LOCAL SUCCESS DOES NOT COMPOSE: BENCHMARKING LARGE LANGUAGE MODELS FOR COMPOSITIONAL FORMAL VERIFICATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Despite rapid advances in code generation, current Large Language Models (LLMs) still lack an essential capability for reliable and verifiable code generation: compositional reasoning across multi-function programs. To explore this potential and important gap, we introduce DAFNYCOMP, a benchmark designed to systematically evaluate LLMs on the generation of compositional specifications in Dafny. Unlike prior benchmarks that primarily target single-function annotation, DAFNYCOMP focuses on programs composed of multiple interacting functions with necessary data dependencies, requiring LLMs to produce specifications that ensure correctness across component boundaries. Our benchmark comprises 300 automatically synthesized programs, each carefully constructed by combining 2–5 originally independent functions in a chain-based manner through LLM-driven synthesis. We evaluate LLMs from five leading research groups that represent the current frontier of reasoning-centric AI, including the GPT, CLAUDE, GEMINI, DEEPSEEK, and QWEN families. Our results reveal a striking dichotomy: while LLMs achieve both high syntax correctness (>99%) and moderate verification rates (>58%) in prior single-function benchmarks, they exhibit degraded syntax correctness (95.67%) and a catastrophic verification failure (3.69%) in DAFNYCOMP's compositional tasks—a 92% performance gap. Even the most powerful LLM achieves only 7% verification at Pass@8, with most LLMs below 2%. Further analysis reveals that LLMs systematically fail at cross-functional reasoning through three primary failure modes: *specification fragility* (39.2%), *implementation-proof* misalignment (21.7%), and *reasoning instability* (14.1%). These failures clearly reveal the absence of compositional reasoning capabilities in current LLMs. DAFNYCOMP thus establishes a diagnostic benchmark for tracking progress in verifiable code generation with LLMs, highlighting that the path from local to compositional verification remains largely uncharted.

1 INTRODUCTION

Large language models (LLMs) have transformed software development through their remarkable code generation capabilities, enabling developers to produce complex programs from natural language descriptions (Chen et al., 2021; Austin et al., 2021). These advances have driven widespread adoption of programming assistants and development environments, fundamentally transforming how modern software is developed. As LLM-generated code becomes increasingly integrated into production systems, a critical question emerges: *how to ensure the correctness of automatically synthesized programs*. Unlike human-written code that can be manually reviewed and tested, the scale and complexity of LLM outputs demand systematic approaches to verification that go beyond traditional debugging methods. On the other hand, conventional testing provides only partial confidence and cannot rule out rare corner cases or subtle specification mismatches.

Formal verification provides a principled solution to this challenge by offering mathematical guarantees of program correctness through rigorous specification and proof techniques. Programming languages like Dafny enable developers to express precise contracts—preconditions, postconditions, and invariants—that can be mechanically verified against implementations (Leino, 2010b). However, the adoption of formal verification has historically been constrained by what we often refer to as the "specification bottleneck": writing comprehensive annotations not only demands specialized

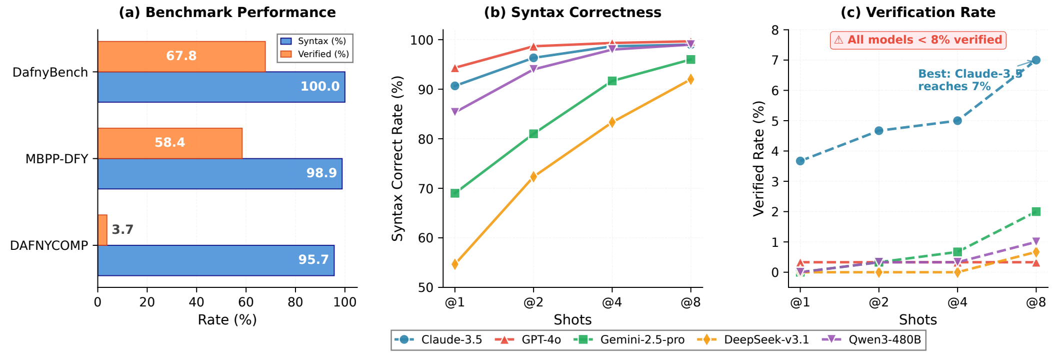


Figure 1: **The formal verification gap: high syntax success versus low verification rates.** (a) Benchmark performance reveals a dramatic gap between syntax correctness and verification success, with DafnyComp showing a 92% drop from 95.67% to 3.69%. (b) All models converge to high syntax correctness at @8 shots, with performance ranging from 92% to 99%. (c) Verification rates remain critically low (<8%) across all models despite increased sampling, with Claude-3.5 achieving the highest rate at only 7%.

expertise but also produces specification code that is comparable in size to the implementation itself (Leino et al., 2017; Loughridge et al., 2024b). Recent research has explored the use of LLMs to automate this specification generation process, demonstrating promising results where models can complete missing annotations for individual functions and achieve moderate verification success rates (Loughridge et al., 2024a; Yan et al., 2025). However, current work in this area often suffers from a critical limitation: existing benchmarks, such as DafnyBench, primarily evaluate annotation completion within isolated functions (Loughridge et al., 2024a), failing to address the compositional reasoning ability required for real-world, sophisticated software systems, where correctness emerges from complex interactions between multiple components (Keysers et al., 2020).

To fill this gap, we introduce DafnyComp, the first benchmark explicitly designed to evaluate the generation of compositional specifications for programming languages equipped with formal verification. Concretely, we make the following contributions:

Contribution 1. To address the limitations of prior verification benchmarks, we present DafnyComp, a new benchmark explicitly designed for compositional formal verification (§3). Unlike existing datasets such as DafnyBench (Loughridge et al., 2024a) that focus on specification generation for isolated single functions, DafnyComp evaluates LLMs on programs composed of multiple interacting functions with real data dependencies. The benchmark consists of 300 Dafny programs synthesized by combining 2–5 independent functions, forcing models to reason across function boundaries to ensure end-to-end correctness. Note that our design is the first to require actual compositional reasoning in specification generation, bridging a critical gap left by prior benchmarks and reflecting the complexities of real-world software systems.

Contribution 2. We comprehensively evaluate 13 state-of-the-art LLMs on the constructed DafnyComp (§4) benchmark, including advanced models like GPT-4o, Claude 3.5, Gemini 2.5, DeepSeek-v3.1, and Qwen3-coder. The results reveal a dramatic collapse in verification performance despite high syntactic accuracy: while the models produce syntactically correct code for approximately 95.7% of the tasks, only 3.7% of their outputs actually pass the formal verifier. This staggering 92% gap between syntax success and semantic correctness persists across all model families, prompt settings, and sampling strategies. Even with up to 8 attempts per problem, the best model attains only around 7% verification success, indicating that increasing sampling or context does not remedy the fundamental limitation.

Contribution 3. We carefully analyze the failure cases in the benchmark, which pinpoints three primary failure modes underlying this breakdown (§5), highlighting systemic obstacles to compositional reasoning in current LLMs:

- (**i**) *Specification fragility*: we observe the brittleness of generated specifications wherein small omissions, over-/under-strengthening, or inconsistent framing (`reads`/`modifies`) clauses can invalidate downstream proofs. Concretely, in compositional settings, a missing or slightly weaker postcondition at one stage can fail to imply a callee's precondition, triggering a domino effect along the call chain even when each component appears locally reasonable.

- (**ii**) *Implementation–proof misalignment*: We identify inconsistencies between the produced code and its associated specifications or proofs, indicating that models often generate implementations and annotations via largely independent pathways. Typical symptoms include plausible-but-false loop invariants, contradictory `requires`/`ensures` obligations, or termination metrics that do not match control flow, any of which cause verification to fail despite syntactic well-formedness, and in composed programs they immediately block end-to-end proofs because component contracts can no longer be trusted as summaries.
- (**iii**) *Reasoning instability*: we witness a tendency to lose the inductive thread of the argument over multiple steps, leading to invariants that are not preserved, incomplete coverage of cases, or missing well-founded `decreases` measures. These errors are most evident in composition, where maintaining complex state relationships across iterations and function boundaries is crucial for end-to-end correctness.

These failure modes were pervasive in the models' outputs, revealing a fundamental absence of robust compositional reasoning capabilities. By identifying these issues, DAFNYCOMP serves as a diagnostic benchmark for the community, enabling systematic measurement of progress toward LLMs that can verify complex multi-component programs.

2 RELATED WORK

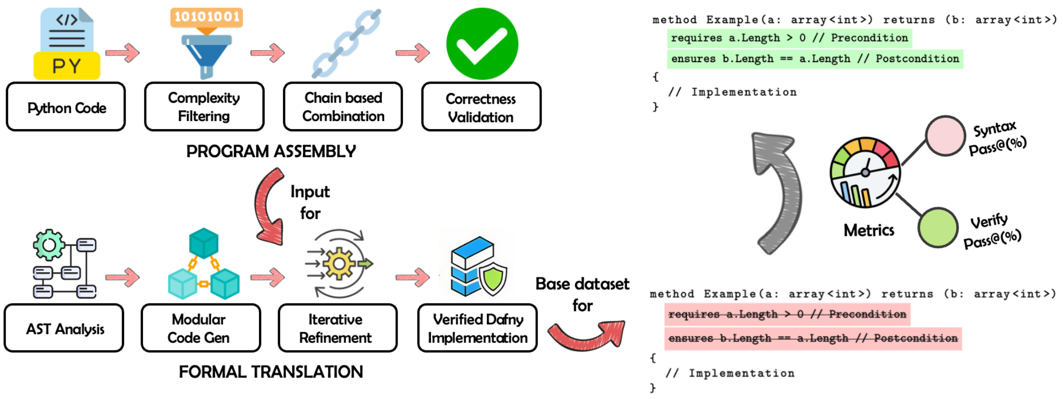
Formal Verification Benchmarks. Existing benchmarks for verifiable code generation can be categorized into two types. Single-function benchmarks, such as DAFNYBENCH (Loughridge et al., 2024a) and MBPP-DFY (Misu et al., 2024), evaluate annotation completion within isolated methods, achieving moderate success rates (50-60%) but failing to capture inter-function dependencies. Interactive theorem proving benchmarks (miniCodeProps (Lohn & Welleck, 2024), FVAPPS (Dougherty & Mehta, 2025)) target proof synthesis in systems like Lean (De Moura et al., 2015) but require extensive manual validation and remain disconnected from practical programming. DAFNYCOMP bridges this gap by evaluating compositional specification generation—a prerequisite for scaling verification beyond toy programs to production systems. Unlike prior work, we explicitly construct multi-function programs with data dependencies, exposing the compositional reasoning deficit in current models (see Appendix C for detailed comparison).

Dynamic Benchmark Generation. Static benchmarks suffer from contamination and overfitting (Hu et al., 2025; Zhang et al., 2024). Dynamic generation techniques—creating new tasks, transforming problems, or perturbing reasoning structures (Zhu et al., 2024; 2023)—show promise in mathematics and logic but neglect formal verification's unique demands: specifications must be syntactically valid, semantically precise, and correct across all execution paths. Our synthesis pipeline addresses this by generating verifiable multi-function Dafny programs through controlled composition, ensuring both novelty and correctness while maintaining the semantic complexity that exposes compositional reasoning gaps.

Compositional Reasoning in LLMs. The ability to systematically combine simpler units into correct larger structures remains a frontier challenge (Li et al., 2024; Dziri et al., 2023). While progress exists in natural language and symbolic domains, formal verification imposes stricter demands: specifications must preserve invariants across components and ensure global correctness. Existing training paradigms favor pattern matching over principled proof construction. By requiring models to generate specifications bridge function boundaries with explicit data dependencies, DAFNYCOMP provides the first diagnostic benchmark for compositional reasoning in formal verification.

3 BENCHMARK CONSTRUCTION

DAFNYCOMP synthesizes 300 verified multi-function programs through a two-stage pipeline (Figure 2), including program assembly (§3.1) and formal translation (§3.2), which bridges the gap between practical Python implementations and verification-oriented Dafny specifications. Within this pipeline, program assembly ensures the construction of compositional Python programs with functional correctness, while specification translation with refinement ensures the quality and reliability of the resulting data. We also provide format details of evaluation tasks (§3.3) and the key characteristics of the benchmark (§3.4).



```
method Example(a: array<int>) returns (b: array<int>)
   requires a.Length > 0 // Precondition
   ensures b.Length == a.Length // Postcondition
{
  // Implementation
}
```

```
method Example(a: array<int>) returns (b: array<int>)
   requires a.Length > 0 // Precondition
   ensures b.Length == a.Length // Postcondition
{
  // Implementation
}
```

Figure 2: Two-stage benchmark synthesis: (1) Assembly combines independent Python functions with controlled data flow, ensuring algorithmic complexity while maintaining tractability; (2) Formal translation converts to verified Dafny through incremental AST-guided transformation.

3.1 PROGRAM ASSEMBLY

We construct compositional programs by systematically combining functions from LEETCODEDATASET (Xia et al., 2025), selected for their algorithmic depth and verification challenges.

Function Selection. We filter the corpus using McCabe's cyclomatic complexity (McCabe, 1976) as a proxy for verification difficulty, retaining only functions with complexity >5 (around top 30% of the dataset) and at least 10 lines of code. This threshold ensures non-trivial control flow—loops with complex termination conditions, nested conditionals, recursive patterns—that stress specification generation. For tractability, we restrict to single-input/single-output functions, yielding 1,847 candidate functions.

Compositional Strategy. Following Hu et al. (2025), we employ chain-based composition where each function's output feeds the next's input, creating explicit data dependencies. While more complex call graphs (trees, DAGs) are theoretically richer, empirical trials showed synthesis success drops from 47% (chains) to <8% (arbitrary graphs) while providing no additional diagnostic value—the chains suffice to expose compositional failures. After composition, we further identify the minimal set of shared import dependencies across the combined Python functions. This step is essential because the original LEETCODEDATASET often relies on broad `import *` statements, which obscure library ownership and names. Without explicit mappings, Dafny cannot interpret external libraries, preventing the synthesis of intermediate functions to replace missing third-party features. We generate programs with 2–5 functions, exploring multiple permutations since function ordering affects both data flow and verification complexity.

Validation Pipeline. After composition, the resulting Python code is subjected to a three-stage validation pipeline, which filters candidates before their use in Section 3.2.

- (i) *Type checking via constraint propagation*: We statically infer candidate types and shapes for each function's inputs and outputs and propagate these constraints along the composition chain. This pass rejects compositions with incompatible interfaces (e.g., scalar-sequence or element-type mismatches) and flags violations of simple value constraints inferred from guards (such as non-negativity or length bounds). The result is a set of compositions whose interfaces are consistent end-to-end, providing a reliable basis for subsequent translation and verification.

- (ii) *Formatting standardization*: We apply a deterministic rewriter, implemented with tools such as `Black` and `isort`, to normalize code style, including indentation, whitespace, line breaking, and import organization. Canonicalizing these incidental variations yields stable, diff-friendly artifacts and reduces prompt variance in later stages. This step preserves semantics while producing uniform program layouts that are easier to parse, translate, and verify.

- (iii) *Test validation*: We execute each composed program against the reference unit tests from LEETCODEDATASET to confirm functional correctness and basic executability. We construct unit tests for composed functions by intersecting the input–output constraints of their constituent functions. If the intersection is empty, the sample is discarded. Programs that raise runtime exceptions, fail assertions, or produce incorrect outputs are discarded, ensuring only behaviorally sound compositions advance. This filtering isolates verification challenges to specification and reasoning rather than implementation errors during the Dafny translation stage.

Following this procedure, we obtain 1,200 valid Python programs with 2–5 functions, which will be used in the next stage.

3.2 FORMAL TRANSLATION

We translate validated Python compositions into Dafny implementations with formal guarantees, focusing on the verification-oriented aspects of the benchmark here.

Translation Challenges. Direct end-to-end translation from Python to Dafny proved largely ineffective, with empirical success rates below 5%. The core difficulty lies in Dafny's demand for explicit specifications, invariants, and termination arguments—semantic elements absent in Python. This semantic gap makes single-pass translation infeasible for non-trivial programs.

Incremental Pipeline. Inspired by Wen et al. (2024), we adopt an incremental approach: the abstract syntax tree (AST) of each Python program is decomposed into function- or control-structure–level fragments. Each fragment is translated into Dafny and immediately verified, localizing errors to the smallest possible unit. Verified fragments are then progressively reassembled according to the AST hierarchy, culminating in a complete Dafny program. Importantly, although translation proceeds incrementally, the Python program must be composed in its entirety before it can be executed. Whole-program composition in Python provides two benefits: (**i**) Python's explicit AST nodes and mature tooling make program assembly more reliable and transparent; and (**ii**) having a coherent Python blueprint ensures that the incremental Dafny translation preserves global logical relationships, rather than producing isolated fragments that fail to compose. Thus, whole-program composition and incremental translation are complementary design choices. To further improve reliability, each candidate Dafny program undergoes up to ten refinement iterations, where specifications are strengthened in response to verifier feedback (e.g., adding loop invariants, refining postconditions, or inserting assertions). We select CLAUDE-4-SONNET-20250514 for the entire synthesis and refinement process; see Appendix D for success rate comparisons and Appendix E for the exact prompts.

In total, the pipeline ultimately yields 564 verified Dafny programs from 1,200 attempts (corresponding to an overall 47% success rate). Translation synthesis errors primarily arise from incomplete specifications (31%), type inference errors (22%), timeouts (18%), and irreconcilable semantic gaps (29%). From these, we retain 300 programs carefully balanced across complexity levels (100 each with 2–3, 3–4, and 4–5 functions). To ensure evaluation integrity, we conduct a thorough contamination analysis against MBPP-DFY (Misu et al., 2024), which is similarly synthesized from Python code as a Dafny benchmark dataset. The results in Appendix F provide strong evidence that our test set is indeed free from bias due to data overlap.

3.3 EVALUATION TASK FORMAT

We adopt a specification reconstruction task inspired by Loughridge et al. (2024a). Still, with a crucial difference: rather than removing all `assert` and `invariant` statements, we strip away the contract clauses (`requires`, `ensures`, `reads`, `modifies`, `decreases`) that appear before the opening brace of each `method` or `function`. LLMs to be evaluated are then required to regenerate these specifications to enable verification. This design isolates the challenge of reconstructing cross-function contracts from implementation concerns, focusing evaluation on whether models can generate specifications that capture emergent correctness properties across component boundaries. Unlike annotation completion tasks that permit purely local reasoning, our multi-function programs require understanding how data flows and invariants propagate through compositions. We employ a unified prompt together with standardized compute settings across all evaluations (see Appendix G and Appendix H).

3.4 BENCHMARK STATISTICAL SUMMARY

The resulting benchmark comprises 300 mechanically verified Dafny test cases that jointly capture three key dimensions: compositional complexity from function-to-function call dependencies, algorithmic diversity across multiple categories, and verification challenges arising from the increased specification burden.

Compositional Complexity. Each program contains 2–5 functions (mean = 3.2) with an average of 8.4 cross-function data dependencies, requiring models to reason about specification alignment across component boundaries. Unlike single-function benchmarks where specification generation is largely local, our programs demand that preconditions of called functions be implied by postconditions of their callers—a requirement that introduces cascading verification challenges when specifications fail to propagate correctly.

Algorithmic Diversity. The benchmark spans 15 algorithmic categories with balanced representation: dynamic programming (18%), string manipulation (20%), number theory (15%), and graph algorithms (12%) constitute the primary categories, with the remainder distributed across sorting, searching, and combinatorial problems. Beyond the balance of individual categories, diverse permutations and combinations of these types yield composed programs with more intricate, layered structures. Consequently, the target of composing function calls is reinforced by the characteristics of the source dataset (LEETCODEDATASET (Xia et al., 2025)), which in turn induces composition at the level of algorithmic logic. This design ensures models must develop general compositional reasoning rather than memorizing category-specific patterns.

Verification Challenges. Every program is mechanically verified by Dafny 4.10.0, thereby providing ground-truth correctness. The median program requires 7 loop invariants and 4 assertions for verification, with 23% demanding intricate termination arguments via `decreases` clauses—a 3.5× increase in annotation density compared to DAFNYBENCH's average of 2 per program. This added specification burden reflects the extra complexity of compositional verification, creating a graduated challenge that pinpoints where current models shift from local reasoning to compositional failure.

4 EXPERIMENTAL SETUP AND RESULTS

In this section, we enumerate the evaluation metrics (§4.1), LLM model selection for the benchmark (§4.2), and the corresponding evaluation results (§4.3).

4.1 METRICS

We evaluate two complementary aspects of model performance:

- **Syntax Correctness**: measures whether generated specifications parse successfully in Dafny. This baseline metric captures models' grasp of the formal language syntax.
- **Verification Rate**: measures the fraction of syntactically correct programs that pass Dafny's verifier—the ultimate test of semantic understanding. This metric is computed only over syntactically valid outputs, as verification requires parseable code.

Following Chen et al. (2021), we report Pass@k for $k \in \{1, 2, 4, 8\}$, measuring the overall probability of successfully solving a problem within k attempts. Pass@1 provides a strict zero-shot baseline of immediate reasoning ability, whereas larger k values further exploit additional test-time compute to improve success on compositional tasks (Snell et al., 2024). In this setting, Pass@8 is particularly informative for clearly distinguishing model robustness and adaptability.

4.2 MODEL SELECTION

We evaluate 13 frontier models spanning five architectural families, chosen for their demonstrated strength in code generation and general reasoning:

- **OpenAI**: GPT-4O (Hurst et al., 2024), GPT-4.1 (OpenAI, 2024), O4-MINI (OpenAI, 2025)
- **Anthropic**: CLAUDE-3.5-SONNET (Anthropic, 2024), CLAUDE-4-SONNET (Anthropic, 2025)
- **Google**: GEMINI-2.5-PRO, GEMINI-2.5-FLASH (Google DeepMind, 2025)
- **DeepSeek**: DEEPSEEK-R1 (Guo et al., 2025), DEEPSEEK-V3 (Liu et al., 2024), DEEPSEEK-V3.1 (DeepSeek-AI, 2025)
- **Alibaba**: QWEN2.5-CODER-32B (Hui et al., 2024), QWEN3-CODER-480B (Qwen Team, 2025a), QWQ-32B (Qwen Team, 2025b)

Table 1: Model performance reveals high syntax mastery but catastrophic verification failure. While syntax correctness reaches 99% with sufficient sampling, verification rates remain below 7% even for the best models, exposing the compositional reasoning gap.

Model	**Syntax Correct Rate (%)**				**Verified Rate (%)**			
	@1	@2	@4	@8	@1	@2	@4	@8
OpenAI Models								
GPT-4O	94.33	98.67	99.33	99.67	0.33	0.33	0.33	0.33
O4-MINI	80.00	92.67	98.00	99.00	0.00	0.00	0.67	0.67
GPT-4.1	59.00	69.67	79.33	86.33	0.00	0.00	0.00	0.00
Anthropic Models								
CLAUDE-3.5-SONNET*	90.67	96.33	98.67	99.00	3.67	4.67	5.00	7.00
CLAUDE-4-SONNET†	95.67	97.33	98.00	98.33	2.33	3.00	3.00	3.33
Google Models								
GEMINI-2.5-FLASH	54.00	64.00	81.00	89.67	0.00	0.00	0.00	0.00
GEMINI-2.5-PRO	69.00	81.00	91.67	96.00	0.00	0.33	0.67	2.00
DeepSeek Models								
DEEPSEEK-R1	85.67	95.33	98.33	99.00	0.33	0.33	0.33	0.33
DEEPSEEK-V3	77.33	88.67	95.33	97.33	0.00	0.00	0.33	0.33
DEEPSEEK-V3.1	54.67	72.33	83.33	92.00	0.00	0.00	0.00	0.67
Alibaba Models								
QWEN3-CODER-480B-A35B-INSTRUCT	85.33	94.00	98.00	99.00	0.00	0.33	0.33	1.00
QWEN2.5-CODER-32B-INSTRUCT	62.00	74.67	85.00	89.00	0.00	0.33	0.33	0.67
QWQ-32B	46.67	61.33	78.00	91.00	0.00	0.00	0.00	0.00

*claude-3.5-sonnet-20241022, †claude-4-sonnet-20250514

4.3 RESULTS AND DISCUSSION

Table 1 presents our main experimental findings. We observe a systematic verification collapse across all evaluated models, with four interesting observations:

Observation 1. *Universal verification failure despite syntactic mastery.* The most striking result is the consistent 92-percentage-point gap between syntax correctness and verification success across all models. At Pass@8, models achieve $\mu = 95.67\%$ (SD = 4.21%) syntax correctness but only $\mu = 3.69\%$ (SD = 2.14%) verification. This gap persists independent of: (i) model scale (480B vs 32B parameters, $p > 0.05$), (ii) training specialization (code-specific vs general-purpose), (iii) architectural family (dense, MoE, constitutional), and (iv) increased sampling (Pass@1 to Pass@8). The universality of this failure suggests a fundamental architectural limitation rather than an optimization or data issue.

Observation 2. *Non-linear scaling reveals compositional breakdown.* Comparing performance degradation from single-function (DAFNYBENCH) to multi-function (DAFNYCOMP) tasks reveals super-linear complexity scaling. With 3.2× increase in functions, we observe a 14.4× decrease in verification success (from $\sim$53% to 3.69%). This disproportionate degradation cannot be explained solely by additive difficulty. Instead, it suggests that specification requirements grow combinatorially with function composition—each function boundary introduces $O(n^2)$ potential specification dependencies that models fail to capture.

Observation 3. *Sampling saturation indicates capability ceiling, not search limitations.* The verification-sampling curve plateaus by Pass@4 for all models, with the marginal improvement from Pass@4 to Pass@8 averaging only 0.8%. In contrast, syntax correctness continues improving (+7.3% on average), demonstrating that models can explore the output space but cannot discover valid specifications. This divergent behavior between syntax and semantics strongly suggests that current architectures lack the inductive biases necessary for compositional reasoning, rather than merely requiring better search strategies or more compute.

Observation 4. *Reasoning-specialized models show no clear advantage, thereby confirming architectural barriers.* Models explicitly optimized for reasoning (QWQ-32B with chain-of-thought focus, DEEPSEEK-R1 with reinforcement learning) perform no better than general-purpose models, with QWQ-32B still achieving 0% verification even at Pass@8. The tight clustering of verification rates (coefficient of variation = 0.58) across diverse training objectives strongly indicates that compositional verification requires fundamentally different architectural primitives—not refinements of existing transformer-based reasoning. This null result is particularly informative: it clearly demonstrates that neither extended reasoning traces nor reward-based optimization can overcome the absence of compositional inductive biases.

5 FAILURE CASE ANALYSIS AND DISCUSSION

Table 2: Distribution of verification failure modes across 900 analyzed cases (3 best-performing models × 300 benchmarks) from DafnyComp.

Failure Mode	**Frequency**	**% of Total**	**Primary Mechanism**
Specification Fragility	353/900	39.2	Contract propagation failure
Implementation–Proof Misalignment	195/900	21.7	Independent generation pathways
Reasoning Instability	127/900	14.1	Inductive chain breakdown
Other (syntax, timeout, misc.)	225/900	25.0	Various

The significant gap between syntax correctness and ultimate verification success demands a clear mechanistic explanation. Through a systematic analysis of 900 (3 best-performing models × 300 benchmarks) observed verification failures across three representative model families, we identify several distinct failure modes that reveal fundamental limitations in how transformers process compositional specifications. Table 2 presents the overall distribution of these failures, which we analyze in detail below.

5.1 SPECIFICATION FRAGILITY: THE DOMINO EFFECT

Specification fragility, the inability to generate contracts that remain valid across function compositions, constitutes the plurality of failures. Consider a representative case from our benchmark: a `digitSum` function correctly implemented but missing the postcondition `ensures result >= 0`. In isolation, this omission appears minor. In composition, it cascades—when `digitSum`'s output feeds a downstream function expecting non-negative input, verification fails globally despite both functions being locally correct. Note that this pattern recurs throughout our dataset. Models generate specifications sufficient for local correctness but insufficient for compositional soundness. A `requires n >= 0` precondition absent from one function invalidates the entire pipeline's verification, even when each component individually passes most test cases. The fragility stems from a fundamental mismatch: LLMs learn specifications as local patterns rather than global contracts. They lack the architectural machinery to reason about how data constraints propagate through function calls—a capability essential for modular verification. The implications extend beyond Dafny. Any system requiring compositional correctness—from distributed systems protocols to smart contract verification—will face similar failures until models can reason about specification flow across component boundaries.

The first key takeaway insight about *specification fragility* is summarized below:

> **Takeaways (i)**: *LLMs handle local specs but fail under composition. Missing contract propagation is the main cause of verification breakdowns.*

5.2 IMPLEMENTATION–PROOF MISALIGNMENT: THE INDEPENDENCE ASSUMPTION

The second failure mode reveals a deeper architectural issue: LLMs treat implementation and specification as independent generation tasks rather than coupled constraints. In 21.7% of failures, syntactically valid code contradicts its own specifications. One striking example: a model generated `assert 0 >= 1;` within otherwise reasonable code, illustrating a recurring pattern of internally contradictory assertions rather than an isolated slip. More subtle misalignments prove equally fatal. Loop invariants like `forall k :: 0 <= k < i ==> cnt[k] >= 0` appear plausible but fail verification because the implementation's array access patterns violate the stated bounds. The model generates invariants that "look right" based on training patterns but don't correspond to the actual code behavior. This pattern is consistent with mechanistic analyses of transformer attention heads, which often uncover relatively local algorithmic behaviors rather than globally consistent reasoning (Olsson et al., 2022; Elhage et al., 2021) Current training paradigms exacerbate this issue. Models learn from code-specification pairs without explicit feedback on the mutual consistency between them. The result: impressive performance on syntax and moderate success on individual functions, but catastrophic failure when consistency is required across boundaries.

We summarize the second key takeaway about *implementation-proof misalignment* as:

> **Takeaways (<u>ii</u>)**: *Code and specs are generated independently, leading to plausible but inconsistent invariants. Future training for this task should enforce better alignment.*

5.3 REASONING INSTABILITY: INDUCTION AS ACHILLES' HEEL

The third failure pattern, which we refer to as *reasoning instability*, exposes perhaps the most fundamental limitation. Formal verification relies on inductive reasoning: proving properties hold initially, maintain their validity through iterations, and compose across calls. LLMs consistently fail this inductive chain. Loop invariants that should accumulate state (e.g., `invariant res == stringToIntHelper(s[..i])`) break because models cannot track how program state evolves through iterations. Recursive functions lack proper termination arguments. Properties proven for base cases fail to extend inductively. This instability reflects the inherently statistical nature of reasoning exhibited by transformer architectures. While capable of pattern-matching similar invariants from training data, models cannot construct the inductive proofs verification demands. They approximate rather than prove, which is sufficient for typical NLP tasks but inadequate for verifiable code generation, where formal verification is required.

We summarize the third insight about *reasoning instability* as:

> **Takeaways (<u>iii</u>)**: *LLMs approximate base cases but fail to sustain inductive reasoning, exposing a structural gap in formal verification.*

6 LIMITATIONS AND FUTURE WORK

While DAFNYCOMP exposes fundamental limitations in compositional reasoning, we want to gently mention several constraints of our evaluation, which indicate some interesting future work.

- **Compositional Patterns.** We restrict to chain-based compositions (sequential function calls) rather than complex topologies (recursive compositions, mutual dependencies) due to synthesis tractability. While chains suffice to demonstrate compositional failure, real systems exhibit richer patterns. Extending to arbitrary call graphs requires solving verification tractability for cyclic dependencies—a challenge independent of LLM capabilities.
- **Specification Types.** Our benchmark tests functional correctness (preconditions, postconditions, invariants) but not liveness properties, resource bounds, or security policies. These orthogonal concerns—e.g., proving memory consumption remains constant across compositions—require different verification techniques and evaluation metrics.
- **Data Scarcity.** The core challenge may be training data availability. Repositories contain only a few verified multi-function programs with compositional specifications. Synthetic data generation or bootstrapped program synthesis could address this gap, although ensuring semantic diversity remains a challenge.

7 CONCLUSION

We introduce DAFNYCOMP, the first benchmark specifically designed to evaluate the generation of compositional specifications for formal verification. Through 300 synthesized multi-function Dafny programs, we systematically assessed 13 state-of-the-art LLMs on their ability to generate specifications that ensure correctness across function boundaries. Our results reveal a fundamental capability gap: while models achieve greater than 99% syntax correctness and more than 58% verification on single-function benchmarks, they collapse to 3.69% verification on compositional tasks—a 92% degradation. This performance cliff persists across all model families despite increased sampling (Pass@8), indicating an architectural rather than search limitation. Error analysis identifies three systematic failure modes: specification fragility (39.2%), implementation-proof misalignment (21.7%), and reasoning instability (14.1%), each reflecting the inability to maintain logical commitments across functional boundaries. In conclusion, DAFNYCOMP provides both a diagnostic tool for current systems and a concrete target for future research. We release the benchmark, evaluation framework, and synthesis pipeline to accelerate progress on this critical challenge.

REFERENCES

Anthropic. Introducing claude 3.5 sonnet. `https://www.anthropic.com/news/claude-3-5-sonnet`, 2024.

Anthropic. Claude 4 sonnet. `https://www.anthropic.com/news/claude-4-sonnet`, 2025. Accessed: 2025-09-21.

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

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374, 2021.

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

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

DeepSeek-AI. Deepseek-v3.1 model introduction. `https://www.deepseek.com/`, 2025. Accessed: 2025-09-21.

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

Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang Lorraine Li, Liwei Jiang, Bill Yuchen Lin, Sean Welleck, Peter West, Chandra Bhagavatula, Ronan Le Bras, et al. Faith and fate: Limits of transformers on compositionality. Advances in Neural Information Processing Systems, 36:70293–70332, 2023.

Nelson Elhage, Neel Nanda, Catherine Olsson, Tom Henighan, Nicholas Joseph, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, et al. A mathematical framework for transformer circuits. Transformer Circuits Thread, 1(1):12, 2021.

Google DeepMind. Gemini 2.5 models. `https://deepmind.google/technologies/gemini/`, 2025.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. arXiv preprint arXiv:2501.12948, 2025.

Wenhao Hu, Jinhao Duan, Chunchen Wei, Li Zhang, Yue Zhang, and Kaidi Xu. Dynacode: A dynamic complexity-aware code benchmark for evaluating large language models in code generation. arXiv preprint arXiv:2503.10452, 2025.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2. 5-coder technical report. arXiv preprint arXiv:2409.12186, 2024.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. Gpt-4o system card. arXiv preprint arXiv:2410.21276, 2024.

Daniel Keysers et al. Compositional generalization in natural language processing. Transactions of the Association for Computational Linguistics, 8:11–23, 2020.

K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Logic for Programming, Artificial Intelligence, and Reasoning, pp. 348–370. Springer, 2010a.

K. Rustan M. Leino. A Tour of the Dafny Program Verifier. Verified Software: Theories, Tools, Experiments, 2010b.

K. Rustan M. Leino et al. Compositional verification of a railway protection system with Dafny. Formal Aspects of Computing, 2017.

Yue Chen Li, Stefan Zetzsche, and Siva Somayyajula. Dafny as verification-aware intermediate language for code generation. arXiv preprint arXiv:2501.06283, 2025.

Zhaoyi Li, Gangwei Jiang, Hong Xie, Linqi Song, Defu Lian, and Ying Wei. Understanding and patching compositional reasoning in llms. arXiv preprint arXiv:2402.14328, 2024.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. arXiv preprint arXiv:2412.19437, 2024.

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

James Loughridge et al. DafnyBench: A Benchmark for Formal Software Verification. Transactions on Machine Learning Research, 2024a. Describes the ‘fill annotations‘ task.

James Loughridge et al. DafnySynth: A Synthetic Dataset for Formal Verification. arXiv preprint arXiv:2411.15143, 2024b.

Thomas J McCabe. A complexity measure. IEEE Transactions on software Engineering, (4):308–320, 1976.

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

Catherine Olsson, Nelson Elhage, Neel Nanda, Nicholas Joseph, Nova DasSarma, Tom Henighan, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, et al. In-context learning and induction heads. arXiv preprint arXiv:2209.11895, 2022.

OpenAI. Gpt-4.1 system card. `https://openai.com/index/introducing-gpt-4-1/`, 2024.

OpenAI. Introducing openai o3 and o4-mini. `https://openai.com/index/introducing-o3-and-o4-mini/`, 2025. Accessed: 2025-09-21.

Gabriel Poesia, Chloe Loughridge, and Nada Amin. dafny-annotator: Ai-assisted verification of dafny programs. arXiv preprint arXiv:2411.15143, 2024.

Qwen Team. Qwen3-coder: Agentic coding in the world. `https://qwenlm.github.io/blog/qwen3-coder/`, 2025a. Accessed: 2025-09-21.

Qwen Team. Qwq-32b: Large-scale reinforcement learning for reasoning models. `https://qwenlm.github.io/zh/blog/qwq-32b/`, 2025b. Accessed: 2025-09-21.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. arXiv preprint arXiv:2408.03314, 2024.

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

Yunhui Xia, Wei Shen, Yan Wang, Jason Klein Liu, Huifeng Sun, Siyue Wu, Jian Hu, and Xiaolong Xu. Leetcodedataset: A temporal dataset for robust evaluation and efficient training of code llms. arXiv preprint arXiv:2504.14655, 2025.

Chuanhao Yan, Fengdi Che, Xuhan Huang, Xu Xu, Xin Li, Yizhi Li, Xingwei Qu, Jingzhe Shi, Zhuangzhuang He, Chenghua Lin, Yaodong Yang, Binhang Yuan, Hang Zhao, Yu Qiao, Bowen Zhou, and Jie Fu. Re:Form – Reducing Human Priors in Scalable Formal Software Verification with RL in LLMs: A Preliminary Study on Dafny. arXiv preprint arXiv:2507.16331, 2025.

Zhehao Zhang, Jiaao Chen, and Diyi Yang. Darg: Dynamic evaluation of large language models via adaptive reasoning graph. Advances in Neural Information Processing Systems, 37:135904–135942, 2024.

Kaijie Zhu, Jiaao Chen, Jindong Wang, Neil Zhenqiang Gong, Diyi Yang, and Xing Xie. Dyval: Dynamic evaluation of large language models for reasoning tasks. arXiv preprint arXiv:2309.17167, 2023.

Kaijie Zhu, Jindong Wang, Qinlin Zhao, Ruochen Xu, and Xing Xie. Dyval 2: Dynamic evaluation of large language models by meta probing agents. arXiv preprint arXiv:2402.14865, 3, 2024.

A THE USE OF LLMS IN WRITING

During the preparation of this manuscript, we employed a large language model (OpenAI GPT-5) to assist with language refinement and editorial improvements. Specifically, the LLM was used to enhance sentence fluency, improve clarity of expression, and ensure consistency with academic writing conventions. The tool was applied exclusively for linguistic polishing—all research design, experimental work, data analysis, and core intellectual contributions remain entirely original.

B INTRODUCTION TO DAFNY

Dafny (Leino, 2010a), developed at Microsoft Research, is a verification-oriented programming language specifically designed to support formal reasoning about software. Unlike conventional languages where correctness is primarily assessed through testing, Dafny integrates an automated program verifier directly into the development workflow, enabling developers to construct code that is mathematically proven to satisfy its specifications. This approach shifts the discovery of defects from the testing phase to the design and implementation phases, thereby improving software reliability.

A distinctive feature of Dafny is that specifications are treated as first-class citizens. Methods can be annotated with preconditions, postconditions, and logical properties that describe intended behavior. For example:

```
method Example(a: array<int>) returns (b: array<int>)
  requires a.Length > 0                 // Precondition
  ensures b.Length == a.Length          // Postcondition
  ensures forall i :: 0 <= i < b.Length ==> b[i] >= 0  // Property
{
  // Implementation
}
```

The Dafny verifier relies on automated theorem proving (via Z3 solver (De Moura & Bjørner, 2008)) to ensure that implementations conform to these specifications, providing mathematical certainty about program behavior. Crucially, the ability to reason about the composition of verified components determines whether verification can scale from toy examples to real-world systems. Without compositional reasoning, verification remains confined to small, isolated programs rather than production-level software.

C AUTOMATED THEOREM PROVING

A complementary line of work contrasts automated verification frameworks with interactive theorem proving (ITP) systems. Languages such as Dafny and Verus rely on SMT solvers to discharge proof obligations, requiring only lightweight annotations (e.g., invariants, assertions). This design lowers the barrier to entry but is constrained by the solver's limited reasoning scope and opaque failure modes. In contrast, ITPs such as Lean expose every proof step explicitly, enabling iterative refinement and error diagnosis. Recent studies even show that LLMs can generate competition-level mathematical proofs in Lean. However, existing Lean-based benchmarks (e.g., miniCodeProps, FVAPPS) either focus narrowly on proof synthesis or lack human validation. By comparison, Dafny offers a more balanced environment for benchmarking LLMs: it combines code, specifications, and automated verification in a way that remains close to mainstream programming practice.

How Dafny Works and Its Core Strengths. Dafny's approach stems from its verification-aware design. Developers embed formal specifications, such as preconditions, postconditions, and loop invariants, directly within the code (Leino, 2010a). These specifications are not merely comments; they are integral components checked by the built-in verifier. The verifier translates Dafny code and its specifications into an intermediate verification language, Boogie, which then generates proof obligations. These obligations are processed by an SMT solver (e.g., Z3) to prove their validity. If all obligations are proven, the code is confirmed to be correct according to its specifications. If a proof fails, Dafny provides precise feedback on the inconsistencies. This methodology supports correctness by construction, helping to reduce common errors like null pointer dereferences or array out-of-bounds access (Poesia et al., 2024). Once verified, Dafny code can be translated into mainstream languages such as Python for execution (Li et al., 2025).

Dafny vs. Python: A Fundamental Difference in Approach. To understand Dafny's position, it's useful to compare it with a widely used language like Python. While both are effective, their fundamental design philosophies and primary objectives differ, as shown in Table 3.

Feature	Dafny	Python
Year Introduced	2010 (Microsoft Research)	1991 (Guido van Rossum)
Type System	Static typing, compile-time checks	Dynamic typing, run-time checks
Formal Verification	**Yes** — built-in contracts and proofs	No — only basic `assert`
Main Use	Verified algorithms, critical systems	General-purpose programming
Execution Model	Compiled with verification	Interpreted (e.g., CPython)

Table 3: Key differences between Dafny and Python.

In summary, Dafny offers a distinct approach to software development by integrating formal verification into the language itself. While Python excels in agile development and broad applicability, Dafny is particularly suited for domains where software correctness and formal guarantees are critical. For more, please refer to the Dafny official website[1].

D MODEL SELECTION FOR FORMAL TRANSLATION

Concretely, we randomly sampled 100 test cases from the 1,200 instances obtained after our assembly procedure to evaluate each model with identical prompts and decoding settings. We define success rate as the fraction of generations whose specifications are valid under the Dafny verifier.

Model	Success rate (%)
Claude-4-Sonnet-20250514	58.00
claude-3.5-sonnet-20241022	55.00
gpt-3.5-turbo	45.00
gpt-4o	31.00
gpt-4o-mini	41.00
o1	36.00
o1-mini	33.00
o3-mini	37.00
gemini-2.0-flash	38.00

Table 4: Success rates for the formal translation step.

As shown, we ultimately chose Claude-4-Sonnet-20250514 because it yielded the highest rate of valid specifications on these cases. This aligns with our design goal to prioritize the quality of generated specifications used downstream in our experiments.

---

[1] `https://dafny.org/dafny/OnlineTutorial/guide`

E PROMPTS FOR SYNTHESIS

The prompt templates used for annotating data with Claude 3.5 Sonnet are shown in the following boxes.

Prompt for Inital Dafny Code Generation

SYSTEM

You are an expert AI assistant that writes Dafny programs. You excel at writing code with formally verified correctness, providing precise preconditions and postconditions, and finding the appropriate loop invariants to ensure all verification conditions are met.

TASK

Below is the Python code:

~~~
```python
<python_code>
```
~~~

Please translate this Python code into Dafny, ensuring:

1. **Method Signatures**: Each piece of functionality should be expressed as a Dafny method (or set of methods) with a well-defined signature.
2. **Preconditions**: Clearly state any ‘requires‘ clauses for each method (e.g., array length constraints, non-null references, numeric domain restrictions, etc.).
3. **Postconditions**: State the logical guarantees about the returned values or final state as ‘ensures‘ clauses (e.g., correctness of returned results, absence of side effects, etc.).
4. **Verification Details**: Include all necessary loop invariants (or other verification hints) so Dafny can prove the postconditions, along with a brief explanation. For example: - Explain how you chose your invariants. - Describe how they ensure the correctness of the loop.

Return the final Dafny code as a self-contained snippet that can be verified by Dafny as-is, with a short explanation of how it connects to the original Python functionality.

AI ASSISTANT

<The LLM’s generated Dafny code with specifications here.>

Dynamic Debugging Prompt for Code Generation

SYSTEM
You are an expert AI assistant that writes and debugs Dafny programs. You excel at diagnosing and fixing verification errors based on Dafny solver messages, while maintaining correct preconditions, postconditions, and loop invariants.

TASK
Below is the Python code:

~~~
```python
<python_code>
```
~~~

And the Dafny code you previously provided (which I tried to verify):

~~~
```dafny
<main_spec>
```
~~~

I ran `dafny verify *.dfy` and received this error message:

~~~
```
<dafny_analysis_result>
```
~~~

Can you please fix the main function specification so that it parses successfully? Output the corrected main function specification only, without any other text.

AI ASSISTANT
<The LLM's generated Dafny code with specifications here.>

F DATA CONTAMINATION ANALYSIS

To validate the novelty of DAFNYCOMP, we conducted a rigorous data contamination analysis against the widely-used MBPP dataset (Austin et al., 2021), used to assess contamination in Python source data. We confirm that our benchmark source data shows no significant overlap, ensuring model performance reflects genuine reasoning capabilities rather than memorization.

Our analysis, focusing solely on code, employs two standard metrics: **Exact Match** to detect verbatim copies, and **n-gram Jaccard Similarity** to identify structurally similar code. We performed this analysis under four distinct configurations, the results of which are summarized in Table 5.

Across all scenarios, we found **zero exact matches**. The n-gram Jaccard similarity remains negligible, peaking at a mere 0.0078 even under the most aggressive settings. These findings provide strong evidence that DAFNYCOMP is free from training data contamination.

Table 5: Summary of Data Contamination Analysis. The table shows results for four testing configurations: **A (Conservative)** with minimal preprocessing; **B (Default)** with moderate preprocessing; **C (Aggressive)** with extensive preprocessing; and **D (Holistic)** for a structure-level check. Across all configurations, results show zero exact matches and negligible n-gram similarity when comparing DAFNYCOMP source data against MBPP, confirming the benchmark's integrity.

Analysis Configuration	N-gram (n)	Exact Overlap	Max Jaccard	vs. sanitized-mbpp
A: Conservative	15	0	0.000078	0
B: Default	11, 13, 15	0	0.000389	0
C: Aggressive	9, 11, 13	0	0.007757	0
D: Holistic	11, 13, 15	0	0.000234	0

G PROMPT FOR EVALUATION

The prompt template used for evaluation is shown in the following box. Note that all model outputs are used directly for Dafny verification.

Evaluation Prompt for Dafny Specification Generation

SYSTEM

You are an expert in Dafny. You will be given tasks dealing with Dafny programs including precise annotations. You should only return code body in all circumstances. No text is allowed.

TASK

Given a Dafny program with function signature, preconditions, postconditions, and code, but with annotations missing. Please return a complete Dafny program with the strongest possible annotation (loop invariants, assert statements, etc.) filled back in. Do not explain or output any text. If you have to explain, put all explanations in comments form. There should only be code body in your output. Please use exactly the same function signature, preconditions, and postconditions. Do not ever modify the given lines.
Below is the program:

~~~
```dafny
<dafny_program_with_missing_annotations>
```
~~~

AI ASSISTANT

~~~
```dafny
<The LLM's generated Dafny code with specifications
here.>
```
~~~

H COMPUTE SETTINGS

Details of Compute Settings

Token budgets:
- No hard constraint is imposed on the number of generated tokens.
- For each model, `max_output_tokens` (or equivalent) is set to the **largest value allowed by the provider**, so that long reasoning traces are not truncated.
- Many evaluated systems are reasoning-oriented (e.g., R1) and may follow long, iterative reasoning trajectories, so token usage is largely emergent and model-dependent.

Inference sampling:
- Default decoding for most models: temperature $= 0.7$, top-p $= 0.8$.
- CLAUDE MODELS: provider default top-p (not exposed in the API), with temperature $= 0.7$.
- O4-MINI: provider default configuration, with fixed temperature $= 1$ and no top-p parameter.

Verifier retry policy:
- Exactly one verification attempt per output.
- SMT solver timeout: 60 s.
- If the solver does not finish within 60 s, the instance is treated as unsolved (not counted as a success).

I REAL EXAMPLES FROM BENCHMARK DATA

I.1 CHALLENGE 1: FORMAL SPECIFICATION SYNTHESIS

The following examples demonstrate failures in generating correct formal specifications, particularly preconditions and postconditions.

I.1.1 EXAMPLE 1.1: POSTCONDITION FAILURE (ORG_INPUT_ID: 2, SELF_ID: 20001)

LLM Response:

```
method digitSum(n: int) returns (sum: int)
    requires n >= 0
    ensures sum >= 0
{
    var temp := n;
    sum := 0;
    while temp > 0
        invariant temp >= 0
        invariant sum >= 0
    {
        sum := sum + (temp % 10);
        temp := temp / 10;
    }
}

method countLargestGroup_1399(n: int) returns (result: int)
    requires n >= 0
    ensures result >= 1
    ensures result <= 1000
{
    var counts := new int[50];
    var i := 0;
    while i < 50
        invariant 0 <= i <= 50
        invariant forall j :: 0 <= j < i ==> counts[j] == 0
    {
        counts[i] := 0;
        i := i + 1;
    }
    var maxCount := 0;
    var groupsWithMaxCount := 0;
    i := 1;
    while i <= n
        invariant 1 <= i <= n + 1
        invariant maxCount >= 0
        invariant groupsWithMaxCount >= 0
        invariant maxCount > 0 ==> groupsWithMaxCount >= 1
        invariant forall j :: 0 <= j < 50 ==> counts[j] >= 0
    {
        var ds := digitSum(i);
        if ds < 50 {
            counts[ds] := counts[ds] + 1;
            if maxCount < counts[ds] {
                maxCount := counts[ds];
                groupsWithMaxCount := 1;
            } else if maxCount == counts[ds] {
                groupsWithMaxCount := groupsWithMaxCount + 1;
            }
        }
        i := i + 1;
    }
    result := groupsWithMaxCount;
    if result == 0 {
        result := 1;
    }
    assert result >= 1;
    if result > 1000 {
        result := 1000;
    }
}

method main_3node_2(o: int) returns (result: int)
    requires o >= 0
    ensures result == -1 || result >= 2
{
    var o1 := countLargestGroup_1399(o);
    var o2 := sumOfMultiples_2652(o1);
    if o2 >= 1 {
        var o3 := nextGreaterElement_556(o2);
        result := o3;
        if result != -1 && result <= 1 {
            result := 2;
        }
    } else {
        result := -1;
    }
}
```

Dafny Error:

```
ex.dfy(137,0): Error: a postcondition could not be proved on this return
    path
ex.dfy(136,25): Related location: this is the postcondition that could
    not be proved
```

I.1.2 Example 1.2: Missing Precondition (org_input_id: 6, self_id: 60005)

LLM Response:

```
method DigitSum(n: int) returns (sum: int)
    requires n >= 0
    ensures sum >= 0
{
    var num := n;
    sum := 0;
    while num > 0
        invariant sum >= 0
        invariant num >= 0
        decreases num
    {
        sum := sum + (num % 10);
        num := num / 10;
    }
}

method countLargestGroup_1399(n: int) returns (result: int)
    requires n >= 1
    ensures result >= 1
{
    var digitSumCounts := map[];
    var maxCount := 0;
    var i := 1;
    while i <= n
        invariant 1 <= i <= n + 1
        invariant maxCount >= 0
        invariant forall k :: k in digitSumCounts ==> digitSumCounts[k]
            >= 0
        decreases n - i + 1
    {
        var digitSum := DigitSum(i);
        var currentCount := if digitSum in digitSumCounts then
            digitSumCounts[digitSum] else 0;
        digitSumCounts := digitSumCounts[digitSum := currentCount + 1];
        if currentCount + 1 > maxCount {
            maxCount := currentCount + 1;
        }
        i := i + 1;
    }
    result := 0;
    var possibleSum := 0;
    while possibleSum <= 9 * 4
        invariant possibleSum >= 0
        invariant result >= 0
        invariant forall k :: k in digitSumCounts ==> digitSumCounts[k]
            >= 0
        decreases 9 * 4 - possibleSum + 1
    {
        if possibleSum in digitSumCounts && digitSumCounts[possibleSum]
            == maxCount {
            result := result + 1;
        }
        possibleSum := possibleSum + 1;
    }
    if result == 0 {
        result := 1;
    }
}

method main_4node_4(o: int) returns (result: int)
    ensures result >= 1
{
    var o1 := nextGreaterElement_556(o);
    var o2: int;
    if o1 == -1 {
        o2 := 1;
    } else if o1 > 10000 {
        o2 := 10000;
    } else {
        o2 := o1;
    }
    var o3 := countLargestGroup_1399(o2);
    var clampedO3: int;
    if o3 > 1000000000000000 {
        clampedO3 := 1000000000000000;
    } else if o3 < -1000000000000000 {
        clampedO3 := -1000000000000000;
    } else {
        clampedO3 := o3;
    }
    var o4 := smallestNumber_2165(clampedO3);
    var finalInput: int;
    if o4 < 1 {
        finalInput := 1;
    } else if o4 > 15 {
        finalInput := 15;
    } else {
        finalInput := o4;
    }
    result := countArrangement_526(finalInput);
}
```

Dafny Error:

```
ex.dfy(177,36): Error: a precondition for this call could not be proved
```

I.2 CHALLENGE 2: ENSURING CODE CORRECTNESS

These examples show implementation issues where the code logic doesn't satisfy the specifications.

I.2.1 EXAMPLE 2.1: ARRAY BOUNDS ERROR (ORG_INPUT_ID: 1, SELF_ID: 10002)

LLM Response:

```
method smallestNumber_2165(num: int) returns (result: int)
{
    var neg := num < 0;
    var n := if num < 0 then -num else num;
    var cnt := new int[10];
    var i := 0;
    while i < 10
        invariant 0 <= i <= 10
        invariant cnt.Length == 10
        invariant forall k :: 0 <= k < i ==> cnt[k] >= 0
    {
        cnt[i] := 0;
        i := i + 1;
    }
    // ... rest of implementation
}

method main_5node_8(o: int) returns (result: int)
{
    var o1 := smallestNumber_2165(o);
    var o2 := clumsy_1006(if o1 <= 10000 then o1 else 10000);
    var o3 := lastRemaining_390(o2);
    var o4 := getMoneyAmount_375(if o3 <= 200 then o3 else 200);
    var o5 := numberOfWays_3183(if o4 <= 100000 then o4 else 100000);
    result := o5;
}
```

Dafny Error:

```
ex.dfy(44,30): Error: this loop invariant could not be proved on entry
Related message: loop invariant violation
```

I.2.2 Example 2.2: Assertion Failure (org_input_id: 6, self_id: 60001)

LLM Response:

```
method DigitSum(n: int) returns (sum: int)
    requires n >= 0
    ensures sum >= 0
{
    var num := n;
    sum := 0;
    while num > 0
        invariant sum >= 0
        invariant num >= 0
        decreases num
    {
        sum := sum + (num % 10);
        num := num / 10;
    }
}

method countLargestGroup_1399(n: int) returns (result: int)
    requires n >= 1
    ensures result >= 1
{
    var digitSumCounts := map[];
    var maxCount := 0;
    var i := 1;
    while i <= n
        invariant 1 <= i <= n + 1
        invariant maxCount >= 0
        invariant forall k :: k in digitSumCounts ==> digitSumCounts[k] >
             0
        decreases n - i + 1
    {
        var digitSum := DigitSum(i);
        var currentCount := if digitSum in digitSumCounts then
            digitSumCounts[digitSum] else 0;
        digitSumCounts := digitSumCounts[digitSum := currentCount + 1];
        if currentCount + 1 > maxCount {
            maxCount := currentCount + 1;
        }
        i := i + 1;
    }

    result := 0;
    var possibleSum := 0;
    while possibleSum <= 9 * 4
        invariant possibleSum >= 0
        invariant result >= 0
        decreases 9 * 4 - possibleSum + 1
    {
        if possibleSum in digitSumCounts && digitSumCounts[possibleSum]
            == maxCount {
            result := result + 1;
        }
        possibleSum := possibleSum + 1;
    }

    if result == 0 {
        result := 1;
    }
}

method main_4node_4(o: int) returns (result: int)
    ensures result >= 1
{
    var o1 := nextGreaterElement_556(o);
    var o2: int;
    if o1 == -1 {
        o2 := 1;
    } else if o1 > 10000 {
        o2 := 10000;
    } else {
        o2 := o1;
    }
    assert o2 >= 1;

    var o3 := countLargestGroup_1399(o2);
    var clampedO3: int;
    if o3 > 1000000000000000 {
        clampedO3 := 1000000000000000;
    } else if o3 < -1000000000000000 {
        clampedO3 := -1000000000000000;
    } else {
        clampedO3 := o3;
    }

    var o4 := smallestNumber_2165(clampedO3);
    var finalInput: int;
    if o4 < 1 {
        finalInput := 1;
    } else if o4 > 15 {
        finalInput := 15;
    } else {
        finalInput := o4;
    }
    assert finalInput >= 1;

    result := countArrangement_526(finalInput);
}
```

Dafny Error:

```
ex.dfy(181,4): Error: assertion might not hold
assert o2 >= 1;
```

I.3 Challenge 3: Proof Generation and Formal Reasoning

Examples demonstrating failures in generating proper loop invariants and formal proofs.

I.3.1 Example 3.1: Invariant Not Maintained (org_input_id: 0, self_id: 6)

LLM Response:

```
method stringToInt(s: seq<char>) returns (result: int)
  requires |s| > 0
  requires forall i :: 0 <= i < |s| ==> ’0’ <= s[i] <= ’9’
  ensures result >= 0
{
  var res := 0;
  var i := 0;
  while i < |s|
    invariant 0 <= i <= |s|
    invariant res >= 0
    invariant res == stringToIntHelper(s[..i])
  {
    var digit := digitToInt(s[i]);
    assert 0 <= digit <= 9;
    res := res * 10 + digit;
    i := i + 1;
  }
  return res;
}

method monotoneIncreasingDigits_738(n: int) returns (result: int)
  requires n >= 0
  ensures result >= 1
  ensures result <= 1000000000
{
  if n == 0 {
    return 1;
  }
  var s := intToString(n);
  var i := 1;
  while i < |s| && digitToInt(s[i-1]) <= digitToInt(s[i])
    invariant 1 <= i <= |s|
    invariant forall j :: 0 <= j < |s| ==> ’0’ <= s[j] <= ’9’
  {
    i := i + 1;
  }
  if i < |s| {
    while i > 0 && i < |s| && digitToInt(s[i-1]) > digitToInt(s[i])
      invariant 0 <= i < |s|
      invariant forall j :: 0 <= j < |s| ==> ’0’ <= s[j] <= ’9’
      decreases i
    {
      var prevDigit := digitToInt(s[i-1]);
      if prevDigit > 0 {
        s := s[i-1 := intToChar(prevDigit - 1)];
      }
      i := i - 1;
    }
    i := i + 1;
    while i < |s|
      invariant i <= |s|
      invariant forall j :: 0 <= j < |s| ==> ’0’ <= s[j] <= ’9’
    {
      s := s[i := ’9’];
      i := i + 1;
    }
  }
  var res := stringToInt(s);
  if res >= 1 && res <= 1000000000 {
    return res;
  } else {
    return 1;
  }
}

method main_3node_2(o: int) returns (result: int)
  requires o >= 0
  ensures result >= 0
{
  var o1 := smallestFactorization_625(o);
  if o1 == 0 {
    return 0;
  }
  var o2 := monotoneIncreasingDigits_738(o1);
  var o3 := minOperations_2571(o2);
  return o3;
}
```

Dafny Error:

```
ex.dfy(25,18): Error: this invariant could not be proved to be maintained
     by the loop
Related message: loop invariant violation
```

I.3.2 EXAMPLE 3.2: RECURSION TERMINATION (ORG_INPUT_ID: 4, SELF_ID: 40003)

LLM Response:

```
function gcd_func(a: int, b: int): int
{
    if b == 0 then a else gcd_func(b, a % b)
}

method gcd(a: int, b: int) returns (result: int)
requires a > 0 && b >= 0
ensures result == gcd_func(a, b)
{
    var x := a;
    var y := b;
    while y != 0
        invariant x > 0 && y >= 0
        invariant gcd_func(a, b) == gcd_func(x, y)
        decreases y
    {
        var temp := x % y;
        x := y;
        y := temp;
    }
    result := x;
}

method distinctSequences_2318(n: int) returns (result: int)
requires n >= 1
ensures result >= 0
{
    if n == 1 {
        return 6;
    }
    var mod := 1000000007;
    var dp := new int[n + 1, 6, 6];
    var k := 0;
    while k <= n
        invariant 0 <= k <= n + 1
    {
        var i := 0;
        while i < 6
            invariant 0 <= i <= 6
        {
            var j := 0;
            while j < 6
                invariant 0 <= j <= 6
            {
                dp[k, i, j] := 0;
                j := j + 1;
            }
            i := i + 1;
        }
        k := k + 1;
    }
    var i := 0;
    while i < 6
        invariant 0 <= i <= 6
    {
        var j := 0;
        while j < 6
            invariant 0 <= j <= 6
        {
            var g := gcd(i + 1, j + 1);
            if g == 1 && i != j {
                dp[2, i, j] := 1;
            }
            j := j + 1;
        }
        i := i + 1;
    }
    k := 3;
    while k <= n
        invariant 3 <= k <= n + 1
    {
        i := 0;
        while i < 6
            invariant 0 <= i <= 6
        {
            var j := 0;
            while j < 6
                invariant 0 <= j <= 6
            {
                var g1 := gcd(i + 1, j + 1);
                if g1 == 1 && i != j {
                    var h := 0;
                    while h < 6
                        invariant 0 <= h <= 6
                    {
                        var g2 := gcd(h + 1, i + 1);
                        if g2 == 1 && h != i && h != j {
                            dp[k, i, j] := (dp[k, i, j] + dp[k - 1, h, i
                                ]) % mod;
                        }
                        h := h + 1;
                    }
                }
                j := j + 1;
            }
            i := i + 1;
        }
        k := k + 1;
    }
    var ans := 0;
    i := 0;
    while i < 6
        invariant 0 <= i <= 6
        invariant ans >= 0
    {
        var j := 0;
        while j < 6
            invariant 0 <= j <= 6
            invariant ans >= 0
        {
            ans := (ans + dp[n, i, j]) % mod;
            j := j + 1;
        }
        i := i + 1;
    }
    result := ans;
}

method main_3node_2(o: int) returns (result: int)
requires o >= 1
ensures result >= 0
{
    var o1 := clumsy_1006(o);
    var o2 := numSquares_279(o1);
    var o3 := distinctSequences_2318(o2);
    result := o3;
}
```

Dafny Error:

```
ex.dfy(3,34): Error: cannot prove termination; try supplying a decreases
    clause
if b == 0 then a else gcd_func(b, a % b)
```

I.4 Challenge 4: Spec-Code-Proof Alignment

These examples show misalignment between specifications and implementation, particularly method vs. function usage.

I.4.1 Example 4.1: Method in Expression Context (org_input_id: 0, self_id: 5)

LLM Response:

```
method stringToInt(s: seq<char>) returns (result: int)
requires forall i :: 0 <= i < |s| ==> ’0’ <= s[i] <= ’9’
ensures result >= 0
{
    var res := 0;
    var i := 0;
    while i < |s|
        invariant 0 <= i <= |s|
        invariant res >= 0
        invariant res == if i == 0 then 0 else stringToInt(s[..i])
    {
        res := res * 10 + digitToInt(s[i]);
        i := i + 1;
    }
    result := res;
}

method monotoneIncreasingDigits_738(n: int) returns (result: int)
  requires n >= 0
  ensures result >= 1
  ensures result <= 1000000000
{
  if n == 0 {
    return 1;
  }
  var s := intToString(n);
  var i := 1;
  while i < |s| && digitToInt(s[i-1]) <= digitToInt(s[i])
    invariant 1 <= i <= |s|
    invariant forall j :: 0 <= j < |s| ==> ’0’ <= s[j] <= ’9’
  {
    i := i + 1;
  }
  if i < |s| {
    while i > 0 && i < |s| && digitToInt(s[i-1]) > digitToInt(s[i])
      invariant 0 <= i < |s|
      invariant forall j :: 0 <= j < |s| ==> ’0’ <= s[j] <= ’9’
      decreases i
    {
      var prevDigit := digitToInt(s[i-1]);
      if prevDigit > 0 {
        s := s[i-1 := intToChar(prevDigit - 1)];
      }
      i := i - 1;
    }
    i := i + 1;
    while i < |s|
      invariant i <= |s|
      invariant forall j :: 0 <= j < |s| ==> ’0’ <= s[j] <= ’9’
    {
      s := s[i := ’9’];
      i := i + 1;
    }
  }
  var res := stringToInt(s);
  if res >= 1 && res <= 1000000000 {
    return res;
  } else {
    return 1;
  }
}

method main_3node_2(o: int) returns (result: int)
  requires o >= 0
  ensures result >= 0
{
  var o1 := smallestFactorization_625(o);
  if o1 == 0 {
    return 0;
  }
  var o2 := monotoneIncreasingDigits_738(o1);
  var o3 := minOperations_2571(o2);
  return o3;
}
```

Dafny Error:

```
ex.dfy(26,43): Error: expression is not allowed to invoke a method (
    stringToInt)
invariant res == if i == 0 then 0 else stringToInt(s[..i])
```

I.4.2 EXAMPLE 4.2: FUNCTION VS METHOD CONFUSION (ORG_INPUT_ID: 16, SELF_ID: 160000)

LLM Response:

```
method isBeautiful(n: int) returns (result: bool)
{
    // Implementation for checking if number is beautiful
    result := true; // Simplified
}

method nextBeautifulNumber_769(n: int) returns (result: int)
{
    var x := n + 1;
    while true
        invariant x >= n + 1
        invariant forall y :: n + 1 <= y < x ==> !isBeautiful(y)
        decreases *
    {
        var beautiful := isBeautiful(x);
        if beautiful {
            result := x;
            return;
        }
        x := x + 1;
    }
}


method main_5node_8(o: int) returns (result: int)
{
  var o1 := reverse_7(o);
  if o1 == 0 {
    result := 1;
    return;
  }
  var o2 := lastRemaining_390(o1);
  var o3 := closestFair_2417(o2);
  var o4 := nextGreaterElement_556(o3);
  if o4 == -1 {
    result := 1;
    return;
  }
  var o5 := nextBeautifulNumber_2048(o4);
  result := o5;
}
```

Dafny Error:

```
ex.dfy(248,46): Error: expression is not allowed to invoke a method (
    isBeautiful)
invariant forall y :: n + 1 <= y < x ==> !isBeautiful(y)
```

I.5 Challenge 5: Complexity and Scalability Issues

Examples where complex programs or large data structures cause verification failures.

I.5.1 Example 5.1: Large Array Allocation (org_input_id: 1, self_id: 10002)

LLM Response:

```
method smallestNumber_2165(num: int) returns (result: int)
{
    var neg := num < 0;
    var n := if num < 0 then -num else num;
    var cnt := new int[10];
    var i := 0;
    while i < 10
        invariant 0 <= i <= 10
        invariant cnt.Length == 10
        invariant forall k :: 0 <= k < i ==> cnt[k] >= 0
    {
        cnt[i] := 0;
        i := i + 1;
    }

    // Count digits
    var temp := n;
    while temp > 0
        invariant temp >= 0
        invariant forall k :: 0 <= k < 10 ==> cnt[k] >= 0
        decreases temp
    {
        var digit := temp % 10;
        cnt[digit] := cnt[digit] + 1;
        temp := temp / 10;
    }

    // Construct result
    result := 0;
    i := if neg then 1 else 0;
    while i < 10
        invariant 0 <= i <= 10
        invariant result >= 0
        decreases 10 - i
    {
        while cnt[i] > 0
            invariant cnt[i] >= 0
            invariant result >= 0
            decreases cnt[i]
        {
            result := result * 10 + i;
            cnt[i] := cnt[i] - 1;
        }
        i := i + 1;
    }

    if neg {
        result := -result;
    }
}

method main_5node_8(o: int) returns (result: int)
{
    var o1 := smallestNumber_2165(o);
    var o2 := clumsy_1006(if o1 <= 10000 then o1 else 10000);
    var o3 := lastRemaining_390(o2);
    var o4 := getMoneyAmount_375(if o3 <= 200 then o3 else 200);
    var o5 := numberOfWays_3183(if o4 <= 100000 then o4 else 100000);
    result := o5;
}
```

Dafny Error:

```
ex.dfy(44,30): Error: this loop invariant could not be proved on entry
Related message: loop invariant violation
invariant ans >= 0
```

I.6 Challenge 6: Avoiding Trivial Solutions

Note: This category had limited representation in the dataset. The theoretical examples that would fit this category include:

- Overly permissive postconditions like `ensures true`
- Methods with missing functional specifications
- Weak preconditions that don’t prevent edge cases
- Specifications that allow degenerate solutions
- Missing bounds on return values
- Incomplete coverage of input space
- Trivial loop invariants that don’t capture semantics
- Weak assertions that don’t prevent bugs
- Generic error handling without specific constraints

I.7 Challenge 7: Limited Training in Formal Domains

Examples showing basic syntax errors and misuse of Dafny constructs.

I.7.1 Example 7.1: Parse Error (org_input_id: 0, self_id: 0)

LLM Response:

```
function digitToInt(c: char): int
  requires ’0’ <= c <= ’9’
  ensures 0 <= digitToInt(c) <= 9
{
  (c as int) - (’0’ as int)
}

function intToChar(i: int): char
  requires 0 <= i <= 9
  ensures ’0’ <= intToChar(i) <= ’9’
{
  ((’0’ as int) + i) as char
}

method stringToInt(s: string) returns (result: int)
  requires forall i :: 0 <= i < |s| ==> ’0’ <= s[i] <= ’9’
  ensures result >= 0
{
  var res := 0;
  var i := 0;
  while i < |s|
    invariant 0 <= i <= |s|
    invariant res >= 0
  {
    res := res * 10 + digitToInt(s[i]);
    i := i + 1;
  }
  result := res;
}

method monotoneIncreasingDigits_738(n: int) returns (result: int)
requires n >= 0
ensures 1 <= result <= 1000000000
{
    if n == 0 {
        return 1;
    }
    var s := intToString(n);
    var digits := s;
    var i := 1;
    while i < |digits| && digitToInt(digits[i-1]) <= digitToInt(digits[i
        ])
        invariant 1 <= i <= |digits|
        invariant forall j :: 0 <= j < |digits| ==> ’0’ <= digits[j] <=
            ’9’
        invariant forall j,k :: 0 <= j < k < i ==> digitToInt(digits[j])
            <= digitToInt(digits[k])
    {
        i := i + 1;
    }
    if i < |digits| {
        while i > 0 && i < |digits| && digitToInt(digits[i-1]) >
            digitToInt(digits[i])
            invariant 0 <= i < |digits|
            invariant forall j :: 0 <= j < |digits| ==> ’0’ <= digits[j]
                <= ’9’
            decreases i
        {
            var prevDigit := digitToInt(digits[i-1]);
            if prevDigit > 0 {
                digits := digits[i-1 := intToChar(prevDigit - 1)];
            }
            i := i - 1;
        }
        i := i + 1;
        while i < |digits|
            invariant 0 <= i <= |digits|
            invariant forall j :: 0 <= j < |digits| ==> ’0’ <= digits[j]
                <= ’9’
        {
            digits := digits[i := ’9’];
            i := i + 1;
        }
    }
    var res := stringToInt(digits);
    if res == 0 {
        return 1;
    }
    if res > 1000000000 {
        return 1000000000;
    }
    return res;
}

method main_3node_2(o: int) returns (result: int)
requires o >= 0
ensures result >= 1
{
    var o1 := smallestFactorization_625(o);
    var o2 := monotoneIncreasingDigits_738(o1);
    var o3 := lastRemaining_390(o2);
    return o3;
}
```

Dafny Error:

```
ex.dfy(22,12): Error: invalid ForLoopStmt
22 |             reads s
method stringToInt(s: string) returns (result: int)

2 parse errors detected in ex.dfy
```