched code-specification pairs are accepted. In order to validate the alignment with the query intention, we further include a second evaluation step: translating formal specifications back to natural language, as used by Ying et al. and Sun et al.. The complete pipeline is illustrated in Figure 1. Note that the effort in translation between natural and formal languages is asymmetric, since natural language can be ambiguous and lack necessary logic (Jiang et al.). Using Claude-4 as a judge (Wang et al., 2025a), we observe a high success rate of 82.98% for Grok-4 translations, confirming its viability as an evaluation metric.

Contribution 2. Equipped with our automated evaluation metric, we introduce *VeriEquiv-Bench*, a benchmark of 2,389 examples with natural language problem descriptions, code and specifications, and additionally, 1,678 synthetic algorithmic problems. VeriEquiv-Bench significantly expands on prior work in both dataset size and problem complexity, a leap demonstrated by the average Cyclomatic Complexity score, which rises from 2.44 in DafnySynthesis to 5.63. The core of our dataset is converted from the LeetCode corpus (Xia et al., 2025), a large and community-validated collection of algorithmic problems well-suited for evaluating a model's reasoning abilities. To supplement this data, we also introduce a synthesis pipeline that uses a structured tagging system to generate novel queries by randomly combining tags for different domains, data structures, and algorithms, introduced in Section 2.3. This provides a scalable method for creating large training datasets of new problem descriptions that are fully compatible with our automated evaluation signal.

108 However, Claude-4 is able to generate qualified Python code for only 15.85% of these novel
 109 synthesis queries, reflecting the model’s limitations.
 110

111 **Contribution 3.** We conduct a concrete evaluation of state-of-the-art LLMs, where Veri-
 112 iEquivBench serves as a testbed for these models to explore and extend the reasoning abilities
 113 on complex problems, beyond human-annotated data (Silver et al., 2021; Ye et al., 2025a;
 114 Shojaee et al., 2025). Our evaluation highlights the profound difficulty of this task and the
 115 effectiveness of our benchmark. The best-performing model, Claude-4-sonnet, which solves
 116 75.81% of the problems in CloverBench, succeeds on only 4.83% of our data, even with a
 117 pass@4 metric. Given this poor performance, and following prior work (Loughridge et al.,
 118 2025; Yan et al., 2025), we include two simpler auxiliary tasks to scaffold the problem: (1)
 119 infilling the necessary clauses to complete a given specification, and (2) generating a speci-
 120 fication from provided Dafny code. To facilitate future research, we establish baselines for
 121 both tasks using reinforcement learning.
 122

2 BENCHMARK OVERVIEW AND CONSTRUCTION PIPELINE

124 In this section, we first present aggregate data statistics for VeriEquivBench. Subsequently,
 125 we introduce the two curated subsets released with the benchmark: (i) the LeetCode-
 126 transformed dataset, and (ii) a tag-composition dataset, called TagComp, the latter being
 127 explicitly constructed to evaluate verifiable agents on novel data without contamination (Tu
 128 et al., 2024; Riddell et al., 2024).

129 Each problem in our benchmark provides a comprehensive set of artifacts: a natural language
 130 query, implementations in both Python and Dafny, unit tests and two versions of formal
 131 specifications: a strong auto-formalized baseline explained in Section 2.1 and a weaker,
 132 verifiable but incomplete version explained in Section 2.2. Additionally, each problem is
 133 annotated with metadata, including its difficulty level and descriptive tags for the relevant
 134 algorithm, data structure, and domain. Unlike LeetCode, our benchmark uses a more
 135 detailed and structured set of tags to categorize problems. This new tagging system is
 136 described in Section 2.3 for future query synthesis.

137 Starting from the original Leetcode split of 2,641 training and 228 test instances, we first
 138 curate 2,174 cases successfully transformed to Dafny. Then we compose new problems
 139 by merging tags, producing 1,893 additional items; the full tag-composition procedure is
 140 described in Section 2.3. For new problems, we ask Claude-4-sonnet to generate pairs of
 141 Python code and corresponding unit tests. For only 300 of new problems, Claude-generated
 142 code passes at least 85% of their corresponding unit tests, forming the cleaned *TagComp*
 143 dataset. Of these, 215 samples clear the weak-baseline pipeline, giving us 2,389 problems
 144 in total that pair natural-language queries with formally annotated code.

145 Table 1 presents key metrics for our annotated Dafny code, which uses the weaker, veri-
 146 fably correct specifications. Our problems are significantly more complex than those in
 147 CloverBench, often involving multiple methods rather than a single one. Furthermore,
 148 the corresponding specifications, while incomplete, contain a substantial number of formal
 149 clauses.

150 Table 1: The table overviews several attributes of our annotated code.
 151

Dataset	Metric	function	method	invariant	ensures	decreases
LeetCode	mean	0.78	1.33	5.12	1.71	0.46
TagComp	mean	0.96	3.18	7.34	3.14	0.70

2.1 LEETCODE AUTOFORMALIZATION

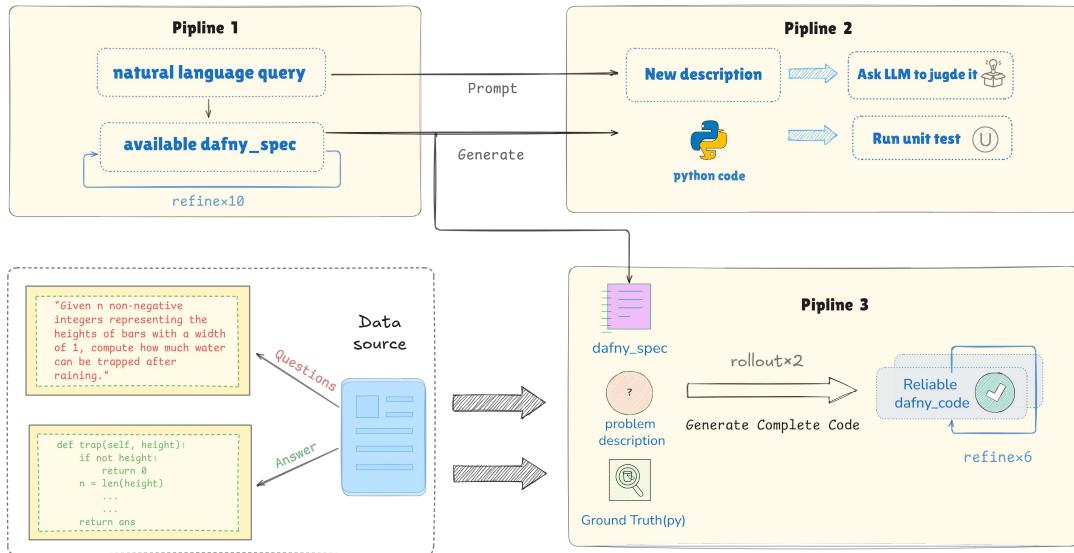
157 Past formal-language sets such as DafnyBench (Loughridge et al., 2025) are still small and
 158 narrow, because hand-written specifications are too costly to scale (Misu et al., 2024). To
 159 obtain large, varied training data without extra human cost, we mine the classic Leetcode
 160 pool, convert problems to formal specifications, stated in Figure 2 Pipeline 1, while keeping
 161

162 query and specification aligned with two short tightening evaluation protocols (Sun et al.,
 163 2024), shown in Figure 2 Pipeline 2.

165 **Specification Generation** We feed the problem description to Claude-4-sonnet to obtain
 166 an initial Dafny specification, yet even the initial drafts often contain syntax errors. Thus,
 167 we revise and resubmit up to ten times until the file has no parse or resolution errors. We
 168 find that supplying two simple examples exploits the model’s in-context learning (Dong
 169 et al., 2023) and sharply lowers the error rate (prompt template in Appendix C).

170 Furthermore, we constrain the model to generate specifications using only first-order logic,
 171 prohibiting recursive or dynamic programming-style definitions. This ensures the speci-
 172 fication describes the problem’s declarative properties without leaking the implementa-
 173 tion’s structure.

175 **Equivalence to NL** The equivalence check follows the protocol proposed by Clover (2024)
 176 and contains two steps: (1) A model (we use Grok4 here) rewrites the description so that
 177 it cleanly mirrors the specification, then another model (Claude-4) judges the equivalence
 178 between the original description and the rewritten one, yielding a score; (2) The specification
 179 alone is translated into Python and executed against the ground-truth LeetCode unit tests.
 180 The unit test passing rate is reported in Appendix G.1.



200 Figure 2: The figure outlines our autoformalization and code generation workflow: Pipeline
 201 1 produces comprehensive and syntax-free specifications; Pipeline 2 checks consistency
 202 between the NL query and the specifications; Pipeline 3 emits fully annotated code that passes
 203 the verifier.

204

205

206 2.2 LEETCODE VERIFIABLE CODE GENERATION

207

208 Owing to the limited performance of state-of-the-art LLMs on challenging formal-language
 209 coding, we adopt the multi-stage pipeline (pipeline 3 of Figure 2): prompted by the previ-
 210 ously generated specifications, the problem description, and a reference Python solution, the
 211 stronger model (Claude-4) produces annotated Dafny code, while a lighter model (Claude-
 212 3.5) then polishes this output, iterating up to six times to eliminate any syntax or parsing
 213 errors.

214

215

In practice, the vast majority of problems converge within three refinement rounds, while a residual subset still fails to yield a well-formed artifact even after the sixth attempt, with the success transformation rate reported in Table 2, and data statistics reported in Table 1.

216 Table 2: The table shows the number of examples without syntax errors in autoformalization
 217 and verifiable examples in code generation.

Dataset	Spec Autoformalization		Dafny Code Generation	
	Number	Rate (%)	Number	Rate (%)
LeetCode	2584	90.1	2174	75.8
TagComp	296	98.7	215	71.7

225 2.3 DATA SYNTHESIS THROUGH TAG COMPOSITION

227 We propose constructing a fine-grained, “template-level” taxonomy to provide an abstract
 228 description of algorithmic problems via tags (Wang et al., 2025c). In our system, every task
 229 is labelled with three orthogonal categories: **domain**, **data-structure**, and **algorithm**
 230 class((Chollet et al., 2025)).

231 To obtain these labels, we **(i)** harvest a high-quality seed pool from the Luogu online judge
 232 ([luo, 2025](#)), and **(ii)** manually prune hallucinated or off-topic tags. Our ontology defines
 233 over 500 fine-grained tags, offering more than seven times the descriptive granularity of the
 234 69 tags used by LeetCode (see Appendix A for a comparison). The tag set is designed
 235 so that, taken together, the tags collectively reflect the complete programming knowledge
 236 entailed by each individual problem, while retaining a modest level of abstraction.

237 The three categories of tags capture complementary aspects of programming knowledge.
 238 First, the domain category encompasses the overarching problem space or application con-
 239 text in which an algorithm operates, such as graph theory. Second, the data structure
 240 category pertains to the foundational mechanisms for manipulating data that underpin the
 241 algorithm’s functionality and efficiency, like arrays. Third, the algorithm category refers
 242 to the core strategic paradigm employed, such as sorting, focusing on the decision-making
 243 logic. These algorithm tags directly shape the overall control flow of a solution, as they
 244 orchestrate the program logic and structure.

245 However, not all problems conform to highly standardized patterns. In contemporary algo-
 246 rithmic competition problems, for instance, many challenges necessitate solvers to discern
 247 the underlying mathematical structures, an approach commonly termed “constructive meth-
 248 ods”. From a coding perspective, these constructive methods typically appear as compact
 249 code blocks that rely solely on fundamental loops or arithmetic operations. Consequently,
 250 it is difficult to categorize them beyond a general “constructive method” tag. From the
 251 problem setter’s viewpoint, such problems and their solutions stem from empirically ob-
 252 served mathematical structural properties, which inherently resist exhaustive coverage by
 253 conventional tags.

254 To synthesize novel queries, we select tags in the following workflow: first, we randomly pull
 255 12 tags from each of three pools, and then let Claude-4 pick any 3–8 tags in total. This short
 256 list is fed back to the model so that Claude can create one clear algorithm question with
 257 roughly 40 unit tests ([Xu et al., 2025](#)). Initially, we create approximately 1,900 problems,
 258 but only retain the 300 that pass at least 85% of their tests ([Xu et al., 2025](#)), and call this
 259 clean set, *TagComp*. The detailed pipeline and prompt templates used can be found in
 260 Appendix B and Appendix C.

262 3 EVALUATION METRICS AND TASKS

264 A verifiable coding agent reduces hallucinations and provides trustworthy solutions aligned
 265 with users’ intentions. As shown in Figure 1, our solution evaluation includes two steps,
 266 which are

- 268 • verifying the equivalence between generated code and specifications, and
- 269 • validating the solution by translating formal specifications back to problem descrip-
 270 tions in natural languages.

```

270
271
272     method Check_Max_Spec(a: int, b: int) returns (max: int)
273     {
274         max := *;
275         assume max >= a;
276         var value := Max(a, b);
277         assert value == max;
278     }
279

```

280 Figure 3: We show an example where the equivalence score proves the given specifications
 281 are underspecified for returning the maximum between two integers. The code presents the
 282 statement to verify whether the specification implies the code.
 283

284 To understand the need for verifying the **equivalence** between the code and the specifications,
 285 consider a simple binary search algorithm. The goal is to return the index of a *key*
 286 in a sorted array *a*, or negative one if the key is not found. A weak but verifiable post-
 287 condition might only state that the output, *idx*, is within a valid range: **ensures** $-1 \leq$
 288 *idx* < *a.Length*. While this specification passes the verifier, it fails to exactly describe the
 289 code. This creates a dangerous loophole: an incorrect implementation that doesn't actually
 290 find the key could still satisfy this weak condition, and the verifier would not catch the error.
 291

292 Existing benchmarks do not offer a metric to formally validate the quality of specifications.
 293 Without one, there is no way to guarantee that the verified code truly aligns with its
 294 intended behaviour. Instead, building up equivalence examines whether the specification is
 295 complete without ambiguities. Our equivalence score accomplishes the task by proving the
 296 bidirectional implication relationship:

- 297 • whether the code falls into the lattices described by the specifications, and
- 298 • whether specifications tightly describe the code behaviour for any inputs.

300 Both proofs can be automatically completed by the Dafny verifier. The first direction can
 301 be verified by passing the annotation to the verifier. The second direction requires creating
 302 a statement that the specification implies the code for the verifier to check.
 303

304 Figure 3 presents a counterexample to illustrate how our equivalence score identifies an
 305 underspecified function. The Max method correctly returns the maximum of two integers *a*
 306 and *b*, but its post-condition (**ensures** *max* >= *a*) is too weak; it doesn't guarantee that the
 307 output is also greater than or equal to *b*. To test if the specification fully implies the code's
 308 behaviour, we use the Check_Max_Spec method. This method creates an arbitrary value
 309 *max*, assumed to satisfy all provided pre-conditions and post-conditions. Our equivalence
 310 score then tests the assertion that variables described by the specifications are equal to the
 311 method outputs. The Dafny verifier is guaranteed to find this assertion to be false without
 312 any false positives. Because the specification is not strong enough to imply the code, this
 313 program would not receive an equivalence score.
 314

315 As mentioned in the introduction, end-to-end formally verifiable code generation is still
 316 challenging for current proprietary LLMs. Dafny has its own programming logic, such
 317 as claiming the invariance of old elements in arrays to support the proof. Therefore, we
 318 re-emphasize the importance of two auxiliary tasks to facilitate understanding of specific
 319 nuances of Dafny, introduced in DafnyBench (Loughridge et al., 2025) and Veri-Code Series
 320 I (2025):

- 321 • **Verifiable Code Refinement:** Given annotated but unverified Dafny code, the
 322 model's goal is to add the necessary intermediate clauses, such as invariants and
 323 lemmas, to make the code pass the verifier. Success is determined by successful
 324 verification.
- 325 • **Code-To-Spec Generation:** Given a Dafny implementation, the model attempts
 326 to generate the strongest formal specification. The quality of the output is evaluated
 327 by measuring its strength improvement over a baseline, using the spec-superior-score
 328 (Yan et al., 2025).

324 Our two sets of formal specifications map onto these auxiliary tasks. For the Verifiable Code
 325 Refinement task, models are challenged to fix our strong auto-formalized specifications. For
 326 the Code-to-Specification Generation task, models improve upon our weaker, but already
 327 verified, specifications.

329 4 EMPIRICAL EVALUATION

330 This section validates the quality of our benchmark and the reliability of our evaluation
 331 metric. We then present the performance of several state-of-the-art LLMs on the end-to-
 332 end verifiable code generation task, followed by an analysis of our baselines on the two
 333 auxiliary tasks.

334 4.1 QUALITY METRICS

335 **Specification Quality** Our strong specification baseline, generated via auto-
 336 formalization, contains the ground-truth specification for 7.14% of the LeetCode-derived
 337 problems and 7.87% of the synthetic TagComp problems, shown in Figure 5. In total, this
 338 process yields 161 complex algorithmic data with rigorously verified specifications. This
 339 significantly enriches the publicly available dataset of ground-truth specifications.

340 **Code Transformation Quality** To evaluate the quality of our Python-to-Dafny code
 341 transformation, we attempt to validate 1,011 Dafny programs from the LeetCode set against
 342 the corresponding unit tests. Due to the mismatch between Python and Dafny unit test
 343 formats, we only successfully execute 648 transformed unit test files. However, the trans-
 344 formation is proven highly reliable, with 81.79% of the translated Dafny programs passing
 345 all tests.

346 **Data Complexity** The average Cyclomatic Complexity (McCabe, 1976) quantitatively
 347 manifests the increasing complexity of our data, which counts the number of linearly inde-
 348 pendent paths in the control flow graph. It is computed using the Radon software package
 349 for Python, listed in Table 3.

350 We list the score for MBPP (Austin et al., 2021), since 50 manually annotated data in
 351 DafnySynthesis are based on MBPP-50 and the other 103 are also transformed from it.
 352 Thus, the analysis represents a comparison to DafnySynthesis. We skip the analysis of
 353 CloverBench due to a lack of Python implementations. Our benchmark’s average score of
 354 5.63 is significantly higher than the 2.44 for DafnySynthesis, indicating more complicated
 355 control flows. Notably, our synthetically generated data is even slightly more complex than
 356 the LeetCode-derived portion, with a score that is 0.25 points higher. This complexity is
 357 further validated by a manual rating from Claude-4, which classified the majority of our
 358 synthetic problems as either medium or hard.

359 Table 3: The table compares the code complexity of a previous benchmark and VeriEquiv-
 360 Bench, indicating a more intricate control flow of our data.

361 Dataset	362 MBPP-50	363 MBPP	364 LeetCode	365 TagComp
Average Cyclomatic Complexity	2.44	2.78	5.38	5.63

370 4.2 VALIDATION OF THE EVALUATION METRICS

371 We first validate our equivalence score on 50 expert-written verifiable code provided in
 372 DafnySynthesis. CloverBench has reviewed their data and reported that 10% of the data
 373 does not give the ground-truth specification. After testing on our evaluation metric, the
 374 equivalence score, we figure out another nine examples where the formal specification con-
 375 tains ambiguities or the original code has errors. An example is shown in Figure 4, where the
 376 formal specification does not specify the invariance of array length and leaves a logic gap.
 377 However, only eight examples out of 14 failures are successfully fixed by us, demonstrating

```

378
379
380     method SwapFirstAndLast(a: array<int>)
381         requires a.Length > 0
382         modifies a
383         ##### ↓ The added post-condition
384         ensures a.Length == old(a).Length
385         ##### ↑
386         ensures a[0] == old(a[a.Length - 1])
387         ensures a[a.Length - 1] == old(a[0])
388         ensures forall k :: 1 <= k < a.Length - 1 ==> a[k] ==
389             old(a[k])
390         {
391             var tmp := a[0];
392             a[0] := a[a.Length - 1];
393             a[a.Length - 1] := tmp;
394         }

```

Figure 4: An example of a weak specification in sample #625 that fails equivalence scoring. The formal specification is ambiguous as it omits a post-condition on the invariance of the array’s length.

the hardness in manual annotation. All examples with wrongly claimed ground-truth are listed in Appendix G.4 with the issues stated.

Next, we evaluate all previous benchmarks and observe a serious quality issue in previously provided ground-truth formal specifications, shown in Table 4. It has been discussed that the equivalence check relying on natural language provided in Clover has limitations, and it turns out that a large number of specifications do not establish the equivalence with the code. Meanwhile, DafnyBench is not designed for checking the completeness of specifications and thus, gives the lowest score.

Furthermore, we evaluate Grok-4’s translation ability, using Claude-4-sonnet as a judge (Wang et al., 2025a). We test it on our filtered auto-formalized specifications derived from LeetCode and observe a high success rate of 82.98%, validating it as a reasonable evaluation metric.

Table 4: We present the percentage of data gaining the equivalence score in previous benchmarks.

Dataset	DafnySynthesis	CloverBench	DafnyBench
Equivalence Score	76.22%	61.29%	43.09%

4.3 VERIFIABLE CODE GENERATION

Figure 5 (b) and (c) present the pass@4 results of three proprietary LLMs on end-to-end formally verifiable code generation, tested on CloverBench and our contamination-free synthetic set, TagComp. We also evaluate three open-source model with complete results presented in Figure 23. On the previous CloverBench benchmark, a capable model like Claude achieves a 75.81% success rate, with most errors stemming from issues in specification writing rather than fundamental code generation flaws. However, on our more challenging TagComp dataset, this performance collapses. While the model achieves code-specification equivalence on 10.34% of the problems, more than half of these successes are undermined by incorrect code generation, demonstrating that our benchmark effectively tests both coding and formal reasoning abilities.

A closer look at our benchmark results reveals the challenge of verifiable code generation. While Claude is most capable of producing syntactically correct Dafny code, all three models

struggle significantly with generating mutually equivalent code and specifications aligned with the query intention. In our rigorous two-step evaluation, the equivalence score measures the formal alignment of code and specifications, while the exact matching score further validates against the original natural language intent. Ultimately, fewer than 5% of the generated solutions from any model pass this framework. This result underscores the difficulty of formally verifiable code generation on complex algorithmic problems, requiring strong coding and formal reasoning abilities.

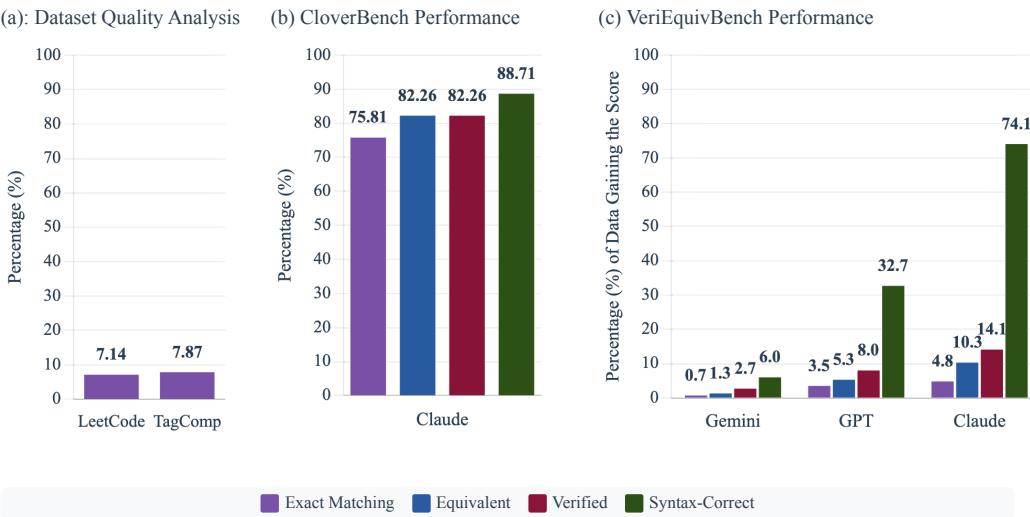


Figure 5: Exact matching score measures the percentage of data passing our two-step evaluation framework, giving solutions aligned with the query intention. Part (a) gives the amount of verified ground-truth solutions in our benchmark. Part (b) shows that the previous CloverBench benchmark is too simple to properly evaluate the advanced reasoning abilities of capable models, as evidenced by a high success rate. Part (c) presents the pass@4 performance of `gemini-2.5-flash`, `gpt-5`, and `claude-4-sonnet` on our end-to-end verifiable code generation task.

4.4 AUXILIARY TASKS

For the two auxiliary tasks mentioned in Section 3, we provide two RL-trained baselines, with the SFT model provided in Veri-Code Series I (2025). As stated, the verifiable code refinement task uses passing the verifier or not as the reward to infill intermediate clauses, while the spec generation task uses the spec superior score introduced by Yan et al. (2025). Spec superior score measures whether the generation specifications described the code better than our weak baseline. The choice of the RL algorithm and hyperparameters follows their implementation as well. We split our LeetCode transformed data into three parts with 1770 training data, 200 validation data and 204 out-of-domain test data, using tags uncovered by the training data.

Our baseline scores 17.68% for the refinement task and 54% for the spec generation task on the validation set. However, in the spec generation task, almost no data generates a complete specification, resulting in an equivalence score. A possible reason is that the SFT model provided is trained on overly simple problems and does not have enough exploration ability. The training curve and results on the test set are presented in Appendix G.3.

5 RELATED WORKS

A central challenge in advancing LLMs is developing metrics that not only assess performance but also provide a clear signal for improvement we desire. **Outcome-based metrics**, such as final-answer accuracy in mathematical reasoning (Cobbe et al., 2021) or pass rates

486 on unit tests in code generation (Austin et al., 2021), are prevalent but limited. They
 487 disregard the fidelity of the reasoning process and remain susceptible to false positives, a
 488 limitation shared by methods employing external solvers for verification (Huang et al., 2025;
 489 Feng et al., 2025).

490 **Formal verification** offers a more rigorous evaluation alternative, using proof checkers like
 491 Dafny (Leino, 2010) or Lean (De Moura et al., 2015) to provide an unambiguous correctness
 492 signal without requiring a ground-truth solution. However, in verifiable code generation,
 493 this signal is fundamentally unidirectional: it validates that the code satisfies a specification
 494 but offers no guarantees about the specification’s quality. This vulnerability allows models
 495 to pass verification using trivial or flawed specifications (Yan et al., 2025). While Yan et al.
 496 (2025) attempt to address this by comparing generated specifications against ground-truth
 497 specifications using a partial order, their method remains dependent on the quality and
 498 availability of trusted ground-truth. In contrast, our work introduces a formal **equivalence**
 499 **metric** that verifies the bidirectional correspondence between code and specification. This
 500 approach ensures the specification fully captures the program’s behavior without relying on
 501 a ground-truth specification.

502 The absence of such a metric has hampered the creation of high-quality benchmarks for **auto-**
 503 **formalization**. Existing datasets often lack the tripartite alignment of natural language,
 504 code, and formal specifications (Lohn & Welleck, 2024; Loughridge et al., 2025; Dougherty
 505 & Mehta, 2025; Yan et al., 2025) or are small-scale due to the high cost of manual annotation
 506 (Misu et al., 2024; Sun et al., 2024; Miranda et al., 2025; Ye et al., 2025b). Attempts
 507 to automate equivalence checking have proven unreliable; for instance, Clover (Sun et al.,
 508 2024) relies on LLM-based judgments that suffer from high error rates. Addressing these
 509 deficiencies, we present **VeriEquivBench**, a benchmark an order of magnitude larger than
 510 prior work. Enabled by our robust equivalence metric, it provides a large-scale, trustworthy
 511 resource for developing and evaluating models for verifiable code generation.

513 6 CONCLUSION

515 In this paper, we confront a foundational challenge hindering the development of reliable
 516 verifiable systems: the dependence on small, manually-annotated benchmarks for formal veri-
 517 fication. This issue limits the scale and complexity of evaluation and has also introduced a
 518 ceiling by human knowledge. Our work breaks the dependency and introduces VeriEquiv-
 519 Bench, a large-scale end-to-end formally verifiable code generation benchmark. Our novel
 520 automated equivalence score provides a rigorous evaluation signal without any need for
 521 human-written, ground-truth specifications. Second, our structured tagging system enables
 522 the scalable, automated synthesis of novel and complex problems, directly addressing the
 523 data generation bottleneck. By using VeriEquivBench to evaluate state-of-the-art LLMs, we
 524 have demonstrated that end-to-end verifiable code generation remains an open challenge, a
 525 fact obscured by the inflated success rates on simpler, older benchmarks. Following the re-
 526 cent discussions on self-evolving agents, our benchmark provides a scalable data generation
 527 engine and a reliable auto-evaluation metric, setting the groundwork to foster trustworthy
 528 AI agents with exact solution generation and sustainably supervise super-intelligence agents.

529 7 REPRODUCIBILITY STATEMENT

532 The code and our dataset are included in the supplementary material and will be publicly
 533 available after the double-blind review process for reproducibility.

535 8 ETHICS STATEMENT

538 This work does not present any foreseeable ethical concerns. The research involves only
 539 publicly available datasets and does not use or analyze sensitive or personally identifiable
 540 information.

540 REFERENCES
541

542 Luogu online judge, 2025. URL <https://www.luogu.com.cn/>. Accessed: 2025-05-28.

543 Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David
544 Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with
545 large language models. *arXiv preprint arXiv:2108.07732*, 2021.

546 Francois Chollet, Mike Knoop, Gregory Kamradt, Bryan Landers, and Henry Pinkard. Arc-
547 agi-2: A new challenge for frontier ai reasoning systems. *arXiv preprint arXiv:2505.11831*,
548 2025.

549 Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz
550 Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher
551 Hesse, and John Schulman. Training verifiers to solve math word problems, 2021. URL
552 <https://arxiv.org/abs/2110.14168>.

553 Domenico Cotroneo, Cristina Improta, Pietro Liguori, and Roberto Natella. Vulnerabilities
554 in ai code generators: Exploring targeted data poisoning attacks. In *Proceedings of the*
555 *32nd IEEE/ACM International Conference on Program Comprehension*, pp. 280–292,
556 2024.

557 David Dalrymple, Joar Skalse, Yoshua Bengio, Stuart Russell, Max Tegmark, Sanjit Seshia,
558 Steve Omohundro, Christian Szegedy, Ben Goldhaber, Nora Ammann, et al. Towards
559 guaranteed safe ai: A framework for ensuring robust and reliable ai systems. *arXiv*
560 *preprint arXiv:2405.06624*, 2024.

561 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International*
562 *conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp.
563 337–340. Springer, 2008.

564 Leonardo De Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob
565 Von Raumer. The lean theorem prover (system description). In *International Conference*
566 *on Automated Deduction*, pp. 378–388. Springer, 2015.

567 Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing
568 Xu, Zhiyong Wu, Tianyu Liu, Baobao Chang, Xu Sun, Lei Li, and Zhifang Sui. A
569 survey on in-context learning. 2023. doi: 10.48550/ARXIV.2301.00234. URL <https://arxiv.org/abs/2301.00234>.

570 Quinn Dougherty and Ronak Mehta. Proving the coding interview: A benchmark for for-
571 mally verified code generation. In *2025 IEEE/ACM International Workshop on Large*
572 *Language Models for Code (LLM4Code)*, pp. 72–79. IEEE, 2025.

573 Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan
574 Jiang, Jinxin Chi, and Wanjun Zhong. Retool: Reinforcement learning for strategic tool
575 use in llms, 2025. URL <https://arxiv.org/abs/2504.11536>.

576 András György, Tor Lattimore, Nevena Lazić, and Csaba Szepesvári. Beyond statis-
577 tical learning: Exact learning is essential for general intelligence. *arXiv preprint*
578 *arXiv:2506.23908*, 2025.

579 Xuhan Huang, Qingning Shen, Yan Hu, Anningzhe Gao, and Benyou Wang. LLMs for
580 mathematical modeling: Towards bridging the gap between natural and mathematical
581 languages. In Luis Chiruzzo, Alan Ritter, and Lu Wang (eds.), *Findings of the Association*
582 *for Computational Linguistics: NAACL 2025*, pp. 2678–2710, Albuquerque, New
583 Mexico, April 2025. Association for Computational Linguistics. ISBN 979-8-89176-195-
584 7. doi: 10.18653/v1/2025.findings-naacl.146. URL <https://aclanthology.org/2025.findings-naacl.146/>.

585 Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Ar-
586 mando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contami-
587 nation free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*,
588 2024.

594 Albert Qiaochu Jiang, Sean Welleck, Jin Peng Zhou, Timothee Lacroix, Jiacheng Liu, Wenda
 595 Li, Mateja Jamnik, Guillaume Lample, and Yuhuai Wu. Draft, sketch, and prove: Guiding
 596 formal theorem provers with informal proofs. In *The Eleventh International Conference
 597 on Learning Representations*.

598 Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and
 599 Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?
 600 In *12th International Conference on Learning Representations, ICLR 2024*, 2024.

601 K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In
 602 *International conference on logic for programming artificial intelligence and reasoning*,
 603 pp. 348–370. Springer, 2010.

604 Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittweiser, Rémi Leblond,
 605 Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level
 606 code generation with alphacode. *Science*, 378(6624):1092–1097, 2022a.

607 Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittweiser, Rémi Leblond,
 608 Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter
 609 Choy, Cyprien de Masson d’Autume, Igor Babuskin, Xinyun Chen, Po-Sen Huang,
 610 Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz,
 611 Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and
 612 Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):
 613 1092–1097, December 2022b. ISSN 1095-9203. doi: 10.1126/science.abq1158. URL <http://dx.doi.org/10.1126/science.abq1158>.

614 Evan Lohn and Sean Welleck. minicodeprops: a minimal benchmark for proving code
 615 properties. *arXiv preprint arXiv:2406.11915*, 2024.

616 Chloe R Loughridge, Qinyi Sun, Seth Ahrenbach, Federico Cassano, Chuyue Sun, Ying
 617 Sheng, Anish Mudide, Md Rakib Hossain Misu, Nada Amin, and Max Tegmark.
 618 Dafnybench: A benchmark for formal software verification. *Transactions on Machine
 619 Learning Research*, 2025. ISSN 2835-8856. URL <https://openreview.net/forum?id=yBgTVWccIx>.

620 Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*,
 621 SE-2(4):308–320, 1976. doi: 10.1109/TSE.1976.233837.

622 Brando Miranda, Zhanke Zhou, Allen Nie, Elyas Obbad, Leni Aniva, Kai Fronsdal, Weston
 623 Kirk, Dilara Soylu, Andrea Yu, Ying Li, et al. Veribench: End-to-end formal verification
 624 benchmark for ai code generation in lean 4. In *2nd AI for Math Workshop@ ICML 2025*,
 625 2025.

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

629 Martin Riddell, Ansong Ni, and Arman Cohan. Quantifying contamination in evaluat-
 630 ing code generation capabilities of language models. In *Proceedings of the 62nd Annual
 631 Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp.
 632 14116–14137, 2024.

633 Parshin Shojaee, Iman Mirzadeh, Keivan Alizadeh, Maxwell Horton, Samy Bengio, and
 634 Mehrdad Farajtabar. The illusion of thinking: Understanding the strengths and lim-
 635 itations of reasoning models via the lens of problem complexity. *arXiv preprint
 636 arXiv:2506.06941*, 2025.

637 David Silver, Satinder Singh, Doina Precup, and Richard S Sutton. Reward is enough.
 638 *Artificial Intelligence*, 299:103535, 2021.

639 Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. Clover: Clo sed-loop ver ifiable
 640 code generation. In *International Symposium on AI Verification*, pp. 134–155. Springer,
 641 2024.

648 Amitayush Thakur, Jasper Lee, George Tsoukalas, Meghana Sistla, Matthew Zhao, Stefan
 649 Zetzsche, Greg Durrett, Yisong Yue, and Swarat Chaudhuri. Clever: A curated bench-
 650 mark for formally verified code generation. *arXiv preprint arXiv:2505.13938*, 2025.

651

652 Shangqing Tu, Kejian Zhu, Yushi Bai, Zijun Yao, Lei Hou, and Juanzi Li. Dice: Detecting
 653 in-distribution contamination in llm's fine-tuning phase for math reasoning. *arXiv preprint*
 654 *arXiv:2406.04197*, 2024.

655 Yutong Wang, Pengliang Ji, Chaoqun Yang, Kaixin Li, Ming Hu, Jiaoyang Li, and Guil-
 656 laume Sartoretti. Mcts-judge: Test-time scaling in llm-as-a-judge for code correctness
 657 evaluation. *CoRR*, 2025a.

658

659 Zhijie Wang, Zijie Zhou, Da Song, Yuheng Huang, Shengmai Chen, Lei Ma, and Tianyi
 660 Zhang. Towards understanding the characteristics of code generation errors made by
 661 large language models. In *2025 IEEE/ACM 47th International Conference on Software*
662 Engineering (ICSE), pp. 717–717. IEEE Computer Society, 2025b.

663

664 Zihan Wang, Jiaze Chen, Zhicheng Liu, Markus Mak, Yidi Du, Geonsik Moon, Luoqi Xu,
 665 Aaron Tua, Kunshuo Peng, Jiayi Lu, Mingfei Xia, Boqian Zou, Chenyang Ran, Guang
 666 Tian, Shoutai Zhu, Yeheng Duan, Zhenghui Kang, Zhenxing Lin, Shangshu Li, Qiang
 667 Luo, Qingshen Long, Zhiyong Chen, Yihan Xiao, Yurong Wu, Daoguang Zan, Yuyi Fu,
 668 Mingxuan Wang, and Ming Ding. Aethercode: Evaluating llms' ability to win in premier
 669 programming competitions, 2025c. URL <https://arxiv.org/abs/2508.16402>.

670

671 Yunhui Xia, Wei Shen, Yan Wang, Jason Klein Liu, Huifeng Sun, Siyue Wu, Jian Hu, and
 672 Xiaolong Xu. Leetcodedataset: A temporal dataset for robust evaluation and efficient
 673 training of code llms, 2025. URL <https://arxiv.org/abs/2504.14655>.

674

675 Zhangchen Xu, Yang Liu, Yueqin Yin, Mingyuan Zhou, and Radha Poovendran. Kodcode:
 676 A diverse, challenging, and verifiable synthetic dataset for coding, 2025. URL <https://arxiv.org/abs/2503.02951>.

677

678 Chuanhao Yan, Fengdi Che, Xuhan Huang, Xu Xu, Xin Li, Yizhi Li, Xingwei Qu, Jingzhe
 679 Shi, Zhuangzhuang He, Chenghua Lin, et al. Re: Form-reducing human priors in scalable
 680 formal software verification with rl in llms: A preliminary study on dafny. *arXiv preprint*
arXiv:2507.16331, 2025.

681

682 Tian Ye, Zicheng Xu, Yuanzhi Li, and Zeyuan Allen-Zhu. Physics of Language Models:
 683 Part 2.1, Grade-School Math and the Hidden Reasoning Process. In *Proceedings of the*
684 13th International Conference on Learning Representations, ICLR '25, April 2025a. Full
 685 version available at <https://ssrn.com/abstract=5250629>.

686

687 Zhe Ye, Zhengxu Yan, Jingxuan He, Timothe Kasriel, Kaiyu Yang, and Dawn Song. Verina:
 688 Benchmarking verifiable code generation. *arXiv preprint arXiv:2505.23135*, 2025b.

689

690 Huaiyuan Ying, Zijian Wu, Yihan Geng, Jiayu Wang, Dahua Lin, and Kai Chen. Lean work-
 691 book: A large-scale lean problem set formalized from natural language math problems.
 692 *Advances in Neural Information Processing Systems*, 37:105848–105863, 2024.

693

694 Boxi Yu, Yuxuan Zhu, Pinjia He, and Daniel Kang. Utboost: Rigorous evaluation of coding
 695 agents on swe-bench. *arXiv preprint arXiv:2506.09289*, 2025.

696

697

698

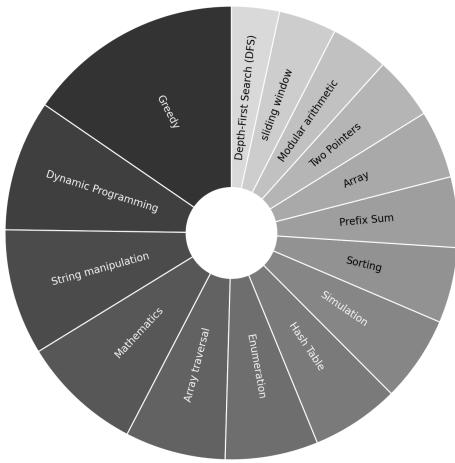
699

700

701

702 A DETAILS ABOUT ALGORITHM TAGS

704 To assemble a suitable tag vocabulary, we first collect high-quality, high-frequency labels
 705 from Luogu—a competitive-programming platform with millions of users and an unusually
 706 fine-grained tag taxonomy—and treat them as a seed set. For each LeetCode problem, the
 707 model is prompted to pick the most relevant domain, data-structure, and algorithm tags
 708 from this pool, and is allowed to introduce new tags only when no suitable match exists. All
 709 model-selected tags are pooled, automatically partitioned into the three coarse categories,
 710 and then manually filtered in a single pass: hallucinated labels are removed, near-duplicates
 711 merged, and overly broad or overly narrow tags discarded. The resulting inventory contains
 712 over 500 clean triples that serve as the controlled vocabulary for subsequent tag-composition.



736 Figure 6: The fifteen most frequently used tags in
 737 our dataset.

Table 5: Statistics of algorithm tags

Tag category	Numbers
Domain	53
Data Structure	68
Algorithm	480

756 The complete curated tag set is listed below, grouped under the three top-level categories:
 757 domain, data-structure, and algorithm.
 758

759 Table 6: Domain tags
 760

761 Category	762 Tags
763 Domain	764 Mathematics, Number Theory, Probability Theory, Combinatorial 765 Mathematics, Linear Algebra, Computational Geometry, Plane geometry, 766 Three-dimensional computational geometry, Graph Theory, Simple 767 Graph Theory, Game Theory, Information Theory, Dynamic Connectivity, 768 expectation, Set Cover Problem, allocation problem, Extremum 769 problem, path problem, Chess Board Problem, Stock Problem, Island 770 Problem, Maze Problem, Josephus problem, Frobenius problem, N- 771 Queens Problem, Knight's Tour Problem, Two-dimensional partial order 772 problem, matching problem, Pairing problem, Interval problems, 773 Knapsack problem, Subset Sum Problem, Jump Game, Maximum Sub- 774 array Problem, Maximum Subsequence Problem, Largest Rectangle in 775 Histogram, longest chain, Path counting, Path Statistics, Connectivity, 776 Reachability analysis, periodic, Discrete Event Simulation, Time 777 constraint, Permutations and Combinations, Counting Principles, Inclusion- 778 Exclusion Principle, Pigeonhole principle, Catalan number, Stirling 779 numbers of the second kind, Combinatorial counting, Combinatorial Opti- 780 mization, Mathematical Techniques

781 Table 7: Data Structure tags
 782

783 Category	784 Tags
785 Data Structure	786 array, Two-dimensional array, Multidimensional array, sorted array, Circular 787 array, tagged array, Difference Array, rolling array, Linked List, doubly 788 linked list, Circular Linked List, Queue, deque, Priority Queue, Stack, 789 monotonic stack, monotonic queue, tree, undirected tree, unrooted tree, 790 Ring tree, Binary Tree, Complete Binary Tree, Perfect Binary Tree, 791 Balanced Binary Tree, Binary Search Tree, Tree data structure, Trie, 792 Segment Tree, Binary Indexed Tree, Heap, heap - min heap, Huffman tree, 793 Set, Hash Table, Adjacency List, Adjacency Matrix, weight graph, Bipartite graph, 794 Complete graph, Undirected graph, directed graph, Reverse graph, Star graph, 795 Directed Acyclic Graph (DAG), Balanced tree, sparse matrix, Disjoint Set Union (DSU), 796 Red-Black Tree, AVL Tree, B-Tree, B+ Tree, Skip List, Bloom Filter, LRU Cache, 797 Prefix Tree, Suffix Tree, Suffix Array, Cartesian Tree, Splay Tree, Scapegoat 798 Tree, Persistent Data Structure, Linear List, Sparse Table, Mo's Algorithm 799 Structure, Leftist Tree, Fibonacci Heap, Pairing Heap

810
811

Table 8: Algorithm tags

812 Category	813 Tags
814 Algorithm-1	815 Compression algorithm,Dynamic Programming,Dynamic Programming 816 - Linear DP,Dynamic Programming-LIS,Dynamic Programming-Prefix 817 Sum,Dynamic Programming - 0/1 Knapsack,Dynamic Programming 818 - State Compression,Dynamic Programming - Interval DP,Dynamic 819 Programming - 2D DP,Dynamic Programming - Prefix Sum Optimization, 820 Dynamic Programming - Top-Down,Dynamic Programming - Iterative, 821 Dynamic Programming,Compression algorithm,Dynamic Programming, 822 Dynamic Programming - Linear DP,Dynamic Programming-LIS,Dynamic 823 Programming-Prefix Sum,Dynamic Programming - 0/1 Knapsack,Dynamic 824 Programming - State Compression,Dynamic Programming - Interval DP,Dynamic 825 Programming - 2D DP,Dynamic Programming - Prefix Sum Optimization,Dynamic 826 Programming - Top-Down,Dynamic Programming - Iterative,Dynamic Programming, 827 State Compression DP,Dynamic Programming (DP),2D DP,Bidirectional 828 DP,Sequence DP,Matrix DP,State Machine DP,Bottom-up Dynamic 829 Programming,Bidirectional BFS,Multi-source BFS,0-1 BFS,Depth-First 830 Search (DFS),Breadth-First Search (BFS),Memoization,State space 831 search,Heuristic search,state search,Grid search,Path Finding,Binary 832 search,Binary Search - Answer,Binary Search - Right Boundary,Binary 833 Search - Left Boundary,Binary Search - Count,Binary Search - Peak 834 Finding,Binary Search - Maximum Value,Binary Search-Prefix 835 Sum,Binary Search - Middle Element,Binary Search - Line Search

836
837
838

Table 9: Algorithm tags

839 Category	840 Tags
841 Algorithm-2	842 Sorting,Merge sort,Quick Sort,Three-way quicksort,Insertion 843 Sort,Counting Sort,Bucket Sort,Sort-Custom Sort,Sorting - Stable 844 Sort,Sorting - Lexicographical Order,Difference Sorting,multi-condition 845 sorting,Wiggle Sort,in-place sorting,Topological sorting,Quick 846 Select,KMP algorithm,Rabin-Karp algorithm,Manacher's algorithm,suffix 847 array,suffix tree,Z-function,prefix function,string pattern matching, 848 string wildcard matching,backtracking,Enumeration,Binary 849 Enumeration,Subset Enumeration,Combinatorial Enumeration,Two- 850 dimensional enumeration,Simulation,Greedy,Greedy - Interval Operation, 851 Divide and conquer,Divide and Conquer - String Splitting,Divide 852 and Conquer - Closest Pair of Points in a Plane,Central Expansion 853 Method,Staining method,Contribution method,sliding window,Two 854 Pointers,Two Pointers - Sliding Window,Fast and slow pointers, 855 Three Pointers,path compression,Path Tracing,Path reconstruction, 856 Path Planning,Single-Source Shortest Path,Multi-Source Shortest 857 Path,Second shortest circuit,Constrained Shortest Path,shortest 858 path,Heap-optimized Dijkstra,Dijkstra's algorithm,Dijkstra's Algo- 859 rithm Variant,Bellman-Ford algorithm,Floyd's cycle-finding algo- 860 rithm,Kruskal's algorithm,Prim's algorithm,Minimum Spanning Tree, 861 Bipartite Matching,Maximum Matching in Bipartite Graphs,Hungarian 862 algorithm,Minimum Cost Maximum Flow,Graham scan,Welzl's algo- 863 rithm,linear sieve,Euler sieve,Eratosthenes sieve,Prime Sieve, Euclidean algorithm,Bézout's identity,Bézout's theorem,Greatest Common Divisor (GCD),Least Common Multiple (LCM),Prime Number Check

864
865

Table 10: Algorithm tags

866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917

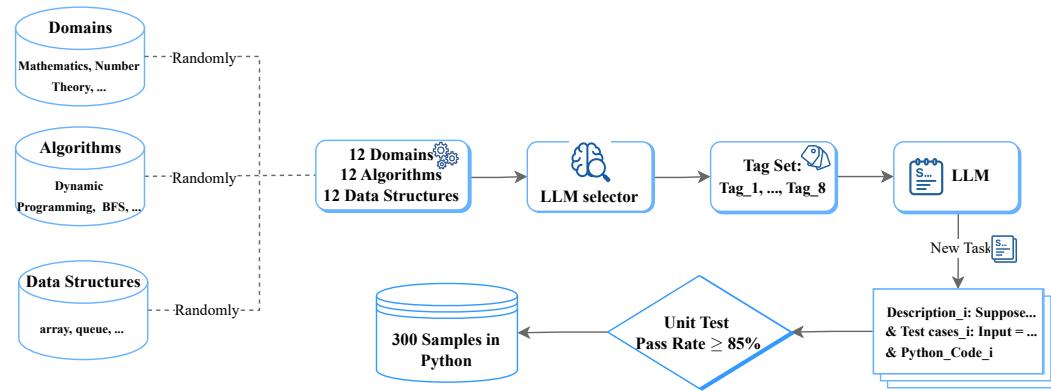
Category	Tags
Algorithm-3	Euclidean algorithm, Bézout's identity, Bézout's theorem, Greatest Common Divisor(GCD), Least Common Multiple(LCM), Prime Number Check, Prime factorization, Factorization, Integer factorization, Cantor expansion, Fast exponentiation, Matrix Fast Exponentiation, Matrix multiplication, matrix rotation, matrix transposition, Matrix operations, rotation matrix, flood fill algorithm, A* algorithm, Tarjan's algorithm, Morris traversal, Preorder Traversal, Inorder Traversal, Postorder traversal, Level order traversal, Level Order Traversal, Reverse inorder traversal, zigzag traversal, spiral traversal, Vertical traversal, Vertical Order Traversal, Boundary traversal, Diagonal Traversal, 2D matrix traversal, Traversal of 2D Array, Graph traversal, Linked list traversal, Tree traversal, Directional traversal, Bidirectional traversal, reverse traversal, Reverse traversal, One-pass traversal, Path Validation, Path counting, Path Statistics, Path Construction, lexicographical comparison, Lexicographically smallest path, Maximum Value Search, Maximum Value Maintenance, Range Maximum, Maximum Column Value, prefix maximum, suffix minimum, suffix product, prefix product, Prefix Sum, Prefix Sum - Difference, Prefix Sum - Modular Arithmetic, Prefix Sum - Binary Search Optimization, 2D prefix sum, suffix sum, partial sum, subarray sum, submatrix sum, Area Sum, Area Calculation, ASCII code manipulation, Character Mapping, Character Count, character frequency, Digital encoding, Digital Parsing, Data Extraction, Number Reversal, Integer Reversal, Integer Square Root, Integer Division, Fraction Addition and Subtraction, Fractional Arithmetic, Fraction simplification, Score Calculation, percentile, Circular shift, Loop Detection, Ring Detection, Periodic Assessment, Bracket Matching, Isomorphic Strings, String comparison, String Case Conversion, String concatenation, string concatenation, String manipulation, String search, string matching, String-Substring Comparison, string-replacement, String replacement, String trimming, string slicing, string splitting, String compression, String decoding, string parsing, string continuity, substring matching, prefix matching, Prefix Check, Longest Common Prefix, Longest Common Suffix, Longest Common Subsequence, Longest Common Subarray, Longest Repeating Substring, Longest Palindromic Subsequence, Longest Non-decreasing Subarray, Longest Consecutive Sequence, longest consecutive characters, Word Chain, Zigzag Conversion, palindrome, Expression parsing, Expression Evaluation, Reverse Polish Notation, Postfix expression, Operator precedence, Lexical Analysis, parsing, Serialization, Deserialization, Encoding, decoding, Run-length encoding, Set Operations, Set Intersection, Bitwise operation, Bitwise operation optimization, Bitwise Operations - State Compression, bitmask, Bitwise OR, AND operation, XOR, binary, Binary Addition, binary splitting, Binary counting, bit count, Hamming distance, Two's complement, Modular arithmetic, modulo 3 operation, Congruence, Congruence theorem, divisible, Divisibility property, divisor, perfect square, square number, Perfect number, Ugly number, trailing zeros, digit separation, Digital Processing, Digital Sum, Gray code, Permutation, Next Permutation, Arrangement, Permutation ring, Cyclic permutation, Pascal's triangle, Fermat's theorem on sums of two, squares, Pythagorean theorem, Triangle inequality, absolute value, absolute value inequality, Big Integer Addition, High precision

Table 11: Algorithm tags

Category	Tags
Algorithm-4	Floating-point processing,Floating-point comparison,floating-point precision,Linear equation,polynomial,Complex Number Operations,Rational number representation,recurring decimal,factorial,Sum of Squares,Sum,Summation formula,arithmetic sequence,Arithmetic sequence summation,path sum,Maximum Sum Path,Maximum spacing,Neighbor Count,Adjacent elements,Adjacent Element Difference,Global Inversion,Local inversion pairs,Inversion pair,anagram,vowel substitution,coordinate,2D coordinates,coordinate system,coordinate comparison,coordinate translation,coordinate compression,2D offset,2D plane,3D space,collinear points,Collinearity detection,convex hull,minimum bounding rectangle,Triangle Area,Rectangle Area Calculation,Overlapping Area Calculation,Rectangle Intersection,Circle-Rectangle Intersection Detection,Minimum Enclosing Circle,Spatial segmentation,2D cutting,Spatial optimization,Space complexity optimization,Constant space complexity,Linear space complexity,Time complexity analysis,Linear time complexity,Linear scan,Pruning,Preprocessing,preprocessing,Offline processing,Dynamic update,Dynamic Maintenance,Dynamic Maintenance Interval,Dynamic Range Maintenance,Single-point modification,Range query,Interval computation,Interval Statistics,Range update,Interval Merging,Interval coverage,Interval Scheduling,Range extrema,Path Intersection Detection,Distance calculation,Euclidean distance,Manhattan distance,Chebyshev distance,projection,cross product,Polar sorting,construct,Binary Construction,Tree Construction,Tree Reconstruction,Sequence Reconstruction,Constructing the answer in reverse order,reverse,Reverse Linked List,Linked List Reversal,String Reversal,Array Rearrangement,Linked List Reordering,Node switching,Segmentation,Split Array,split string,Split and Merge,Convert 1D Array to 2D Array,matrix,2D matrix,sparse matrix,ordered matrix,Rectangle Coverage,Adjacency Matrix,Tree deletion operation,Tree depth,Tree Centroid,Tree Diameter,subtree,Subtree Sum,leaf node,intermediate node,dummy node,sentinel node,Middle of the Linked List,indegree,indegree and outdegree,degree,degree sequence,Monotonicity,Monotonicity Check,monotonic array,Decision Monotonicity,Symmetric,Boolean operations,Logical Operations,Conditional statement,Filter Criteria,Polarity Check,Parity Check,Boundary check,Boundary handling,Edge case handling,Status Check,Status Log,State transition,State Machine,Finite State Automaton,Priority, handling,Query Processing,Path processing,Overflow handling,Carry handling,Recursion,recursive,Inductive method,derivation,traverse,Array traversal,Grid traversal,directional search,State compression,Handling Duplicate Elements,deduplication,Enumeration optimization,Sequence comparison,comparison function,Comparator,Regular Expression,Pointer manipulation,Method chaining,Swap operation,Displacement operation,Row and Column Operations,product,Multiplication Principle,Exponentiation,Base,Base Conversion,Clock issues,loop section,IP address,reordering,Partial Ordering,Equation Solving,Randomization,reverse thinking,Horse Racing Strategy,Connected component,Connected Component,Strongly Connected Component,Lowest Common Ancestor (LCA),Eulerian circuit,Hamiltonian path

972 B PIPELINE OF TAG COMPOSITION
973

974 Figure 7 illustrates our pipeline for generating new programming problems through tag
975 composition. The process begins by creating a candidate pool of 36 tags, randomly selecting
976 12 from each of our three categories: domain, algorithm, and data structure. This pool is
977 provided to an LLM, which is prompted to select a coherent subset of three to eight tags that
978 form a promising basis for a new problem. Using this selected combination, we then instruct
979 the LLM to generate a complete task, comprising a problem description, corresponding unit
980 tests, and a Python solution. As a final quality control step, we filter these generations by
981 executing the unit tests. We retain only those instances where the generated Python code
982 passes all tests, ultimately yielding a dataset of 300 validated programs.

1000 Figure 7: The pipeline for the tag composition process.
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025

1026 C PROMPT TEMPLATES
10271028 C.1 NOVEL TAG COMBINATION
10291030
1031 [Task]

1032 You have three categories of tags: domain, algorithm, and data_structure, each
1033 containing 12 tags. Your task is to select a combination of 3–8 tags from these
1034 categories to form a coherent programming problem. The problem should have a
1035 specified difficulty level: easy, medium, or hard. Ensure the selected tags are compati-
1036 ble and can logically form a single problem. Provide the chosen tags, the difficulty level.
1037

1038 [Requirements]

1039 1.The task is clearly defined, specifying the need to select 3–8 tags from three categories
1040 (domain, algorithm, data_structure) to form a coherent programming problem with a
1041 specified difficulty level.
1042 2.Requirements outline the tag selection process, ensuring compatibility and a reason-
1043 able tag collection, the need for a difficulty level.
1044 3.The selected tags must be compatible and form a reasonable tag collection that
1045 results in a practical and solvable programming problem.
1046 4.The problem must be assigned one of three difficulty levels: easy, medium, or hard,
1047 reflecting the complexity of the problem based on the selected tags.

1048 [Domain tags]

1049 {{ domain_tags }}
1050

1051 [Algorithm tags]

1052 {{ algo_tags }}
1053

1054 [Data Structure tags]

1055 {{ data_tags }}
1056

1057 Output Format

1058 This is the ouput format,You must respond in this specified output format:
1059

```
1060 {  
1061     "all_tags": "Graph Theory, Depth-First Search, Union-Find, Graph,  
1062         Disjoint Set",  
1063     "Domain": "Graph Theory",  
1064     "Algorithm": "Depth-First Search, Union-Find",  
1065     "Data_Structure": "Graph, Disjoint Set" ,  
1066     "Difficulty Level": "medium",  
1067 }  
1068
```

1069 <|Problem End|>

1070 Figure 8: The prompt is for selecting useful tags. We feed the model the 36 real tags from
1071 3 categories randomly that will later drive new-problem generation, it returns the 3–8 tags
1072 that form the most promising combination.
1073
1074
1075
1076
1077
1078
1079

1080
1081 C.2 NOVEL PROBLEM SYNTHESIS
10821083 [Task]
1084

1085 You are an expert algorithm problem creator. Your task is to create an easy or medium
1086 difficulty ranking original coding problem using the given algorithm tags. Analyze the
1087 given tags to generate a new problem. The problem should be completely original
1088 coding problem that is NOT from any existing platforms (LeetCode, Codeforces, etc.)
1089 or textbooks.

1090 [Requirements]
1091

1. Create a truly novel problem scenario with constraints
2. Combine the given tags in innovative ways
3. Ensure the problem is solvable but challenging
4. Provide a clear problem statement, examples, and constraints
5. Rate the difficulty(easy, medium, hard) appropriately

1092 [Algorithm tags]
1093 tags
10941095 Output Format
1096

1097 This is the output format. You must respond in this specified output format:
1098

1099 <|Problem Begin|>
1100 problem
1101 <|Problem End|>
1102

1103
1104 Figure 9: The prompt uses the previously obtained real tags to generate a brand-new
1105 problem.
1106

1107 C.3 SPEC-TO-NL
11081109 [Prompt]
1110

1111 Can you think of a minimal code implementation satisfying the specification? For
1112 example, if the spec just ensures true, then any code can work. If the specification
1113 ensures return values within a range, then assigning any value within the range can
1114 work. Please think of the minimum code implementation and then come up a problem
1115 description this minimal code satisfies.
1116

1117 Below are the specifications:
1118

1119
1120 Figure 10: The prompt asks the model to read the supplied Dafny specification and produce
1121 a concise summary that fully describes the coding problem it defines.
1122

1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133

1134 C.4 LLM-As-A-JUDGE
 1135

1136

1137 [Prompt]

1138

1139 You are an expert in analyzing algorithm problem descriptions. You need to carefully

1140 analyze the equivalence of two algorithm descriptions based on the following dimensions:

1141 1. Core Problem Equivalence: - Is the essence of the problem identical? - Are the

1142 solution objectives consistent?

1143 2. Constraint Comparison: - Input constraints - Boundary case handling - Special case

1144 requirements

1145 3. Complexity Requirements: - Time complexity requirements - Space complexity

1146 requirements

1147 4. Detail Completeness: - Information loss check - Additional information analysis

1148 Please provide an equivalence score from 0-100 and give a detailed analysis of your

1149 reasoning.

1150 Please analyze the equivalence between the following two algorithm descriptions:

1151 Original Description:

1152 New Description:

1153 Please analyze according to the dimensions above and provide a score with detailed

1154 explanation. Only put the score in a code block surrounded by triple backticks ("`")`"

1155

1156 Figure 11: The prompt instructs the model to determine whether the two given programming

1157 problems are semantically equivalent.

1158

1159

1160

1161

1162

1163

1164

1165

1166

1167

1168

1169

1170

1171

1172

1173

1174

1175

1176

1177

1178

1179

1180

1181

1182

1183

1184

1185

1186

1187

1188 C.5 NL QUERY TO VERIFIABLE CODE
1189

1190

1191 [Prompt]
11921193 You will get a problem description. Your task is to give a fully verified Dafny program.
11941195 Refer to the Dafny examples as guidance:
11961197 Fewshot Examples:
11981199 Problem description:
12001201 Please write the Dafny code that implements the functionality while ensuring:
12021203 1. Reference the Python implementation for algorithmic insights;
12041205 2. Add appropriate loop invariants with brief explanations;
12061207 3. Ensure full verification - your code must pass the Dafny verifier.
12081209 Output the complete Dafny program, including both the specification and implemen-
1210 tation.
12111212 Figure 12: The prompt turns a natural-language query into a fully formal, verifiable speci-
1213 fication together with correct-by-construction code.
1214

1215

1216 D MODEL ARGUMENT SETTINGS
12171218 Throughout all experiments, we retained the default values for every hyperparameter except
1219 temperature and top-p. To balance creativity with reliability, we employed a two-level
1220 sampling strategy: during the initial specification-generation stage shown in Pipeline 1 in
1221 Figure 2, temperature was set to 0.7 and top-p to 0.9 to encourage diversity for generating
1222 high-quality formal specifications equivalent to the NL query(Li et al., 2022b).
12231224 In all other phases, including annotated code generation in Pipeline 3 in Figure 2 and model
1225 evaluation, temperature was reduced to 0.5 and top-p to 0.8 to promote deterministic and
1226 consistent outputs.
12271228 The prompts are provided in Section C.
12291230 For model evaluation, the coding agent is provided with the problem in natural language
1231 and is asked to generate four rollouts of annotated Dafny code. The equivalence score is then
1232 evaluated for each rollout. Next, those rollouts that gain the equivalent score are passed
1233 to Grok-4 to translate specifications back into NL. Finally, Claude-4 judges the equivalence
1234 between the translated new description and the original query.
1235

1236

1237

1238

1239

1240

1241

1242 E QUALITATIVE ANALYSIS
12431244 E.1 EXAMPLES OF VERIFICATION FAILURES
12451246 Most failures come from unprovable clauses, including missing intermediate proofs or un-
1247 specified conditions, as shown in Figure 13. However, to be noticed, Dafny has strict re-
1248 quirements for writing specifications in order to pass the verifier. We have provided two
1249 examples, whose specifications are correct and follow the syntax rules, in Figure 14 and 15.
1250 However, the Dafny verifier requires re-expressing the code in a different way to pass the
1251 verifier.
1252

1253 [Example 1]

```
1255     decreases grid.Length0 * grid.Length1 - CountVisitedLandCells(grid
1256         , visited)
```

1257 Error: *decreases* expression might not decrease.
1258 Error: *decreases* expression must be bounded below by 0 at the end of the loop iteration.

1260 [Example 2]

```
1262     totalCost := totalCost + energyCosts[reachable[i]];
```

1263 Error: index out of range.

1264
1265 Figure 13: We provide two examples which cannot pass the verifier with missing intermediate
1266 clauses.

```
1270     while i <= |text| - |pattern|
1271     invariant 0 <= i <= |text| - |pattern| + 1
1272     invariant forall j :: 0 <= j < i ==> text[j..j+|pattern|] != pattern
1273     {
1274         if i + |pattern| <= |text| && text[i..i+|pattern|] == pattern {
1275             return true;
1276         }
1277         i := i + 1;
1278     }
```

1279 Figure 14: An example of Claude-generated code and specifications that cannot be verified
1280 by the Dafny verifier, although all specifications are correctly written.1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295

```

1296
1297
1298
1299
1300
1301     method MountainPathNavigation(elevations: seq<int>, queries: seq<int>
1302         >) returns (results: seq<int>)
1303         requires |elevations| > 0
1304         requires forall i :: 0 <= i < |elevations| - 1 ==> elevations[i] <=
1305             elevations[i+1] // non-decreasing
1306         requires forall k :: k in queries ==> 0 <= k < |elevations|
1307         ensures |results| == |queries|
1308         ensures forall i :: 0 <= i < |results| ==>
1309             (results[i] == -1 <=> (forall pos :: 
1310                 queries[i] <= pos < |elevations| ==> !HasDescendingPath(
1311                     elevations, pos))) &&
1312             (results[i] != -1 ==> queries[i] <= results[i] < |elevations| &&
1313                 HasDescendingPath(elevations, results[i]) &&
1314                 (forall pos :: queries[i] <= pos < results[i] ==> !
1315                     HasDescendingPath(elevations, pos)))
1316     {
1317         // First, precompute all valid positions
1318         var validPositions := FindValidPositions(elevations);
1319
1320         // Answer each query using binary search
1321         results := [];
1322         for i := 0 to |queries|
1323             invariant |results| == i
1324             invariant forall j :: 0 <= j < i ==>
1325                 (results[j] == -1 <=> (forall pos :: 
1326                     queries[j] <= pos < |elevations| ==> !HasDescendingPath(
1327                         elevations, pos))) &&
1328                 (results[j] != -1 ==>
1329                     queries[j] <= results[j] < |elevations| &&
1330                     HasDescendingPath(elevations, results[j]) &&
1331                     (forall pos :: queries[j] <= pos < results[j] ==> !
1332                         HasDescendingPath(elevations, pos)))
1333             {
1334                 var result := BinarySearchLeftmost(validPositions, queries[i]);
1335                 results := results + [result];
1336             }
1337         }
1338
1339         predicate HasDescendingPath(elevations: seq<int>, start: int)
1340             requires 0 <= start < |elevations|
1341             {
1342                 exists end :: start < end < |elevations| &&
1343                     (forall k :: start <= k < end ==> elevations[k] > elevations[
1344                         k+1])
1345             }
1346
1347
1348
1349

```

Figure 15: An example of Claude-generated code and specifications that cannot be verified by the Dafny verifier. Although all specifications are correctly written, it does not follow the Dafny grammar.

1350 The example in Figure 15 shows the limitation of the Dafny verifier. Consider the following
 1351 two preconditions:

1352 requires forall $k :: k$ in queries $\Rightarrow 0 \leq k < |\text{elevations}|$,

1353 and

1354 requires forall $i :: 0 \leq i < |\text{queries}| \Rightarrow 0 \leq \text{queries}[i] < |\text{elevations}|$.

1355 Although two preconditions express the same semantic content, the first condition generated
 1356 by Claude causes verification errors for the highlighted part in Figure 15; the range of each
 1357 element in queries cannot be proven. However, switching to the second precondition solves
 1358 the issue because the second precondition limits the range of each position needed for the
 1359 verifier.

1360

1361 E.2 AN EXAMPLE OF AMBIGUOUS SPECIFICATIONS

1362

1363 In this subsection, we present an example whose specifications are too weak to describe the
 1364 code behaviour and cannot pass our whole pipeline without alignment with the original user
 1365 intention.

1366

1367 **Problem Description** You are a security consultant for a museum that has a complex
 1368 layout of interconnected rooms. The museum has motion sensors that detect when visitors
 1369 move between rooms, and you need to validate if a recorded sequence of room visits repre-
 1370 sents a valid path through the museum. The museum layout is represented as an adjacency
 1371 matrix where 1 indicates a direct connection between two rooms, and 0 indicates no direct
 1372 connection. Additionally, the museum has special 'checkpoint rooms' that visitors must pass
 1373 through in a specific order when moving between certain sections. Your task is to validate
 1374 a given path and determine if it's physically possible given the room connections, and also
 1375 verify that all checkpoint rooms are visited in the correct sequence.

1376 First line of the input contains integer n (number of rooms). Next n lines contain the
 1377 adjacency matrix ($n \times n$) representing room connections. Next line contains integer k
 1378 (number of checkpoint rooms). Next line contains k integers representing the required order
 1379 of checkpoint rooms. Finally, the last line contains the path as a sequence of room numbers
 1380 to validate.”,

1381 Return 'VALID' if the path is valid (all consecutive rooms are connected and checkpoints
 1382 are visited in order), 'INVALID_CONNECTION' if there's an invalid room transition,
 1383 'INVALID_CHECKPOINT' if checkpoints are not visited in the required order, or 'MISS-
 1384 ING_CHECKPOINT' if not all checkpoints are visited.

1385

1386 **Ambiguous Spec Generated By Claude** The specification is shown in Figure 16.

1387

```
1388
1389   method ValidateMuseumPath(n: int, adjacency_matrix: array2<int>,
1390     checkpoints: array<int>, path: array<int>)
1391   returns (result: string)
1392   requires n >= 0
1393   requires adjacency_matrix.Length0 == n &&
1394     adjacency_matrix.Length1 == n
1395   requires forall i, j :: 0 <= i < n && 0 <= j < n ==>
1396     adjacency_matrix[i, j] == 0 || adjacency_matrix[i, j] == 1
1397   requires forall i :: 0 <= i < checkpoints.Length ==>
1398     0 <= checkpoints[i] < n
1399   ensures result in {"VALID", "INVALID_CONNECTION", "INVALID_CHECKPOINT"
1400     ", "MISSING_CHECKPOINT"}
```

1401 Figure 16: Here is an example whose postconditions are too loose to describe the code
 1402 behaviour.

1403

```

1404
1405
1406     method ValidateMuseumPath(n: int, adjacencyMatrix: array2<int>, k:
1407         int, checkpoints: array<int>, path: array<int>) returns (result:
1408         int)
1409
1410     requires 3 <= n <= 100
1411     requires 1 <= k <= n
1412     requires k <= path.Length <= 1000
1413
1414     requires adjacencyMatrix.Length0 == n && adjacencyMatrix.Length1 == n
1415     requires checkpoints.Length == k
1416     requires path.Length >= 1
1417     requires forall i, j :: 0 <= i < n && 0 <= j < n ==>
1418         adjacencyMatrix[i, j] == 0 || adjacencyMatrix[i, j] == 1
1419     requires forall i :: 0 <= i <= k ==> 0 <= checkpoints[i] < n
1420     requires forall i :: 0 <= i < path.Length ==> 0 <= path[i] < n
1421     requires forall i, j :: 0 <= i < j < k ==>
1422         checkpoints[i] != checkpoints[j]
1423
1424     ensures 0 <= result <= 3
1425
1426     ensures result == 0 <=> (
1427         (forall i :: 0 <= i < path.Length - 1 ==>
1428             adjacencyMatrix[path[i], path[i+1]] == 1) &&
1429             (forall cp :: 0 <= cp < k ==> exists i :: 0 <= i < path.Length &&
1430                 path[i] == checkpoints[cp]) &&
1431                 (forall cp1, cp2 :: 0 <= cp1 < cp2 < k ==>
1432                     exists i1, i2 :: 0 <= i1 < i2 < path.Length &&
1433                         path[i1] == checkpoints[cp1] && path[i2] == checkpoints[cp2])
1434     )
1435
1436     ensures result == 1 <=> (
1437         exists i :: 0 <= i < path.Length - 1 &&
1438             adjacencyMatrix[path[i], path[i+1]] == 0
1439     )
1440
1441     ensures result == 3 <=> (
1442         (forall i :: 0 <= i < path.Length - 1 ==>
1443             adjacencyMatrix[path[i], path[i+1]] == 1) &&
1444             (exists cp :: 0 <= cp < k && forall i :: 0 <= i < path.Length ==>
1445                 path[i] != checkpoints[cp])
1446     )
1447
1448     ensures result == 2 <=> (
1449         (forall i :: 0 <= i < path.Length - 1 ==>
1450             adjacencyMatrix[path[i], path[i+1]] == 1) &&
1451             (forall cp :: 0 <= cp < k ==> exists i :: 0 <= i < path.Length &&
1452                 path[i] == checkpoints[cp]) &&
1453                 (exists cp1, cp2 :: 0 <= cp1 < cp2 < k &&
1454                     forall i1, i2 :: (0 <= i1 < i2 < path.Length &&
1455                         path[i1] == checkpoints[cp1] &&
1456                             path[i2] == checkpoints[cp2]) ==> false)
1457     )
1458
1459     ensures result == 1 ==> (
1460         exists i :: 0 <= i < path.Length - 1 &&
1461             adjacencyMatrix[path[i], path[i+1]] == 0
1462     )
1463
1464     ensures (forall i :: 0 <= i < path.Length - 1 ==>
1465             adjacencyMatrix[path[i], path[i+1]] == 1) ==>
1466             (result == 0 || result == 2 || result == 3)

```

Figure 17: Here is the strong specification generated using our pipeline. Our pipeline translates natural language to specifications first and then reinserts them into the generated code.

1458 **Translation by Grok-4** This method addresses a simplified museum path validation
 1459 problem, where the museum is modeled as a graph with n rooms (nodes) connected via a
 1460 binary adjacency matrix, a set of checkpoint rooms that might need to be visited, and a pro-
 1461 posed path sequence through the rooms. However, due to a policy of unconditional approval
 1462 (perhaps for a prototype or always-valid scenario), the function always deems the path valid
 1463 and returns "VALID", without checking connections, checkpoints, or path validity. It as-
 1464 sumes all inputs are well-formed (valid matrix size, binary entries, and checkpoint ranges)
 1465 and provides no detailed error reporting, effectively serving as a placeholder validator that
 1466 optimistically approves any tour route in reinforcement learning datasets for graph traversal
 1467 experiments. If needed, it could return other statuses like "INVALID_CONNECTION" in
 1468 variants, but here it guarantees approval to satisfy minimal validation specs.

1469 **Equivalence Judgment by Claude-4** Final Equivalence Score: 0.

1471 These two descriptions have zero equivalence. They describe completely unrelated prob-
 1472 lems: Different domains: Mountain elevation analysis vs. graph-based museum navigation;
 1473 Different data structures: 1D sorted array vs. 2D adjacency matrix with paths; Different
 1474 operations: Search for valid positions vs. unconditional approval; Different computational
 1475 requirements: Actual algorithm vs. placeholder function; Different I/O formats: Numerical
 1476 queries/positions vs. validation strings.

1477 The only superficial similarity is that both involve some form of path or traversal concept,
 1478 but even this is fundamentally different (descending sequences in an array vs. connected
 1479 paths in a graph). There is no meaningful relationship between these two algorithm descrip-
 1480 tions.

1481
 1482 **Specification from Our Pipeline** Figure 17 presents the generated specifications using
 1483 our pipeline for the same question. Our pipeline translates natural language to specifications
 1484 first and then reinserts them into the generated code. It leads to a noticeable improvement
 1485 in quality compared to code and specification generation being done simultaneously. This
 1486 reveals an interesting weakness of current models: models can separately generate speci-
 1487 fications and code, but fail to do them together.

1488 E.3 A HARD EXAMPLE FOR SPEC-TO-NL TRANSLATION

1491 In the above example, we remove any post-conditions on verifying the connectivity of the
 1492 path in the specification to create a hard test example for Spec-to-NL translation. This dif-
 1493 ference is successfully captured when translating the specifications back to natural language
 1494 and observed by the Judge model.

1495 **Translation by Grok-4** Problem: Subsequence Order Validator

1497 Given an integer n (unused, for compatibility), an $n \times n$ adjacency matrix (unused, for
 1498 compatibility), an integer k ($1 \leq k \leq n$), an array of k unique checkpoints (room indices
 1499 from 0 to $n-1$), and an array representing a path (sequence of room indices from 0 to $n-1$,
 1500 length at least k), validate whether the path contains the exact sequence of checkpoints
 1501 as a (possibly non-consecutive) subsequence. Return 0 if it does (VALID), 2 if all check-
 1502 points appear but at least one pair has all occurrences of the earlier checkpoint after all
 1503 occurrences of the later one (INVALID_CHECKPOINT order violation), 3 if at least one
 1504 checkpoint does not appear in the path (MISSING_CHECKPOINT), and never return 1
 1505 (INVALID_CONNECTION, as connections are not checked). This can be used to check if
 1506 a sequence of events includes required milestones in a specified relative order, ignoring any
 1507 graph structure.

1508 **Equivalence Judgment by Claude-4** Equivalence Score: 30.

1509 Why this low score (30/100): Missing core functionality (-40 points): The new description
 1510 explicitly removes graph connectivity validation, which is a major component of the original
 1511 problem. This is not a minor detail - it's half the problem.

1512 Different output behavior (-20 points): Cannot return INVALID_CONNECTION, leading
 1513 to incorrect results for inputs with connection violations.
 1514
 1515 Explicit admission of incompleteness (-10 points): The new description acknowledges inputs
 1516 are "unused, for compatibility," indicating it's deliberately solving a subset problem.
 1517
 1518 Why not lower (0-30): Shared checkpoint validation (+20 points): The checkpoint ordering
 1519 logic appears identical between both descriptions.
 1520
 1521 Same input format (+5 points): Both accept the same input structure (even if new ignores
 1522 parts).
 1523
 1524 Overlapping outputs (+5 points): Three of four output codes have the same meaning.

1524 E.4 AN EXAMPLE WITH TIGHT SPECIFICATIONS BUT MISMATCHED CODE

1525 We also provide another example of wrongly generated code. Although the generated spec-
 1526 ifications accurately describe the code, the alignment between the code and the natural-
 1527 language query is not guaranteed and requires a second-step verification. Our pipeline
 1528 successfully captures the mismatch, which cannot be achieved by any other code generation
 1529 benchmark. Considering that this incorrect code passes 85% of co-generated unit tests, we
 1530 doubt the reliability of using LLM-generated unit tests to verify code.
 1531

1532 **Problem Description** A chess tournament organizer needs to create a bracket system
 1533 for N players. The tournament follows a complete binary tree structure where each leaf
 1534 represents a player, and each internal node represents a match between winners of its sub-
 1535 trees. The organizer has a sorted array of player ratings and wants to generate all possible
 1536 valid tournament brackets using backtracking. A valid bracket must satisfy: 1) Each player
 1537 appears exactly once as a leaf, 2) The tournament tree is complete (all levels filled except
 1538 possibly the last), 3) Players are assigned to positions such that the bracket can accom-
 1539 modate exactly N players. You need to find the number of distinct ways to arrange players in
 1540 valid tournament brackets.

1541 **Incorrect Code** The code, with its corresponding specification, is shown in Figure 18.
 1542

1543 **Translation by Grok-4** Problem: Tournament Outcome Counter You are organizing
 1544 a single-elimination tournament where players are eliminated one by one until only the
 1545 champion remains. Given a list of players (with 1 to 16 players), write a function tourna-
 1546 ment_bracket_count that calculates the total number of possible outcomes for the tourna-
 1547 ment, assuming a fixed bracket structure. Each game has exactly two possible results (one
 1548 player wins, the other loses), and a tournament with n players requires exactly $n-1$ games.
 1549 Therefore, the number of possible outcomes is $2^{(n-1)}$.
 1550

1551 Input: A list of players (e.g., a sequence of integers representing player IDs).
 1552

1553 Constraints: $1 \leq$ number of players ≤ 16 .
 1554

1555 Output: An integer representing the number of possible tournament outcomes.
 1556

1557 **Equivalence Judgment by Claude-4** Equivalence Score: 5.
 1558

1559 The score of 5/100 reflects that while the new description uses similar terminology (peaks,
 1560 revenues, maximum sum), it describes implementations that: Do not solve the original
 1561 problem - returning constant 1 or the first element is not finding a maximum mountain-
 1562 pattern subsequence; Ignore all constraints - no mountain pattern validation whatsoever;
 1563 Have no algorithmic content - $O(1)$ stub vs. $O(n \times m)$ algorithm; Are explicitly described as
 1564 placeholders - the text itself admits these are "simplified" versions that "ignore the actual
 1565 revenue data".

1566 The 5 points are awarded only for: Using the same domain context (ski resort/mountain
 1567 revenues); Sharing vocabulary (peaks, revenues, maximum); Acknowledging the existence
 1568 of input data structures.

```

1566     function exponential(base: int, exp: nat): int
1567         decreases exp;
1568     {
1569         if exp == 0 then 1
1570         else base * exponential(base, exp-1)
1571     }
1572
1573     method TournamentBracketCount(players: seq<int>) returns (count: int)
1574         requires 1 <= |players| <= 16
1575         ensures count == exponential(2, |players|-1)
1576     {
1577         var n: int := |players|;
1578         count := 1;
1579         var i: int := 1;
1580         if n == 1 {
1581             return;
1582         }
1583         while i < n
1584             invariant 1 <= i <= n
1585             invariant count == exponential(2, i-1)
1586             decreases n - i
1587         {
1588             count := count * 2;
1589             i := i + 1;
1590         }
1591     }
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619

```

Figure 18: Here is an example whose generated code does not satisfy the user request but has tight specifications generated along with it.

This is essentially not an equivalent algorithm but rather a pair of stub functions that might be used as placeholders before implementing the actual solution. The new description reads like documentation for incomplete/prototype code rather than a working implementation of the mountain revenue problem.

1620 F EXAMPLES OF SYNTHESIS PROBLEMS
1621

```

1622 {
1623     "title": "Magical Crystal Collection",
1624     "difficulty": "medium",
1625     "description": "You are an adventurer exploring a magical cave with n
1626         crystals, each having a power value. The cave has a special
1627         property: you can only collect crystals whose total power equals
1628         exactly one of k magical target values. However, there's a twist
1629         - the cave becomes unstable after collecting crystals, so you
1630         want to find the target value that can be achieved using the
1631         minimum number of crystals. If multiple targets require the same
1632         minimum number of crystals, return the smallest target value. You
1633         need to efficiently determine which magical target is achievable
1634         and requires the fewest crystals.",
1635     "input_format": "First line contains two integers n and k. Second
1636         line contains n integers representing crystal power values. Third
1637         line contains k integers representing the magical target values
1638         .",
1639     "output_format": "Return the magical target value that can be
1640         achieved with minimum crystals, or -1 if no target is achievable
1641         .",
1642     "constraints": [
1643         "1 <= n <= 20",
1644         "1 <= k <= 1000",
1645         "1 <= crystal power <= 1000",
1646         "1 <= target values <= 20000",
1647         "All crystal powers are positive integers",
1648         "All target values are distinct"
1649     ],
1650     "examples": [
1651         {
1652             "input": "4 3\n2 3 5 7\n10 12 17",
1653             "output": "10",
1654             "explanation": "Target 10 can be achieved with crystals [3,7]
1655                 (2 crystals), target 12 can be achieved with [2,3,7] (3
1656                 crystals), target 17 can be achieved with [2,3,5,7] (4
1657                 crystals). Minimum is 2 crystals for target 10."
1658         },
1659         {
1660             "input": "3 2\n1 4 6\n8 11",
1661             "output": "11",
1662             "explanation": "Target 8 cannot be achieved with any subset.
1663                 Target 11 can be achieved with [1,4,6] (3 crystals). So
1664                 answer is 11."
1665         },
1666         {
1667             "input": "3 3\n2 4 8\n6 14 16",
1668             "output": "6",
1669             "explanation": "Target 6 can be achieved with [2,4] (2
1670                 crystals), target 14 can be achieved with [2,4,8] (3
1671                 crystals), target 16 cannot be achieved. Minimum is 2
1672                 crystals for target 6."
1673         }
1674     ],
1675     "tags_used": ["Subset Sum Problem", "Binary search", "bitmask", "
1676         array", "Heap"],
1677     "time_complexity": "O(2^n * k + k log k)",
1678     "space_complexity": "O(2^n + k)"
1679 }

```

1673 Figure 19: Problem 301's natural-language description and example unit test.

```

1674
1675 F.1 QUALIFIED PYTHON CODE GENERATED
1676
1677 def magical_crystal_collection(n, k, crystals, targets):
1678     """
1679         Find the magical target value that can be achieved with minimum
1680         crystals.
1681
1682     Args:
1683         n: number of crystals
1684         k: number of target values
1685         crystals: list of crystal power values
1686         targets: list of magical target values
1687
1688     Returns:
1689         The target value achievable with minimum crystals, or -1 if none
1690         achievable
1691         """
1692         # Dictionary to store minimum crystals needed for each possible sum
1693         min_crystals_for_sum = {}
1694
1695         # Generate all possible subsets using bit manipulation
1696         for mask in range(1, 1 << n): # Start from 1 to exclude empty subset
1697             subset_sum = 0
1698             crystal_count = 0
1699
1700             for i in range(n):
1701                 if mask & (1 << i):
1702                     subset_sum += crystals[i]
1703                     crystal_count += 1
1704
1705                     # Update minimum crystals needed for this sum
1706                     if subset_sum not in min_crystals_for_sum or crystal_count <
1707                         min_crystals_for_sum[subset_sum]:
1708                             min_crystals_for_sum[subset_sum] = crystal_count
1709
1710             # Find the best target
1711             best_target = -1
1712             min_crystals_needed = float('inf')
1713
1714             for target in targets:
1715                 if target in min_crystals_for_sum:
1716                     crystals_needed = min_crystals_for_sum[target]
1717                     if (crystals_needed < min_crystals_needed or
1718                         (crystals_needed == min_crystals_needed and target <
1719                             best_target)):
1720                         min_crystals_needed = crystals_needed
1721                         best_target = target
1722
1723             return best_target
1724
1725
1726
1727

```

Figure 20: Problem 301 passes all 40 unit tests.

1728 F.2 UNQUALIFIED PYTHON CODE GENERATED

```

1729
1730
1731 def forest_fire_spread(n, firefighters, edges):
1732     """
1733     Find the maximum total number of firefighters that can be impacted
1734     when starting the fire from the optimal station.
1735
1736     Args:
1737         n: number of stations
1738         firefighters: list of firefighters at each station (0-indexed)
1739         edges: list of tuples representing bidirectional trails
1740
1741     Returns:
1742         Maximum total number of firefighters that can be impacted
1743     """
1744     # Build adjacency list
1745     graph = [[] for _ in range(n)]
1746     for u, v in edges:
1747         graph[u-1].append(v-1) # Convert to 0-indexed
1748         graph[v-1].append(u-1)
1749
1750     def dfs(start, visited, current_firefighters):
1751         """
1752         Perform DFS to spread fire from current station.
1753         Fire spreads to neighbor only if current station has more
1754         firefighters.
1755         """
1756         visited.add(start)
1757         total_impact = firefighters[start]
1758
1759         for neighbor in graph[start]:
1760             if neighbor not in visited and current_firefighters >
1761                 firefighters[neighbor]:
1762                 total_impact += dfs(neighbor, visited, firefighters[
1763                     neighbor])
1764
1765         return total_impact
1766
1767     max_impact = 0
1768
1769     # Try starting fire from each station
1770     for start_station in range(n):
1771         visited = set()
1772         impact = dfs(start_station, visited, firefighters[start_station])
1773         max_impact = max(max_impact, impact)
1774
1775     return max_impact
1776
1777
1778
1779
1780
1781

```

Figure 21: Problem 1547 has 22 passing and 19 failing test-cases. The DFS function incorrectly propagates state.

```

1782
1783
1784 def digital_library_traversal(input_str):
1785     """
1786         Performs preorder traversal of book dependencies starting from a
1787         given book.
1788
1789     Args:
1790         input_str: String containing book dependencies and starting book
1791
1792     Returns:
1793         String with books in preorder traversal order, one per line
1794     """
1795     lines = input_str.strip().split('\n')
1796     n = int(lines[0])
1797
1798     # Parse dependencies
1799     dependencies = {}
1800
1801     for i in range(1, n + 1):
1802         line = lines[i]
1803         if ':' in line:
1804             book, deps_str = line.split(':', 1)
1805             book = book.strip()
1806             deps_str = deps_str.strip()
1807
1808             if deps_str:
1809                 deps = [dep.strip() for dep in deps_str.split(',')]

1810                 # Sort dependencies alphabetically
1811                 deps.sort()
1812             else:
1813                 deps = []
1814
1815             dependencies[book] = deps
1816
1817         starting_book = lines[n + 1].strip()
1818
1819         # Check if starting book exists
1820         if starting_book not in dependencies:
1821             return "Book not found"
1822
1823         # Perform preorder traversal
1824         result = []
1825
1826         def preorder_traversal(book):
1827             # Visit current book
1828             result.append(book)
1829
1830             # Visit dependencies in alphabetical order
1831             if book in dependencies:
1832                 for dependency in dependencies[book]:
1833                     if dependency in dependencies:
1834                         preorder_traversal(dependency)
1835
1836         preorder_traversal(starting_book)
1837
1838         return '\n'.join(result)
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885

```

Figure 22: Problem 508 has 38 passing and 2 failing test-cases. Two unit tests violate the problem's stated constraints, preventing the otherwise-correct solution from passing all cases.

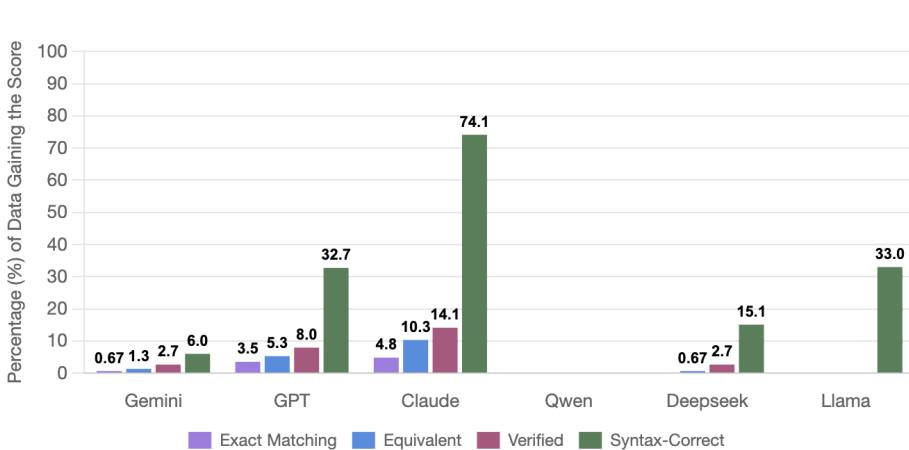


Figure 23: We evaluate three proprietary models and three open-sourced models using our pipeline to verify the alignment between users’ intention and code generation. It turns out that business-purpose models in large sizes outperform small, open-sourced models, especially in writing less ambiguous specifications.

G EXPERIMENTAL RESULT DETAILS

G.1 QUALITY ASSURANCE OF PYTHON2DAFNY TRANSFORMATION

With 1,011 questions randomly selected, we transform their unit tests in Python to Dafny. However, due to the mismatch in Dafny grammar, not all questions are successfully transformed without syntax errors; for example, our script uses `seq<int>` to define lists, but some Dafny code requires `array<int>`. Also, not all unit tests satisfy the preconditions, in which cases, the unverified Dafny code cannot be executed. Also, a few codes take too long to compile and are stopped after 30 minutes. The details are listed in Table 12.

Finally, with 648 successfully executed code, 530 pass all unit tests with a full pass rate at 81.79%.

Table 12: The table shows the success rate of transforming Python unit tests to Danfy.

Selected Questions	Syntax Errors	Verification Errors	Timeout	Successful Execution
1011	297	62	4	648
100%	29.38%	6.13%	0.4%	64.1%

G.2 TESTING ON MORE OPEN-SOURCED MODELS

We evaluate three proprietary models and three open-sourced models using our pipeline to verify the alignment between users’ intentions and code generation. Here, we use Claude-4-sonnet, GPT-5, Gemini-2.5-flash, DeepSeek-R1, Qwen-2.5-Coder-14B-Instruct and Llama3-70B. Business-purpose models significantly outperform open-sourced models in writing syntax-correct and tight specifications.

G.3 TRAINING CURVES ON AUXILIARY TASKS

We use the 14B SFT model provided by the Veri-Code Team and their code to RL-train models using GRPO.

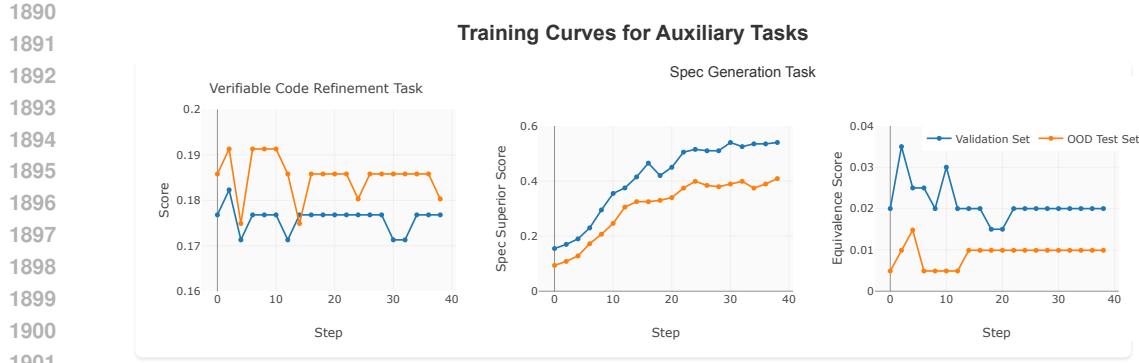


Figure 24: During the verifiable code refinement task, the model barely improves, demonstrating that RL training is not enough. During the spec generation task, the generated specification quality keeps enhancing, but still fails to capture code behaviours without ambiguities.

G.4 DETAILS ABOUT DAFNYSYNTHESIS INSPECTION

This section details our analysis of 14 ground-truth samples identified as problematic. Our investigation revealed that two samples failed initial verification due to implementation errors or timeouts, precluding further analysis. These were #566 and #632, the latter of which was previously reported by Clover (Sun et al., 2024).

The primary issue in the remaining 12 samples was specification ambiguity stemming from insufficient post-conditions. We successfully rectified this in eight cases by strengthening their post-conditions, with the fixes validated through equivalence testing. Although our refinements improved the specifications for two other samples, they still did not pass the equivalence check. We were unable to resolve the ambiguities in the final two samples.

A significant portion of these ambiguous samples were newly discovered. Specifically, eight samples (#579, #602, #625, #629, #733, #755, #793, #807) were not documented in the prior work by Clover (Sun et al., 2024). Of these, we successfully fixed five (#625, #733, #755, #793, #807) and refined one (#602). Corresponding code examples are shown in Figures 25-32.

Regarding the issues previously reported by Clover, our findings for samples #567, #576, #644, and #803 largely concur. We fixed three (#567, #644, #803) and refined one (#576), with details in Figures 33-35. Conversely, sample #472 passed our equivalence checks, which aligns with Clover’s classification of its issue as a natural-language inconsistency rather than a specification defect. As noted, sample #632 was excluded due to a timeout.

```

1944
1945
1946
1947
1948
1949
1950
1951 predicate InArray(a: array<int>, x: int)
1952     reads a
1953 {
1954     exists i :: 0 <= i < a.Length && a[i] == x
1955 }
1956
1957 method DissimilarElements(a: array<int>, b: array<int>) returns (result:
1958     seq<int>)
1959     ensures forall x :: x in result ==> (InArray(a, x) != InArray(b, x))
1960     ensures forall i, j :: 0 <= i < j < |result| ==> result[i] != result[
1961         j]
1962     ##### The post-conditions here omit that the first half of result
1963     is in array a, while the second half is in b.
1964 {
1965     var res: seq<int> := [];
1966     for i := 0 to a.Length
1967         invariant 0 <= i <= a.Length
1968         invariant forall x :: x in res ==> InArray(a, x)
1969         invariant forall x :: x in res ==> InArray(a, x) != InArray(b, x)
1970         invariant forall i, j :: 0 <= i < j < |res| ==> res[i] != res[j]
1971     {
1972         if !InArray(b, a[i]) && a[i] !in res
1973         {
1974             res := res + [a[i]];
1975         }
1976     }
1977
1978     ghost var partialSize := |res|;
1979     for i := 0 to b.Length
1980         invariant 0 <= i <= b.Length
1981         invariant forall k :: partialSize <= k < |res| ==> InArray(b, res
1982             [k])
1983         invariant forall k :: 0 <= k < |res| ==> InArray(a, res[k]) !=
1984             InArray(b, res[k])
1985         invariant forall i, j :: 0 <= i < j < |res| ==> res[i] != res[j]
1986     {
1987         if !InArray(a, b[i]) && b[i] !in res
1988         {
1989             res := res + [b[i]];
1990         }
1991     }
1992
1993     result := res;
1994 }
1995
1996
1997

```

Figure 25: An unresolved specification ambiguity in DafnySynthesis sample #579. The post-condition is insufficient because it fails to enforce the preservation of the relative order of elements from the input array in the output.

```

1998
1999
2000
2001
2002
2003 method FindFirstRepeatedChar(s: string) returns (found: bool, c: char)
2004     ensures found ==> exists i, j :: 0 <= i < j < |s| && s[i] == s[j]
2005         && s[i] == c && (forall k, l :: 0 <= k < l < j
2006             && s[k] == s[l] ==> k >= i)
2007             ensures !found ==> (forall i, j :: 0 <= i < j < |s| ==> s[i] != s[j])
2008             ##### ↓ The added post-condition
2009             ensures !found ==> c == ' '
2010             ##### ↑
2011 {
2012     c := ' ';
2013     found := false;
2014     var inner_found := false;
2015     var i := 0;
2016     while i < |s| && !found
2017         invariant 0 <= i <= |s|
2018         invariant found == inner_found
2019
2020         invariant found ==> exists ii, jj :: 0 <= ii < i
2021             && ii < jj < |s| && s[ii] == s[jj] && s[ii] == c
2022                 && (forall k, l :: 0 <= k < l < jj && s[k] == s[l] ==> k >= ii)
2023
2024             invariant !found <=> (forall ii, jj :: 0 <= ii < i
2025                 && ii < jj < |s| ==> s[ii] != s[jj])
2026             ##### ↓ The added loop invariant
2027             invariant !found ==> c == ' '
2028             ##### ↑
2029     {
2030         var j := i + 1;
2031         while j < |s| && !inner_found
2032             invariant i < j <= |s|
2033             invariant inner_found ==> exists k :: i < k < |s|
2034                 && s[i] == s[k] && s[i] == c
2035                 invariant !inner_found
2036                     <=> (forall k :: i < k < j ==> s[i] != s[k])
2037                     ##### ↓ The added loop invariant
2038                     invariant !inner_found ==> c == ' '
2039                     invariant !found
2040                     ##### ↑
2041             {
2042                 if s[i] == s[j] {
2043                     inner_found := true;
2044                     c := s[i];
2045                 }
2046                 j := j + 1;
2047             }
2048             found := inner_found;
2049             i := i + 1;
2050         }
2051     }

```

Figure 26: A refined but unfixed specification for sample #602. While the shown refinement fails the equivalence test, a stricter post-condition ($k > i$) could not be verified due to a timeout.

```

2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071 method SwapFirstAndLast(a: array<int>)
2072   requires a.Length > 0
2073   modifies a
2074   ##### ↓ The added post-condition
2075   ensures a.Length == old(a.Length)
2076   ##### ↑
2077   ensures a[0] == old(a[a.Length - 1])
2078   ensures a[a.Length - 1] == old(a[0])
2079   ensures forall k :: 1 <= k < a.Length - 1 ==> a[k] == old(a[k])
2080   {
2081     var tmp := a[0];
2082     a[0] := a[a.Length - 1];
2083     a[a.Length - 1] := tmp;
2084   }

```

Figure 27: A successfully resolved specification ambiguity in DafnySynthesis sample #625. The original specification was ambiguous as it lacked a constraint on the output array’s length. The ambiguity was rectified by introducing a post-condition ensuring the length remains invariant.

```

2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105

```

```

2106
2107
2108
2109
2110
2111
2112
2113
2114 predicate IsEven(n: int)
2115 {
2116     n % 2 == 0
2117 }
2118
2119 method FindEvenNumbers(arr: array<int>) returns (evenList: seq<int>)
2120
2121     ensures forall i :: 0 <= i < |evenList| ==> IsEven(evenList[i])
2122     && evenList[i] in arr[..]
2123     ensures forall i :: 0 <= i < arr.Length && IsEven(arr[i])
2124     ==> arr[i] in evenList
2125     ##### The post-conditions here do not ensures the order preserving
2126     between the input array and output array
2127 {
2128     evenList := [];
2129     for i := 0 to arr.Length
2130         invariant 0 <= i <= arr.Length
2131         invariant 0 <= |evenList| <= i
2132         invariant forall k :: 0 <= k < |evenList| ==> IsEven(evenList[k])
2133         && evenList[k] in arr[..]
2134         invariant forall k :: 0 <= k < i && IsEven(arr[k]) ==> arr[k] in
2135             evenList
2136         {
2137             if IsEven(arr[i])
2138             {
2139                 evenList := evenList + [arr[i]];
2140             }
2141         }
2142     method FindEvenNumbers_check(arr: array<int>) returns (evenList: seq<int>
2143         >)
2144     {
2145         evenList := *;
2146         assume forall i :: 0 <= i < |evenList| ==> IsEven(evenList[i]) &&
2147             evenList[i] in arr[..];
2148         assume forall i :: 0 <= i < arr.Length && IsEven(arr[i]) ==> arr[i] in
2149             evenList;
2150         var val_0 := FindEvenNumbers(arr);
2151         assert evenList[..] == val_0[..];
2152     }
2153
2154
2155
2156
2157
2158
2159

```

Figure 28: An unresolved specification ambiguity in DafnySynthesis sample #629. The post-condition is insufficient because it fails to enforce the preservation of the relative order of elements from the input array.

```

2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170 method FindFirstOccurrence(arr: array<int>, target: int) returns (index: int)
2171     requires arr != null
2172     requires forall i, j :: 0 <= i < j < arr.Length ==> arr[i] <= arr[j]
2173     ensures 0 <= index < arr.Length ==> arr[index] == target
2174     ensures index == -1 ==> forall i :: 0 <= i < arr.Length ==> arr[i] != target
2175     ensures forall i :: 0 <= i < arr.Length ==> arr[i] == old(arr[i])
2176     ##### ↴ The added post-condition
2177     ensures 0 <= index < arr.Length || index == -1
2178     ensures 0 <= index < arr.Length ==> ((forall i :: 0 <= i < index ==> arr[i] < arr[index]) && (forall j :: index <= j < arr.Length ==> arr[j] >= arr[index]))
2179     #####
2180     {
2181     index := -1;
2182     for i := 0 to arr.Length
2183         invariant 0 <= i <= arr.Length
2184         invariant index == -1 ==> forall k :: 0 <= k < i ==> arr[k] != target
2185         invariant 0 <= index < i ==> arr[index] == target
2186         invariant forall k :: 0 <= k < arr.Length ==> arr[k] == old(arr[k])
2187         [
2188         {
2189             if arr[i] == target
2190             {
2191                 index := i;
2192                 break;
2193             }
2194             if arr[i] > target
2195             {
2196                 break;
2197             }
2198         }
2199     }
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213

```

Figure 29: A successfully resolved specification ambiguity in DafnySynthesis sample #733. The original specification was insufficient, lacking detail for cases where the input index is non-negative. The issue was fixed by refining the post-condition to explicitly define the expected behavior for this scenario.

```

2214     function MinPair(s: seq<int>) : (r: int)
2215         requires |s| == 2
2216         ensures s[0] <= s[1] <=> r == s[0]
2217         ensures s[0] > s[1] ==> r == s[1]
2218     {
2219         if s[0] <= s[1] then s[0] else s[1]
2220     }
2221     function min(s: seq<int>) : (r: int)
2222         requires |s| >= 2
2223         ensures forall i :: 0 <= i < |s| ==> r <= s[i]
2224     {
2225         if |s| == 2 then MinPair(s)
2226         else MinPair([s[0], min(s[1..])])
2227     }
2228     method SecondSmallest(s: array<int>) returns (secondSmallest: int)
2229         requires s.Length >= 2
2230         requires exists i, j :: 0 <= i < s.Length && 0 <= j < s.Length
2231             && i != j && s[i] == min(s[..]) && s[j] != s[i]
2232         ensures exists i, j :: 0 <= i < s.Length && 0 <= j < s.Length
2233             && i != j && s[i] == min(s[..]) && s[j] == secondSmallest
2234         ensures forall k :: 0 <= k < s.Length && s[k] != min(s[..])
2235             ==> s[k] >= secondSmallest
2236         ##### ↓ The added post-condition
2237         ensures (exists i, j :: i != j && 0 <= i < s.Length
2238             && 0 <= j < s.Length && s[i] == s[j] && s[i] == min(s[..]))
2239             ==> secondSmallest == min(s[..])
2240         ensures !(exists i, j :: i != j && 0 <= i < s.Length
2241             && 0 <= j < s.Length && s[i] == s[j] && s[i] == min(s[..]))
2242             ==> ( (exists k :: 0 <= k < s.Length && s[k] == secondSmallest)
2243                 && (forall k :: 0 <= k < s.Length && s[k] > min(s[..])
2244                     ==> s[k] >= secondSmallest) && secondSmallest > min(s[..]) )
2245         #####
2246     {
2247         var minIndex := 0;
2248         var secondMinIndex := 1;
2249         if s[1] < s[0] {
2250             minIndex := 1;
2251             secondMinIndex := 0;
2252         }
2253         for i := 2 to s.Length
2254             invariant 0 <= i <= s.Length
2255             invariant 0 <= minIndex < i
2256             invariant 0 <= secondMinIndex < i
2257             invariant minIndex != secondMinIndex
2258             invariant forall k :: 0 <= k < i ==> s[k] >= s[minIndex]
2259             invariant forall k :: 0 <= k < i && k != minIndex ==> s[k] >= s[
2260                 secondMinIndex]
2261         {
2262             if s[i] < s[minIndex] {
2263                 secondMinIndex := minIndex;
2264                 minIndex := i;
2265             } else if s[i] < s[secondMinIndex] {
2266                 secondMinIndex := i;
2267             }
2268         }
2269         secondSmallest := s[secondMinIndex];
2270     }

```

Figure 30: A successfully resolved specification ambiguity in DafnySynthesis sample #755. The original specification was insufficient, failing to distinguish between cases with a unique minimum value and those with multiple occurrences of the minimum. The ambiguity was rectified by refining the post-condition to explicitly detail the expected behavior for both scenarios.

```

2268
2269
2270
2271
2272
2273
2274
2275
2276
2277 method LastPosition(arr: array<int>, elem: int) returns (pos: int)
2278     requires arr.Length > 0
2279     requires forall i, j :: 0 <= i < j < arr.Length ==> arr[i] <= arr[j]
2280     ##### ↓ Original post-condition
2281     // ensures pos == -1 || (0 <= pos < arr.Length && arr[pos] == elem &&
2282     // (pos <= arr.Length - 1 || arr[pos + 1] > elem))
2283     ##### ↑
2284     ##### ↓ The fixed post-condition
2285     ensures pos == -1 <=> (forall j :: 0 <= j < arr.Length ==> arr[j] != elem)
2286     ensures pos != -1 <=> (0 <= pos < arr.Length && arr[pos] == elem &&
2287     // (pos == arr.Length - 1 || arr[pos + 1] > elem))
2288     ##### ↑
2289     ensures forall i :: 0 <= i < arr.Length ==> arr[i] == old(arr[i])
2290 {
2291     pos := -1;
2292     for i := 0 to arr.Length ##### Originally, the upper bound is arr.
2293     // Length - 1, but it was buggy
2294     invariant 0 <= i <= arr.Length
2295     ##### ↓ Original loop invariant
2296     // invariant pos == -1 || (0 <= pos < i && arr[pos] == elem &&
2297     // pos == i - 1 || arr[pos + 1] > elem)
2298     ##### ↓ The fixed loop invariant
2299     invariant pos == -1 <=> (forall j :: 0 <= j < i ==> arr[j] != elem)
2300     invariant pos != -1 <=> (0 <= pos < i && arr[pos] == elem &&
2301     // pos == i - 1 || arr[pos + 1] > elem)
2302     ##### ↑
2303     invariant forall k :: 0 <= k < arr.Length ==> arr[k] == old(arr[k])
2304     [
2305     if arr[i] == elem
2306     {
2307         pos := i;
2308     }
2309 }

```

Figure 31: A successfully resolved specification ambiguity in DafnySynthesis sample #793. The original specification was insufficient as it failed to define distinct behaviors based on the sign of the input parameter ‘pos’. The ambiguity was rectified by refining the post-condition to explicitly handle the cases where ‘pos’ is negative and non-negative, respectively.

```

2314
2315
2316
2317
2318
2319
2320
2321

```

```

2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334 predicate IsOdd(x: int)
2335 {
2336     x % 2 != 0
2337 }
2338
2339 method FindFirstOdd(a: array<int>) returns (found: bool, index: int)
2340     requires a != null
2341     ensures !found ==> forall i :: 0 <= i < a.Length ==> !IsOdd(a[i])
2342     ensures found ==> 0 <= index < a.Length && IsOdd(a[index])
2343         && forall i :: 0 <= i < index ==> !IsOdd(a[i])
2344     ##### ↓ The added post-condition
2345     ensures !found ==> index == a.Length
2346     ##### ↑
2347 {
2348     found := false;
2349     index := 0;
2350     while (index < a.Length)
2351         invariant 0 <= index <= a.Length
2352         invariant !found ==> forall i :: 0 <= i < index ==> !IsOdd(a[i])
2353         invariant found ==> IsOdd(a[index - 1]) && forall i :: 0 <= i <
2354             index - 1 ==> !IsOdd(a[i])
2355     {
2356         if IsOdd(a[index])
2357         {
2358             found := true;
2359             return;
2360         }
2361         index := index + 1;
2362     }
2363 }
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375

```

Figure 32: A successfully resolved specification ambiguity in DafnySynthesis sample #807. The original specification was insufficient, as it only described the behavior for successful outcomes. The ambiguity was resolved by strengthening the post-condition to explicitly define the program’s state in failure cases, ensuring comprehensive and predictable behavior.

```

2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386 method IsSorted(a: array<int>) returns (sorted: bool)
2387     requires a.Length > 0
2388     ##### ↓ Original post-condition
2389     // ensures sorted <= forall i, j :: 0 <= i < j < a.Length
2390     // ==> a[i] <= a[j]
2391     // ensures !sorted ==> exists i, j :: 0 <= i < j < a.Length
2392     // && a[i] > a[j]
2393     ##### ↑
2394     ##### ↓ The fixed post-condition
2395     ensures sorted <=> forall i, j :: 0 <= i < j < a.Length
2396     // ==> a[i] <= a[j]
2397     ##### ↑
2398 {
2399     sorted := true;
2400     for i := 0 to a.Length - 1
2401         invariant 0 <= i < a.Length
2402         ##### ↓ Original loop invariant
2403         // invariant sorted <= forall k, l :: 0 <= k < l < i
2404         // ==> a[k] <= a[l]
2405         // invariant !sorted ==> exists k :: 0 <= k < i && a[k] > a[k+1]
2406         ##### ↑
2407         ##### ↓ The fixed post-condition
2408         invariant sorted <=> forall k, l :: 0 <= k < l <= i
2409         // ==> a[k] <= a[l]
2410         ##### ↑
2411     {
2412         if a[i] > a[i + 1]
2413         {
2414             sorted := false;
2415             break;
2416         }
2417     }
2418     sorted := sorted;
2419 }

```

Figure 33: A successfully resolved specification ambiguity in DafnySynthesis sample #567, an issue also identified by the Clover. The original post-condition was overly permissive, stating only a sufficient condition for the desired outcome. The ambiguity was rectified by strengthening this to a necessary and sufficient condition (an equivalence).

```

2421
2422
2423
2424
2425
2426
2427
2428
2429

```

```

2430
2431
2432
2433
2434
2435
2436
2437 method Reverse(a: array<int>)
2438     modifies a
2439     ##### ↓ The added post-condition
2440     ensures a.Length == old(a.Length)
2441     ##### ↑
2442     ensures forall k :: 0 <= k < a.Length ==> a[k] == old(a[(a.Length-1) -
2443         k])
2444 {
2445     var l := a.Length - 1;
2446     var i := 0;
2447     while (i < l-i)
2448         invariant 0 <= i <= (l+1)/2
2449         invariant forall k :: 0 <= k < i || l-i < k <= l ==> a[k] == old(a[l-
2450             k])
2451         invariant forall k :: i <= k <= l-i ==> a[k] == old(a[k])
2452     {
2453         a[i], a[l-i] := a[l-i], a[i];
2454         i := i + 1;
2455     }
2456 }
2457 method ReverseUptoK(s: array<int>, k: int)
2458     modifies s
2459     requires 2 <= k <= s.Length
2460     ##### ↓ The added post-condition
2461     ensures s.Length == old(s.Length)
2462     ##### ↑
2463     ensures forall i :: 0 <= i < k ==> s[i] == old(s[k - 1 - i])
2464     ensures forall i :: k <= i < s.Length ==> s[i] == old(s[i])
2465 {
2466     var l := k - 1;
2467     var i := 0;
2468     while (i < l-i)
2469         invariant 0 <= i <= (l+1)/2;
2470         invariant forall p :: 0 <= p < i || l-i < p <= l ==> s[p] == old(s[l-
2471             p]);
2472         invariant forall p :: i <= p <= l-i ==> s[p] == old(s[p]);
2473         invariant forall p :: k <= p < s.Length ==> s[p] == old(s[p])
2474     {
2475         s[i], s[l-i] := s[l-i], s[i];
2476         i := i + 1;
2477     }
2478 }
```

Figure 34: A successfully resolved specification ambiguity in DafnySynthesis sample #644, an issue also identified by the Clover. The original specification was ambiguous as it lacked a constraint on the output array's length. The ambiguity was rectified by introducing a post-condition ensuring the length remains invariant.

```

2479
2480
2481
2482
2483
```

```

2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500 method IsPerfectSquare(n: int) returns (result: bool)
2501     requires n  $\geq 0$ 
2502     #####  $\Downarrow$  Original post-condition
2503     // ensures result == true ==> (exists i: int ::  $0 \leq i \leq n \ \&\& i * i == n$ )
2504     // ensures result == false ==> (forall a: int ::  $0 < a * a < n \Rightarrow a * a \neq n$ )
2505
2506     #####  $\Uparrow$ 
2507     #####  $\Downarrow$  The fixed post-condition
2508     ensures result  $\Leftrightarrow$  (exists i: int ::  $0 \leq i \leq n \ \&\& i * i == n$ )
2509     #####  $\Uparrow$ 
2510 {
2511     var i := 0;
2512     while (i * i < n)
2513         invariant  $0 \leq i \leq n$ 
2514         invariant forall k ::  $0 \leq k < i \Rightarrow k * k < n$ 
2515     {
2516         i := i + 1;
2517     }
2518     return i * i == n;
2519 }

```

Figure 35: A successfully resolved specification ambiguity in DafnySynthesis sample #803, an issue also identified by the Clover. The original post-condition was overly permissive, stating only necessary conditions for the desired outcome. The ambiguity was rectified by strengthening this to a necessary and sufficient condition (an equivalence).

```

2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537

```

```

2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549 method IsSublist(sub: seq<int>, main: seq<int>) returns (result: bool)
2550     ##### ↓ Original post-condition
2551     // ensures true <= (exists i :: 0 <= i <= |main| - |sub| && sub ==
2552         main[i..i + |sub|])
2553     ##### ↑
2554     ##### ↓ The refined post-condition
2555     ensures result ==> (exists i :: 0 <= i <= |main| - |sub| && sub ==
2556         main[i..i + |sub|])
2557     ensures result ==> (exists i :: |sub| <= i <= |main| && sub == main[i
2558         - |sub|..i])
2559     #####
2560 {
2561     if |sub| > |main| {
2562         return false;
2563     }
2564     result := false;
2565     for i := 0 to |main| - |sub| + 1
2566         ##### ↓ The original loop invariant
2567         // invariant result ==> (exists j :: 0 <= j < i && sub == main[j
2568             ..j + |sub|])
2569         #####
2570         ##### ↓ The refined loop invariant
2571         invariant 0 <= i <= |main| - |sub| + 1
2572         #####
2573     {
2574         if sub == main[i..i + |sub|] {
2575             result := true;
2576         }
2577     }
2578     result := false;
2579 }
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591

```

Figure 36: An unresolved specification ambiguity in DafnySynthesis sample #576, an issue also identified by the Clover. The original post-condition was effectively meaningless, providing no meaningful constraints. Although the post-condition was refined to be more specific, the resulting specification still fails to pass the equivalence test, indicating that the ambiguity has not been fully resolved and requires further investigation.

2592 H THE USE OF LARGE LANGUAGE MODELS
25932594 Multiple LLM products, including GPT-5 and Gemini-2.5-pro, are deployed to polish the
2595 writing. However, none of the paragraphs is written by LLMs directly, and all research
2596 ideas are independently proposed by authors without any AI assistance. Claude-Opus-4.1
2597 and Sonnet are used to create figure generation code for Figure 5 and 24. Cursor is included
2598 to assist coding, but all generated code is then carefully inspected by authors. Other uses
2599 of LLMs in data curation and synthesis are clearly stated in the paper.
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645