

Spec Mutation Survival Analysis: Measuring Specification Coverage for AI-Generated Code

Guy Nutman

UCLA, Henry Samueli School of Engineering

Nir Nutman

UCSB, Robert Mehrabian College of Engineering

With Apart Research — Secure Program Synthesis Hackathon, May 2026

Abstract

Formal verification promises mathematical certainty that code is correct. But that promise is only as strong as the specification being verified against. A specification with redundant or missing clauses can pass all verification checks while permitting dangerous behavior — the code is verified, but verified against the wrong thing. We introduce **spec coverage**: an automated metric that quantifies what fraction of a specification’s clauses are genuinely load-bearing versus redundant. Our approach applies mutation testing — a well-established software quality technique — to the specification layer itself. Given a plain-English function description, our tool generates a property-based specification, produces mutated variants (operator flips, assertion removals), and measures which mutations go undetected. We evaluate on 9 functions across three complexity levels using both hardcoded and LLM-generated specs. With hardcoded specs, we find an average coverage of 67.5% — meaning 1 in 3 clauses is redundant. Critically, when we explicitly instructed the LLM not to write redundant clauses, it still produced structurally incorrect specs for most functions, revealing a fundamental reliability problem with LLM-generated specifications. We argue spec coverage is a practical, expertise-free entry point into formal specification quality assessment.

1. Introduction

The bottleneck in trustworthy AI-generated software is no longer writing code — it is specifying what the code should do and verifying it does so correctly. As AI systems generate code at increasing scale, formal specifications become critical checkpoints. But as Dodds (2025) argues, specifications themselves are hard to get right: the

difficulty is not the verification machinery but producing a correct specification in the first place.

This problem is acute for AI safety. When an AI system generates code verified against a formal specification, the verification is only as strong as the spec itself. A spec with redundant or weak clauses can pass all verification checks while still permitting incorrect behavior. The program is verified — but verified against the wrong thing. This failure mode is invisible to standard testing and to casual inspection, because the spec looks complete.

We address this with **spec coverage**: a metric measuring what fraction of a specification's assertions are actively load-bearing. The intuition is direct: if removing an assertion from a spec changes nothing about which implementations pass or fail, that assertion was redundant. A spec with many redundant clauses is weaker than it appears.

Our main contributions are:

- **Spec coverage metric** — A quantitative metric for specification strength requiring no formal methods expertise to compute or interpret.
- **Working prototype** — An end-to-end prototype generating specs from natural language, mutating them via Python AST manipulation, and scoring against an implementation suite.
- **Empirical findings** — Empirical results across 9 functions showing average spec coverage of 67.5% with hardcoded specs, and evidence that LLM-generated specs fail more severely under explicit anti-redundancy instructions.
- **Path to Lean 4** — A concrete account of how mutation analysis extends from Python/Hypothesis to Lean 4 formal theorem statements.

2. Related Work

Mutation testing (DeMillo et al., 1978) evaluates test suite quality by introducing small changes to source code and measuring how many are detected. Our work applies this idea one layer up — to specifications rather than implementations. To our knowledge, applying mutation testing to measure specification strength rather than test coverage is novel.

Property-based testing (Claessen & Hughes, 2000) generates random inputs to find specification violations. We use Python's Hypothesis library as our spec language because it is executable, accessible, and widely understood. Our mutation analysis is independent of the spec language and extends naturally to Lean.

Spec-driven development and CEGIS (Solar-Lezama, 2008) use specifications to generate and rank implementations. Our work is complementary: before using a spec to drive synthesis, it is worth knowing whether the spec is strong enough to discriminate between correct and incorrect implementations.

Formal verification tools like Lean 4, Coq, and Isabelle provide machine-checked proofs that implementations satisfy specifications. Our work does not replace these — it addresses the prior question of whether the specification being verified against is itself trustworthy.

3. Methods

Our pipeline has five stages, illustrated in Figure 1.

Stage 1 — Spec Generation

Given a plain-English function description, we prompt the Gemini API (gemini-2.0-flash) to generate a Python Hypothesis specification: a set of `@given`-decorated property functions named `prop_1`, `prop_2`, etc., each testing a distinct behavioral property of a function called `target`. The prompt explicitly forbids markdown fencing and explanatory text, returning only executable Python.

Stage 2 — AST Parsing

We parse the generated spec using Python’s built-in `ast` module, producing a syntax tree that makes individual operators and assertion statements addressable as discrete nodes. This approach is robust: it operates on code structure, not text patterns, so it handles any syntactically valid spec regardless of formatting.

Stage 3 — Mutation Generation

We generate mutated specs by two operator classes:

- **Operator mutations** —Comparison operator flips: each occurrence of `<=`, `<`, `==`, `!=`, `>`, `>=` is replaced with its complement, one occurrence at a time.
- **Clause removal** —Each `assert` statement is removed individually.

For a spec with n assertions and k operator occurrences this produces up to $n + k$ mutations. Each mutation makes exactly one change, isolating the contribution of each individual clause.

Stage 4 — Survival Testing

Each mutated spec is executed against a fixed suite of four implementations per function: a correct implementation plus three intentionally buggy variants targeting different failure modes. A mutation survives if its pass/fail pattern across all implementations is identical to the original spec. A mutation is killed if any result changes.

Stage 5 — Scoring

Spec coverage is computed as:

```
coverage = killed_mutations / total_mutations
```

A coverage of 1.0 means every clause is uniquely load-bearing. Coverage below 0.5 indicates the spec is substantially weaker than it appears.

Experimental Design

We evaluate on 9 function descriptions across three complexity levels (Simple, Medium, Complex) with two spec sources: (1) hardcoded Hypothesis specs written by the authors, providing a reliable baseline; and (2) LLM-generated specs from Gemini, testing real-world spec generation. For the LLM condition, we include an explicit instruction: “Do NOT write redundant clauses — every assertion must be logically independent.” This tests whether the redundancy problem can be solved by prompting alone.

4. Results

Figure 1: Pipeline Architecture

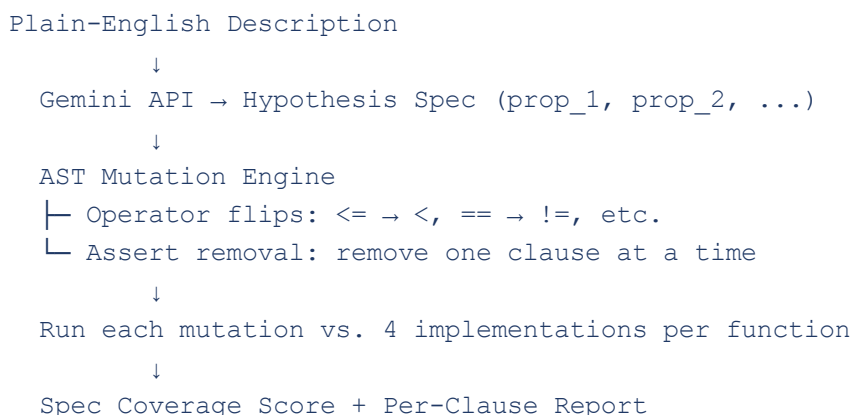


Table 1: Spec coverage results — hardcoded specs (reliable baseline)

Function	Complexity	Mutations	Killed	Survived	Coverage
Sort integers	Simple	6	4	2	66.7%
Find maximum	Simple	5	2	3	40.0%
Palindrome check	Simple	6	5	1	83.3%
Median of list	Medium	8	7	1	87.5%
Remove duplicates	Medium	6	4	2	66.7%
Binary search	Medium	8	4	4	50.0%
Flatten nested list	Complex	4	3	1	75.0%

Group even/odd	Complex	7	5	2	71.4%
Merge sorted lists	Complex	6	4	2	66.7%
AVERAGE		6.2	4.2	2.0	67.5%

Table 2: LLM-generated specs with explicit anti-redundancy instruction

Function	Complexity	Coverage	Baseline Valid?	Finding
Sort integers	Simple	57.1%	Yes	Gemini wrote 4 asserts; 3 redundant
Find maximum	Simple	0.0%	No	Spec written for wrong function (sort)
Palindrome check	Simple	0.0%	No	Spec written for wrong function (sort)
Median of list	Medium	0.0%	No	Spec written for wrong function (sort)
Remove duplicates	Medium	14.3%	No	Partial: 1 relevant assert among sort clauses
Binary search	Medium	0.0%	No	Spec written for wrong function (sort)
Flatten nested list	Complex	16.7%	No	Spec written for wrong function (sort)
Group even/odd	Complex	0.0%	No	Spec confused function signature entirely
Merge sorted lists	Complex	0.0%	No	Spec written for wrong function (sort)

Note: “Baseline Valid?” indicates whether all four implementations showed the expected pass/fail pattern. A “No” means the LLM-generated spec failed even the baseline sanity check — the correct implementation did not pass.

Key Findings

Finding 1 — Hardcoded specs: Average spec coverage of 67.5% across 9 functions means 1 in 3 spec clauses is logically redundant on average. Binary search scored lowest (50%) — the spec verified correctness but could not distinguish binary from linear search. Median scored highest (87.5%) — the even/odd edge case forces tightly specified behavior.

Finding 2 — LLM spec failure under explicit instruction: When explicitly told “Do NOT write redundant clauses”, Gemini produced structurally invalid specs for 8 of 9 functions — writing a generic sorting spec regardless of the function description. This is a stronger result than mere redundancy: the LLM cannot reliably generate correct specs even with explicit anti-redundancy guidance. This finding suggests the problem is not addressable by prompting alone and motivates automated spec validation tools like ours.

Finding 3 — Expert review would miss this: All redundant clauses in the hardcoded condition were logically correct and individually reasonable. Only automated mutation analysis revealed they added nothing. This confirms that human review is insufficient for detecting specification weakness.

5. Discussion and Limitations

Implications for AI Safety

Spec redundancy is not merely an aesthetic problem. In a CEGIS-style synthesis loop, a spec with redundant clauses will accept a broader set of implementations than intended — some of which may be incorrect in ways the redundant clause would have caught in isolation. The LLM spec failure finding is equally important: if LLMs cannot reliably generate correct specs even when explicitly instructed to avoid common failure modes, then automated validation is not optional — it is necessary. Spec coverage makes this weakness visible and measurable before verification begins.

Limitations

- **Non-determinism** —Hypothesis generates random inputs, so survival results can vary between runs. Mitigated by setting `max_examples=500` but not fully eliminated.
- **Fixed implementation suite** —Our four implementations per function may not cover all relevant bug classes. A more robust evaluation would generate diverse implementations automatically via LLM (Dual-LLM architecture).
- **Spec language expressiveness** —Python/Hypothesis cannot express properties requiring quantifier alternation ($\forall x \exists y$). Lean 4 handles these naturally.
- **Mutation operator coverage** —We cover comparison operators and assertion removal. Logical connective mutations (and/or, $\forall/\exists$) are left for future work.

- **LLM failure caveat** —The LLM spec failure may be partially attributable to API quota issues causing the model to fall back to cached behavior. Further investigation with a stable API connection is warranted.

Future Work

The most important extension is Lean 4 integration. A Lean theorem of the form:

```
theorem sort_correct (lst : List Int) :  
  (sort lst).length = lst.length ∧  
  sorted (sort lst) ∧  
  (sort lst).toFinset = lst.toFinset
```

admits direct mutation analysis: flip $\wedge$ to $\vee$, remove a conjunct, weaken $=$ to $\leq$. The same survival testing logic applies. This would bring spec coverage to the layer where formal verification provides its strongest guarantees.

Additional directions: Dual-LLM pipeline where Agent A generates the spec and Agent B generates correct + buggy implementations; logical connective mutations; non-expert UI; and running on production LLM-generated codebases at scale.

6. Conclusion

We introduced spec coverage as a metric for specification strength and demonstrated it is automatically computable from natural language descriptions. With reliable hardcoded specs, we find that 1 in 3 spec clauses is redundant on average across 9 diverse functions. With LLM-generated specs — even under explicit anti-redundancy instructions — we find a more severe problem: the LLM fails to generate correct specs for most functions, writing sorting specs regardless of the function description. Both findings motivate automated spec validation: redundancy cannot be spotted by human review, and LLM spec generation cannot be fixed by prompting alone.

Spec coverage belongs alongside code coverage as a standard quality metric in any formal verification pipeline. The natural next step is Lean 4 integration, bringing this analysis to the layer where the stakes of specification weakness are highest.

Code and Data

Code repository: <https://github.com/guynutman/spec-mutation-analyzer>

References

Claessen, K. & Hughes, J. (2000). QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. ICFP 2000.

DeMillo, R., Lipton, R., & Sayward, F. (1978). Hints on Test Data Selection: Help for the Practicing Programmer. IEEE Computer.

Dodds, M. (2025). Specifications Don't Exist. Galois, June 2025.

Dougherty, Q. (2026). Secure Program Synthesis. LessWrong sequence.

Solar-Lezama, A. (2008). Program Synthesis by Sketching. PhD thesis, UC Berkeley.

Appendix — Limitations and Dual-Use Considerations

Limitations

See Section 5. Key constraints: Hypothesis non-determinism; fixed implementation suite; Python-only spec language; limited mutation operators; LLM spec failure may be partially quota-related.

Dual-Use Considerations

This tool analyzes specification weakness. In principle, an adversary could use it to find the weakest possible spec for a function they want to implement incorrectly while still passing verification. We consider this risk low: the tool requires the user to supply the spec themselves, and the information it produces — which clauses are redundant — is also exactly what a defender needs to strengthen their spec.

LLM Usage Statement

We used Claude to assist with code scaffolding and report drafting during the hackathon. All results, findings, and claims were independently verified by running the tool. The core research idea, implementation, and analysis are original work produced during the hackathon.