

Proof Assistance as Verification Oracle for Porting

Abhinav Chand

Balaji R

With Apart Research

Abstract

As AI agents take on increasing shares of large-scale software porting work, the need for scalable semantic verification grows urgent. This paper presents a three-stage harness pipeline in which Claude Code uses Dafny as a formal semantic lens to detect divergences between AI-generated Rust ports and their Zig source implementations. In Stage 1 the target Rust function is translated mechanically into Dafny; in Stage 2 behavioral theorems are discovered from the Zig source; and in the Stage 3 stage, every discovered theorem is attempted against the Rust-derived Dafny model. Each proof result is classified as proved, likely bug/divergence, missing Dafny spec, proof gap, ambiguous theorem, or unknown, producing a per-theorem semantic audit of the port. This approach identified real bugs through genuine Dafny verification failures: in both cases the verifier rejected a Zig-derived postcondition against the Rust-based model, yielding a proof-driven divergence signal rather than a heuristic code review. [These bugs were reported](#). The dominant failure mode is insufficient model coverage: when the relevant behavior is absent from the Dafny model, no proof failure can occur. These results demonstrate that formal verification is a viable signal source for AI safety in automated porting and suggest that scalable model coverage is the key engineering challenge for operationalizing the approach.

1. Introduction

As software systems evolve, porting is an essential process that keeps these systems adaptable, maintainable, secure and scalable. Yet porting is inherently risky: even careful, manual translations can introduce subtle bugs, break behavioral invariants, or silently alter semantics in ways that only surface under specific conditions.

A recent [pull request](#) to Anthropic's Bun repository with a Rust port made headlines in the developer tabloids [\[1\]](#). The merge took place shortly after, adding more than a million lines of code written by AI. In the past, this kind of translation would usually take years of effort from teams of developers, testing every aspect of the port. As AI agents take on an increasing share of the porting work, it has become important to introduce formal verification in the pipeline, not least because human review is not scalable in this regime.

In this project we create a harness for AI agents to use Dafny to check whether the Rust port faithfully preserves the semantics of the Zig source. In this harness, the AI agent translates a Rust function into Dafny, derives behavioral theorems from the Zig source code, and then attempts to prove those theorems against the Dafny model. If a theorem derived from Zig cannot be proved against the Rust-based Dafny model, that is evidence of a semantic divergence — i.e. a bug in the Rust port.

2. Related Work

A growing body of work investigates using LLMs to produce or assist with formally verified code. Clover [\[2\]](#) proposes a closed-loop pipeline where LLMs generate both code and Dafny specifications, and a consistency checker flags mismatches. More directly related to our setting, work on AI-assisted synthesis of verified Dafny methods [\[3\]](#) empirically evaluates LLM performance at generating Dafny postconditions and invariants, while PREFACE couples LLMs with a reinforcement learning agent to iteratively repair Dafny code until the verifier accepts it. Our approach differs from all of these: rather than generating verified code from scratch, we use Dafny as a semantic lens to check whether an already-generated translation is faithful to its source.

3. Methods

The harness targets a Rust port and its corresponding Zig source implementation in the Bun repository. A run is invoked with a Rust file and corresponding Zig file. The Rust file is treated as the implementation reference, while the Zig file is treated as the semantic reference.

The pipeline runs three stages. Each stage is a separate invocation of Claude Code with a specialized system prompt. The Bun repository is used as the working directory so Claude can read and modify source files directly. All invocations run with filesystem access and emit transcripts and stage outputs for later inspection.

Stage 1 — Rust → Dafny Port

- **Input:** The full content of the target Rust file plus the function name
- **Output:** A Dafny file written to disk next to the Rust source (same name, .dfy extension)

The stage 1 prompt instructs Claude to produce a mechanical Dafny model of the Rust code. The goal is not to repair or redesign the implementation, but to model what the Rust code actually does. The port preserves the Rust function’s control flow, local variable structure, branch conditions, loop structure, and return-value behavior as closely as possible. This stage establishes the Rust-derived semantic model that later proof attempts will target. If the Rust port contains a semantic divergence from Zig, that divergence should be reflected in the Dafny model rather than silently corrected during translation.

Stage 2 — Theorem Discovery from Zig

- **Input:** The full content of the target Zig file
- **Output:** A numbered list of theorem packets

The stage 2 prompt instructs Claude to treat the Zig source as the sole source of truth and enumerate its full behavior space as discrete theorem packets. It is explicitly forbidden from consulting the Dafny model at this stage — all claims must be derived from Zig alone. This decoupling is deliberate: if stage 2 could see the Dafny model it would tend to produce theorems the model already satisfies, defeating the purpose.

Claude reads the Zig source (the ground truth) and enumerates the full behavior space as a discrete theorem. Each theorem captures one behavioral claim and includes:

- `theorem_id`
- `source_file` / `source_reference` — exact Zig file and line numbers
- `statement` — the behavioral claim in plain English
- `preconditions` — what must be true for the theorem to apply
- `expected_behavior` — what the Zig code does under those preconditions
- `edge_cases_covered` — specific inputs or states the theorem exercises
- `why_this_is_zig_derived` — which Zig lines or constructs justify the claim
- `ambiguities_or_assumptions` — anything unclear or assumed

The theorem discovery stage is intentionally decoupled from the Dafny model so that the theorems represent what Zig actually does, not what the Dafny model already happens to encode.

Stage 3 — Theorem Proving

- **Input:** The theorems from stage 3
- **Output:** A structured proof report

The stage 3 prompt instructs Claude to attempt to prove the theorem against the fixed Dafny model using only proof-level additions — preconditions, postconditions, assertions, loop invariants, framing, and helper lemmas. It must not modify the Dafny method body, weaken the theorem to make it pass, or replace a failed claim with a different one. If verification fails after adding a Zig-derived obligation, that failure is the signal: the prompt instructs Claude to treat it as evidence of a divergence rather than a deficiency in the proof scaffolding. The core rules:

- The Dafny method body must not be changed (except to complete an obvious unimplemented stub from Rust).
- Proof work is limited to adding preconditions, postconditions, assertions, loop invariants, framing, and helper lemmas.
- Claude runs `dafny verify` and reads the verifier output.
- If verification fails after adding a Zig-derived postcondition, that is classified as a divergence signal.

Failure is classified into one of four categories:

Classification	Meaning
<code>likely bug</code> / <code>divergence</code>	Dafny verifier rejected a Zig-derived postcondition — the model's behavior is formally incompatible with Zig

missing Dafny spec	The relevant function or type is absent from the Dafny model; the theorem cannot be tested
proof gap	The theorem is well-formed but the proof scaffolding is insufficient to verify it; not necessarily a bug
ambiguous theorem	The theorem statement is unclear or contradictory; cannot attempt proof

The key design principle: failure to prove is evidence, not just a gap. If the baseline Dafny model verifies cleanly (as it should, since it was written to mirror the Rust code), but adding a Zig-derived obligation causes a verifier error, that error points to where the Rust port diverges from Zig.

4. Results

The theorem harness identified real bugs through genuine Dafny verification failures. In these cases, the verifier rejected Zig-derived postconditions against the Rust-derived Dafny model, producing a proof-driven divergence signal rather than a heuristic code review finding.

[Two such bugs were identified, reported, and had real-world impact.](#) These results show that the approach can surface meaningful semantic mismatches in production porting work. The main limitation remains model coverage: when the relevant behavior is absent from the generated Dafny model, the harness cannot produce a proof failure for that behavior.

5. Discussion and Limitations

Formal verification will not solve the alignment problem by itself, and it is not a universal fix for AI safety. But these results suggest it can significantly reduce the attack surface of AI-generated code. The analogy is like replacing a wooden door with a steel door: it does not guarantee the house cannot be robbed, but it makes failure harder. Historically, functional programming and programming-languages research have developed powerful tools for avoiding whole classes of bugs, but the bottleneck has often been implementation difficulty and developer effort. With AI assistance, that bottleneck may shrink.

Future Work

The main limitation we observed is model coverage: if the Dafny translation does not include the behavior where the bug occurs, the verifier has nothing meaningful to reject. In future work, we plan to experiment with integrating our harness with [Aeneas](#), which translates Rust into verification-oriented representations. This may improve the coverage and fidelity of the Rust-derived formal model, reducing the amount of manual or agent-generated modeling needed before theorem proving can begin. If successful, this would make the harness more scalable and better suited for larger real-world ports.

6. Conclusion

This project shows that formal verification can be a practical safety layer for AI-generated ports. By proving Zig-derived theorems against Rust-derived Dafny models, the harness found real bugs through verifier failures rather than heuristic review. The main limitation is model coverage: the verifier can only catch behavior that is represented in the model. Overall, the results suggest that proof-driven auditing can make AI-generated code more trustworthy.

References

1. Anderson, T. (2026, May 14). Anthropic's Bun Rust rewrite merged at speed of AI. The Register.
<https://www.theregister.com/devops/2026/05/14/anthropics-bun-rust-rewrite-merged-at-speed-of-ai/5240381>
2. Sun, C., Sheng, Y., Padon, O., & Barrett, C. (2023). Clover: Closed-loop verifiable code generation (arXiv:2310.17807). arXiv. <https://doi.org/10.48550/arXiv.2310.17807>
3. Misu, M. R. H., Lopes, C. V., Ma, I., & Noble, J. (2024). Towards AI-assisted synthesis of verified Dafny methods. Proceedings of the ACM on Software Engineering, 1(FSE).
<https://doi.org/10.1145/3643763>

LLM Usage Statement

We used Claude to write the major portions of the report.