ProofPulse: Interactive Proof Coverage Analysis for Dafny

Álvaro F. Silva✉
INESC TEC, Faculdade de Engenharia, Universidade do Porto
Porto, Portugal
up201603524@fe.up.pt

Ruben Martins
Carnegie Mellon University
Computer Science Department
Pittsburgh, USA
rubenm@andrew.cmu.edu

Alexandra Mendes*
INESC TEC, Faculdade de Engenharia, Universidade do Porto
Porto, Portugal
alexandra@archimendes.com

Abstract

Deductive verification ensures that an implementation satisfies its specification, but successful verification does not guarantee the quality of the specification. As such, weak specifications and redundant invariants may create overconfidence in "verified" code.

We present ProofPulse, a tool for Dafny that diagnoses specification quality using a three-valued proof coverage model. By analyzing proof dependencies, ProofPulse distinguishes between elements that contribute to specification intent, those used only for auxiliary checks, and those irrelevant to any proof obligation.

Evaluated against an oracle of 252 programs from the dafny-synthesis benchmark, ProofPulse provides a high-precision signal for specification weaknesses, particularly unnecessary preconditions and vacuous proofs. With unsat-core minimization, ProofPulse achieves perfect precision for precondition classification and reduces false positives across all evaluated categories. These results show that proof coverage is a practical complement to verification. Although it cannot fully capture semantic intent, it can reveal weak proof coupling in programs that otherwise appear fully verified.

Just as a pulse check distinguishes vitality from the mere absence of symptoms, ProofPulse exposes weaknesses in proofs that technically verify successfully but still suffer from inadequate or redundant code and specifications.

Demo: https://www.youtube.com/watch?v=8pO3NAodjoQ
Code: https://github.com/VeriFixer/ProofPulse
Prebuilt Docker image: https://doi.org/10.5281/zenodo.21174686

CCS Concepts

• **Theory of computation** → **Logic and verification**; • **Software and its engineering** → *Formal software verification.*

Keywords

Proof Coverage, Dafny, Specification Quality, Verification Tools

1 Introduction

Deductive software verification tools, such as Dafny [5], ensure that implementations satisfy their specifications. However, a successful verification only proves that the implementation satisfies the *given* specification, it does not ensure that the contract is adequately coupled to the code it governs. Weak postconditions, redundant invariants, or unnecessary preconditions may allow verification to succeed while large parts of the program remain unconstrained. In the extreme case, the Dafny verifier may report a "correct" status for a program that lacks verification conditions. In such instances, the IDE provides a passing "tick" despite the absence of meaningful relationship between the specification and the code, creating confidence in an implementation that remains effectively unverified.

Tomb and Joshi [9] introduced *verification coverage*, using unsatisfiable cores to identify which program elements are relevant to a proof. While effective, their approach is binary (covered/uncovered) and primarily exposed via command-line output. Such output can be difficult to inspect, since developers must manually connect coverage information back to source locations and proof dependencies. This makes it hard to distinguish elements that establish the main specification from elements used only for auxiliary checks, such as bounds checks or intermediate assertions.

We present **ProofPulse**, a tool that makes proof coverage more interactive and informative, while integrating it into the IDE. Our key insight is that not all "covered" elements play the same role: some contribute directly to the specification, while others only support auxiliary checks. Our main **contributions** are:

- **ProofPulse VSCode Extension**: An interactive tool (VSCode + web viewer) for exploring proof dependencies, designed for Dafny users across all levels of expertise.
- **Three-valued coverage mapping**: A coverage model distinguishing essential, auxiliary, and irrelevant proof elements.
- **Z3 unsat-core minimization**: An optional unsatisfiable core minimization layer over Z3 that improves attribution quality.
- **Empirical Evaluation**: Evaluation on 252 programs showing that ProofPulse provides a high-precision signal for unnecessary preconditions, vacuous proofs, and weak proof coupling.

2 Proof Coverage

ProofPulse refines binary coverage into a three-value proof relevance model computed from Dafny's dependency information.

From Binary to Three-Valued Coverage. Tomb and Joshi [9] define *verification coverage* as a semantic property of proof relevance: a program element is *covered* if modifying it may cause verification to fail, and *uncovered* if it can be changed arbitrarily without affecting the proof. Their approach instruments Boogie's verification condition (VC) generation with labels on assertions and assumptions. After discharging the VC, the unsatisfiable core returned by the SMT solver identifies the covered elements.

In addition, they provide a semantic interpretation of *uncovered* elements depending on their role in the verification process:

Assignments or calls: unconstrained or unreachable code.
Assumptions: unnecessary assumptions, including `assume` statements or preconditions of the verified procedure.
Callee preconditions: potentially vacuous calls.
Callee postconditions: assumptions not required by the caller.
Procedure postconditions: vacuously proved guarantees.

*Author order follows a senior-author-last convention.

```
1  method CylinderVolume(radius :real, height: real)
2    returns (volume: real)
3    requires radius >= 0.0 && height >= 0.0
4    ensures volume >= 0.0
5  {volume := 3.14 * radius * radius * height;}
6
7  method main(){
8    var a := CylinderVolume(3.0,4.0);
9    //assert(a >= 0.0);
10 }
```

Figure 1: ProofPulse VSCode extension. The precondition `radius >= 0.0` is classified as `CovTest` (yellow highlight).

Assertions: vacuous goals or unused intermediate proof steps.
Loop invariants: vacuous proofs when proved, or unnecessary assumptions when used in the loop body.

ProofPulse adopts the same semantic view of proof relevance. Uncovered elements are those that do not contribute to any proof obligation, matching Tomb and Joshi's notion of irrelevance. However, ProofPulse refines their binary covered/uncovered classification by separating fully irrelevant elements from elements that are relevant only to auxiliary obligations.

Some elements are verified and used by the proof, but not to establish the main specification. For example, a callee specification may be checked or available at a call site without being needed to prove the caller's postcondition. Rather than reporting such elements as fully uncovered, ProofPulse marks them as CovTest, indicating that they support verification but are not essential to the primary proof objective. As a result, ProofPulse preserves the core semantic insight of prior work while providing a more fine-grained interpretation of proof relevance.

Coverage Computation. ProofPulse computes coverage in two phases. First, it builds a proof dependency graph and assigns a uniform *internal* coverage status to every node via breadth-first search propagation. Second, a type-aware *refinement* step maps internal statuses to a final three-valued classification depending on each node's semantic role.

Proof Dependency Graph. From a Dafny source file and verification log, ProofPulse constructs a directed graph $G = (V, E, T)$ where:

- Each node $v \in V$ corresponds to a source-level element identified by its file location span.
- Each directed edge $(u, v) \in E$ indicates that v was used to discharge the proof obligation of u.
- $T \subseteq V$ is the set of *top nodes*: the proof obligations that Dafny reports (postconditions, manual assertions, automatic assertions such as index-in-range checks).

Each node is classified by a *token type* $\tau(v) \in \{$`Postcondition`, `Precondition`, `AssertManual`, `AssertAuto`, `CodeLine`$\}$, inferred from the proof message text. Every node starts as `Uncovered`. The algorithm then propagates coverage from top nodes downward through the dependency edges. After these passes, every node v has an internal status $\sigma_{int}(v) \in \{$`CovComplete`, `CovTest`, `Uncovered`$\}$. A node is marked `CovComplete` when it participates in proving a postcondition, `CovTest` when it participates only in proving non-postcondition obligations, such as index safety or manual assertions, and `Uncovered` when no proof obligation depends on it. The graph can be navigated interactively in the ProofPulse web viewer.

Type-Aware Refinement. Statuses are further refined based on the element type. The mapping resembles the one proposed by Tomb and Joshi, differing relative to the `CovTest` cases:

(1) When a *postcondition* is never needed by any callee, either because the method is never called or because the postcondition is not used to prove any property in the callee, we mark it as `CovTest` instead of `Uncovered`. This distinguishes it from the stronger case in which the postcondition is proved vacuously without relying on any code line or lemma in the method body. In the example of Figure 1, uncommenting line 9 would cause the postcondition to be classified as `CovComplete`.
(2) A *precondition* is marked as `CovTest` if it is used by a callee, but is not strictly necessary to prove the method's postcondition. This distinction is useful because some preconditions exist primarily to constrain caller behavior rather than to serve as requirements for establishing the postcondition, as illustrated in Figure 1. If line 8 were commented out, the precondition would instead be classified as `Uncovered`, since it would neither be exercised by callers nor required to prove the postcondition.

The introduction of `CovTest` addresses cases that would otherwise be misleadingly classified as either `Uncovered` or `CovComplete`. In Figure 1, the precondition `radius >= 0.0` is classified as `CovTest` because it is exercised by the caller at line 8, yet it is unnecessary for proving the postcondition, since `radius * radius` is always non-negative. Marking it as `Uncovered` would incorrectly suggest that the precondition is useless, despite its important semantic role in constraining caller behavior (i.e., in preventing negative radius values), while marking it as `CovComplete` would hide the fact that it could potentially be weakened or removed without affecting the proof. Thus, `CovTest` acts as an intermediate warning category, highlighting specifications that participate in verification but may deserve further review or simplification.

3 ProofPulse Architecture

ProofPulse is implemented in TypeScript with two packages: a shared core library (`@proofpulse/core`) and a VSCode extension. The pipeline, presented in Figure 2, operates in four stages:

1. Verification with coverage logging. ProofPulse invokes Dafny with the `--verification-coverage-report` flag, which enables the coverage instrumentation described by Tomb and Joshi [9]. Dafny then produces a prover log with proof dependencies.

2. Log parsing and graph construction. The core library parses the prover log and constructs the proof dependency graph. Each source span becomes a node and dependency relationships become directed edges.

3. Coverage computation. The graph is traversed to assign coverage statuses. The algorithm processes top-level obligations first, then propagates status through the dependency edges.

4. Visualization. Two front-ends consume the proof graph:

- *VSCode extension*: Provides gutter decorations (red for `Uncovered`, yellow for `CovTest`), and hover diagnostics showing the proof message and coverage status (Figure 1).
- *Web viewer*: A two-column layout with a tokenized source editor with clickable spans, and a detail panel for exploring dependencies. It can be activated through the VSCode extension (Figure 3).

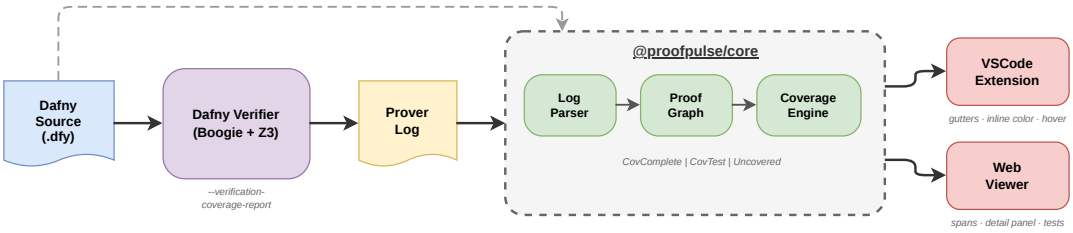

Figure 2: ProofPulse pipeline. The Dafny verifier produces a prover log with proof dependencies. The core library parses this into a proof dependency graph, computes three-valued coverage, and feeds two visualization front-ends.

Table 1: Comparison of ProofPulse classification performance of the base configuration (Base) and core minimization (Min). The table reports confusion-matrix counts (True Positive - TP, False Positive - FP, False Negative - FN, True Negative - TN) together with precision (Prec), recall (Rec), and accuracy (Acc)

Cat.	Cfg	TP	FP	FN	TN	Prec	Rec	Acc
Post.	Base	184	24	0	11	0.88	1.00	0.89
	Min	184	21	0	13	0.90	1.00	0.90
Pre.	Base	63	3	7	96	0.95	0.90	0.94
	Min	63	0	6	98	1.00	0.91	0.96
Inv.	Base	77	13	0	11	0.86	1.00	0.87
	Min	77	10	0	13	0.89	1.00	0.90

4 Evaluation

We evaluate whether proof coverage is a useful practical proxy for specification quality by comparing ProofPulse against the manually curated oracle from the `dafny-synthesis` benchmark [7], which contains 252 verified Dafny programs. Proof coverage and semantic specification strength are related but distinct notions. A specification is semantically weak when it under-specifies intended behavior. This weakness often manifests itself as weak proof coupling, where specifications, invariants, or code do not contribute meaningfully to verification. Although coverage cannot determine whether a specification fully captures program intent, it can reveal redundancy, vacuity, and under-constrained proofs. We therefore evaluate how well coverage signals align with human judgments of specification quality.

4.1 Benchmark and Oracle

The `dafny-synthesis` benchmark [7] contains Dafny programs generated by GPT-4 and PaLM-2 under multiple prompting strategies. In Misu et al. [7], each verified program is manually annotated with labels for postcondition strength (Strong/Weak/Wrong), precondition necessity (Required/Optional), and loop invariant strength (Strong / Weak). These oracle labels are semantic judgments, whereas ProofPulse measures proof coupling. For example, an oracle-Strong postcondition captures intended behavior, while a covered postcondition only means that the proof depends on it. We map ProofPulse's statuses to Oracle's categories as follows:

- *Postconditions*: Strong if all postconditions are covered (`CovTest` or `CovComplete`) and all body code lines are `CovComplete`; Weak otherwise. If parts of the code are not required for proving postconditions, this indicates either under-constrained specifications or irrelevant code.
- *Preconditions*: Required if any precondition is covered; Optional if all preconditions are `Uncovered`. Uncovered preconditions can be safely removed. Hence, they are Optional.
- *Invariants*: Strong if all loop invariant nodes are covered; Weak if any is `Uncovered`. Same reasoning as postconditions.

4.2 Results

Effect of Unsat-Core Minimization. During the development of ProofPulse, we observed that non-minimal Z3 unsatisfiable cores can affect proof-coverage attribution by introducing unnecessary dependencies. Tomb and Joshi [9] similarly note that SMT solvers do not guarantee minimal unsat cores. To mitigate this issue, ProofPulse optionally applies deletion-based unsat-core minimization [6] before coverage classification. As shown in Table 1, minimization reduces false positives across all categories (e.g., from 24 to 21 for postconditions) and consistently improves precision, while preserving recall. In our benchmark, the minimized cores were small enough that minimization added little overhead while remaining practical for interactive use.

Result Analysis. As shown in Table 1, ProofPulse performs particularly well for preconditions, achieving perfect precision and 0.96 accuracy with core minimization. This aligns with the underlying semantics: an uncovered precondition is genuinely unnecessary for verification, closely matching the oracle's Optional label.

Postcondition precision is also high (0.90), indicating that ProofPulse's coverage-based Strong classifications usually agree with the oracle. Precision stays below 1.00 because a postcondition can be fully covered, with every line exercised, and still be weak, in which case ProofPulse classifies it as Strong while the oracle labels it weak. These fully-covered-but-weak postconditions are the false positives that lower precision. Figure 1 illustrates one such case, where every line of `CylinderVolume` is covered and exercised, yet the postcondition remains weak. Postcondition recall is perfect (1.00). When ProofPulse classifies a postcondition as weak, it is almost certainly weak. A weak flag means that some behavior of the implementation is left unconstrained by the postcondition. The exception is dead code: if unreachable code contains behavior the postcondition does not pin down, ProofPulse still flags the postcondition as weak, even though the specification could be strong, since

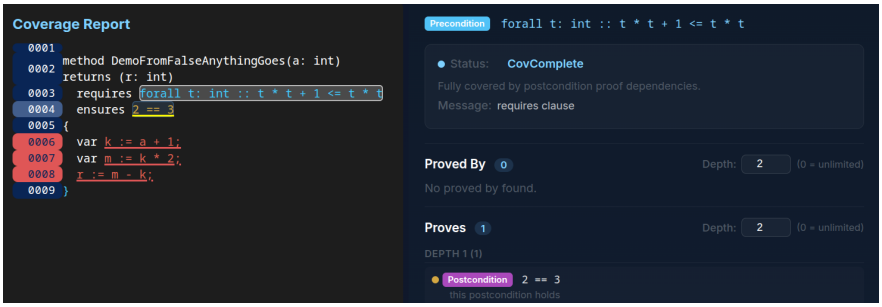

Figure 3: ProofPulse web viewer showing a Dafny method with an unsatisfiable precondition. On the left, the code is displayed with red underlining indicating `Uncovered` elements, yellow highlighting indicating `CovTest`, and blue highlighting indicating the currently selected item. On the right, more detailed information about the selected item is shown, including its status, proof dependencies, and the elements that prove it.

that code can never execute. Such a case would be a false negative and would lower recall. In our evaluation no such cases occurred, so no truly strong postcondition was ever flagged as weak, leading to a perfect recall.

Invariant accuracy closely follows the behavior observed for postcondition classification. This is expected, since both invariants and postconditions constrain program behavior in a similar way. Loop invariants additionally have a dual role: they act as both preconditions and postconditions at different program points.

ProofPulse additionally detects vacuous proofs arising from contradictory assumptions. Figure 3 shows a method with an unsatisfiable `forall` precondition, where the method body is marked as `Uncovered` (marked in red). The web viewer's proof graph further reveals that the contradictory `requires` clause alone suffices to discharge the postcondition (`2 == 3`), exposing the vacuous reasoning.

Limitations. ProofPulse's main limitation is imprecision in the Dafny-to-Boogie-to-Z3 pipeline, especially the lossy mapping from Boogie IVL back to Dafny source spans. The major problem is:

- *Quantified expressions (`forall`).* Dafny encodes quantified specifications as Boogie axioms with SMT triggers. These triggers are not part of the core logical proof and do not appear in Z3's unsat core, so the corresponding source lines are always reported as uncovered, even when they are necessary for verification — reflecting a limitation in quantifier coverage tracking.

These are engineering limitations of the Dafny/Boogie pipeline rather than the coverage approach itself. Improved source tracking would directly improve ProofPulse.

5 Related Work

Verification coverage. Tomb and Joshi [9] formalize static coverage for deductive verification using unsatisfiable cores, building on the coverage–vacuity duality from model checking [3]. ProofPulse builds directly on their Boogie/Dafny implementation, extending it with a three-valued formalism, a proof dependency graph, and interactive tooling. Ghassabani et al. [1, 2] describe inductive validity cores for unbounded model checking and use them to measure implementation coverage.

Specification quality. Le et al. [4] combine proof and test coverage using mutation-based metrics. Smoke testing [9] inserts `assert false` at program points to detect vacuity, but produces many warnings and does not identify redundant specifications. ProofPulse provides finer-grained feedback with fewer false positives.

Autoformalization evaluation. Misu et al. [7] evaluate LLM-generated Dafny specifications and provide the oracle used by us. Their manual classification of specification strength motivates automated approaches like ProofPulse for scalable quality assessment.

Verification IDEs. Dafny's built-in IDE support provides verification status (verified/error) but does not expose coverage information. The Dafny VSCode extension shows verification errors inline. ProofPulse complements this by showing *what the proof used*, not just whether it succeeded.

Usability. Oliveira et al. [8] studied the challenges practitioners face when using verification-aware languages. Their findings highlight the need for improved usability feedback and greater access to the internal reasoning of verification tools. The design of ProofPulse is motivated by these challenges, providing detailed verification feedback and exposing proof dependencies in an accessible way.

6 Conclusion and Future Work

We presented ProofPulse, a tool for interactive proof coverage analysis of Dafny programs. By constructing a proof dependency graph and applying a three-valued coverage formalism, ProofPulse provides actionable feedback about specification quality directly within the developer's IDE. We also extend prior work with an optional deletion-based unsat-core minimization, which improves attribution precision by reducing spurious proof dependencies introduced by non-minimal SMT unsat cores.

Our evaluation on 252 LLM-generated Dafny programs demonstrates perfect precision for precondition classification and promising results for postconditions and invariants. Enabling unsat-core minimization further reduced false positives across all evaluated categories while preserving recall. The remaining limitations can largely be traced to Boogie-to-Dafny attribution gaps rather than fundamental issues with the approach itself.

Future work includes: (1) finer-grained sub-expression-level attribution, (2) user studies measuring the impact on proof debugging time and specification understanding, (3) extension to other verification-aware languages, and (4) improved quantifier handling.

Data Availability Statement

The data associated with this work are publicly available through a Zenodo repository [10] and can be accessed via the following DOI: https://doi.org/10.5281/zenodo.21174686

Acknowledgments

Alexandra Mendes was partially funded by National Funds through the FCT - Fundação para a Ciência e a Tecnologia, I.P. (Portuguese Foundation for Science and Technology) within the project VeriFixer, with reference 2023.15557.PEX (DOI: 10.54499/2023.15557.PEX) and by an Amazon Research Award, Fall 2024.

Ruben Martins was partially supported by the National Science Foundation (NSF) under Award CCF2427581 and DARPA Agreement FA8750-24-9-1000.

Álvaro Silva was co-financed by national funds through FCT – Fundação para a Ciência e a Tecnologia, I.P., under the support UID/50014/2025 (https://doi.org/10.54499/UID/50014/2025), by National Funds through the FCT - Fundação para a Ciência e a Tecnologia, I.P. (Portuguese Foundation for Science and Technology) within the project VeriFixer, with reference 2023.15557.PEX (DOI: 10.54499/2023.15557.PEX), and Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) through the Carnegie Mellon Portugal Program under the fellowship reference PRT/BD/155045/2024.

References

[1] Elaheh Ghassabani, Andrew Gacek, and Michael W. Whalen. 2016. Efficient generation of inductive validity cores for safety properties. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 314–325. doi:10.1145/2950290.2950346

[2] Elaheh Ghassabani, Andrew Gacek, Michael W. Whalen, Mats P. E. Heimdahl, and Lucas Wagner. 2017. Proof-based coverage metrics for formal verification. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 194–199. doi:10.1109/ASE.2017.8115632

[3] Orna Kupferman, Wenchao Li, and Sanjit A. Seshia. 2008. A Theory of Mutations with Applications to Vacuity, Coverage, and Fault Tolerance. In *2008 Formal Methods in Computer-Aided Design*. 1–9. doi:10.1109/FMCAD.2008.ECP.29

[4] Viet Hoang Le, Loïc Correnson, Julien Signoles, and Virginie Wiels. 2018. Verification Coverage for Combining Test and Proof. In *Tests and Proofs*, Catherine Dubois and Burkhart Wolff (Eds.). Springer International Publishing, Cham, 120–138. doi:10.1007/978-3-319-92994-1_7

[5] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 348–370. doi:10.1007/978-3-642-17511-4_20

[6] Joao Marques-Silva and Ines Lynce. 2011. On Improving MUS Extraction Algorithms. In *Theory and Applications of Satisfiability Testing - SAT 2011*, Karem A. Sakallah and Laurent Simon (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 159–173. doi:10.1007/978-3-642-21581-0_14

[7] Md Rakib Hossain Misu, Cristina V. Lopes, Iris Ma, and James Noble. 2024. Towards AI-Assisted Synthesis of Verified Dafny Methods. *Proc. ACM Softw. Eng.* 1, FSE, Article 37 (July 2024), 24 pages. doi:10.1145/3643763

[8] Francisco Oliveira, Alexandra Mendes, and Carolina Carreira. 2025. What Challenges Do Developers Face When Using Verification-Aware Programming Languages?. In *2025 IEEE 36th International Symposium on Software Reliability Engineering (ISSRE)*. 203–214. doi:10.1109/ISSRE66568.2025.00031

[9] Aaron Tomb and Anjali Joshi. 2025. Static Coverage in Deductive Software Verification. In *Proceedings of the 25th Conference on Formal Methods in Computer-Aided Design – FMCAD 2025 (Conference Series: Formal Methods in Computer-Aided Design, Vol. 6)*, Ahmed Irfan and Daniela Kaufmann (Eds.). TU Wien Academic Press, Wien, 251–261. doi:10.34727/2025/isbn.978-3-85448-084-6_32

[10] Silva Álvaro, Martins Ruben, and Mendes Alexandra. 2026. *ProofPulse: Interactive Proof Coverage Analysis for Dafny (Artifact)*. doi:10.5281/zenodo.21174686

Received 2026-05-12; accepted 2026-06-19