
Moving Beyond Specification Validation to Specification Refinement with Mutation Verification

Archie Licudi

Imperial College London
archie.licudi20@imperial.ac.uk

Adam Y. J. Jones

Imperial College London
adam.y.jones20@imperial.ac.uk

Abstract

Mutation verification introduces bugs into programs, and checks these mutated programs against a formal specification. Mutants which aren't killed by the specification may suggest weaknesses where the specification is under-constrained and should be tightened. MutDafny is a framework for performing mutation verification of Dafny programs. However, while MutDafny is able to find many mutants, manual analysis by a human is required for separating equivalent mutants and alive mutants which do not point to any specification weaknesses from legitimate alive mutants which expose an overly-weak specification. This makes the process of mutation verification extremely time-consuming and difficult to apply in practice. We use constraint-solving to discard equivalent mutants, and then use Large Language Models (LLMs) to separate alive mutants which point to legitimate weaknesses in the specification from spurious ones, and further use the LLM to refine the specification to kill these mutants, while ensuring we don't introduce any regressions causing previously killed mutants to become alive again. We find that this approach is both effective and quick, and suggests that we may move beyond the usage of mutation verification for specification validation towards its usage for specification refinement.

1 Introduction

Mutation testing, first introduced in DeMillo et al. [1978], has an extensive history in the testing community. The core idea is to introduce small bugs into a program, to check that the tests for the program catch these bugs, thereby "killing" the mutations. Central to the theory is the "competent programmer hypothesis", which states that experienced programmers produce programs that are either correct or close to correct, introducing small mistakes such as a wrong operator or an off-by-one error. This justifies the introduction of small local changes to code as a proxy for real bugs. If a test suite correctly catches these artificial bugs, then it is likely that it is an effective test suite for catching real bugs as well.

The problem of equivalent mutants - mutants which have the same execution behaviour and lie in the same equivalence class as the original program - is a longstanding problem faced by mutation testing. Early approaches to detecting equivalent mutants were explored in papers such as Offutt and Craft [1994], which uses compiler optimisation techniques to detect equivalent mutants, and Offutt and Pan [1997], which uses a constraint-based approach.

The idea of using a similar approach to mutation testing, termed *mutation verification*, to assess the validity of a formal specification was explored in early vision papers by Phipathananunth Phipathananunth, which explored the use of mutations of Solidity code to assess the quality of specifications for the Certora verifier Certora [2026], and Cordy et al. Cordy et al., which proposed the use of mutations of Promela code for the SPIN model checker Holzmann [1997] to assess the quality of LTL specifications Pnueli [1977]. The central idea is that, much like a test suite implicitly defines a

class of programs which pass the tests, a spec implicitly defines a class of programs which satisfy the spec. A test suite is, in many ways, an implicit specification, except rather than defining properties which must be true for all inputs, it defines the correct outputs for a number of inputs and hopes this is enough to constrain the system.

Ideally, the specification should accept an equivalence class of programs - that is, a set of programs which have the same input/output behaviour - which satisfy the constraints the programmer has in mind. If we introduce mutations which cause the program to have incorrect behaviour (for example, introducing a bug changing a plus sign to a minus sign), then the specification should be able to catch and "kill" such an error as the program is no longer equivalent to the original. If it fails to do so, we know the specification is under-constrained and must be tightened.

In this work, we build off of work in the recent paper MutDafny Amaral et al., which uses mutation verification to find overly weak specifications for the Dafny Leino [2010] programs in DafnyBench Loughridge et al. [2025]. Dafny is a verification-aware language, and differs from most verification frameworks in that it blends the code the formal specification, which is implemented as preconditions, postconditions, invariants, and termination measures directly in the source. Under the hood, Dafny uses the SMT solver Z3 De Moura and Bjørner [2008] to discharge proof obligations. MutDafny includes a number of common mutations used in mutation testing as well as some novel mutations in its suite of mutations.

While MutDafny is quite effective in finding alive mutants, there are a number of ways it can be improved. The authors generate mutants for 794 programs in DafnyBench and find 30,459 surviving mutants. Of these, they note that only 11,035 mutants affect methods with post-conditions. Of the mutants affecting methods with post-conditions, many will be equivalent mutants which do not expose any weaknesses in the specification, and even of the non-equivalent mutants many will expose spurious rather than legitimate weaknesses in the specification. They randomly select 24 files from DafnyBench, and analyse the 284 surviving mutants in methods with postconditions. After filtering out equivalent mutants and spurious mutants, they find only 5 specification weaknesses. This took eight hours of one person's effort (approximately 1.7 minutes per mutant). Clearly, this isn't sustainable or scalable for the in-practice use of mutation testing for detecting overly-weak specifications. Thus, this work proposes an approach for not only discarding the vast majority of uninformative alive mutations, but also for automatically *refining* the specification based off of the legitimate alive mutants.

2 Related Work

Covered in the introduction.

3 Methods

Once mutants that escape being killed by the spec are discovered, we can formalise the pipeline that a human operator needs to go through to turn these into actionable improvements to the spec, the time bottleneck identified by Amaral et al..

- **Equivalent-Mutant Testing** Out-of-the-box, MutDafny performs no equivalence checking - an alive mutant is always recorded as a *true positive*, with no attempt to verify if behaviour is actually changed. For example: it will commonly emit "mutants" which swap two independent assignment statements or change the condition in an `else if` block into an equivalent condition once the first branch is taken into account.
- **Root-Cause Identification** Mutants are, by construction, highly local and syntactic. The fuzzing engine will be able to identify lines of code where, e.g. arbitrary arithmetic can be substituted without violating the spec, but semantic awareness of the program context and a collection of multiple related mutants is usually required to make a useful inference about the actual gaps in the spec that were able to be slipped through.
- **Spec Strengthening** Once a root cause is identified, the spec needs to be modified to also handle this case. This becomes a problem of spec elicitation enhanced by an abstract root cause and concrete mutants that violate it. Consequently, a human-in-the-loop is often essential for this step: without a reliable source of truth, the synthesis model needs to ask

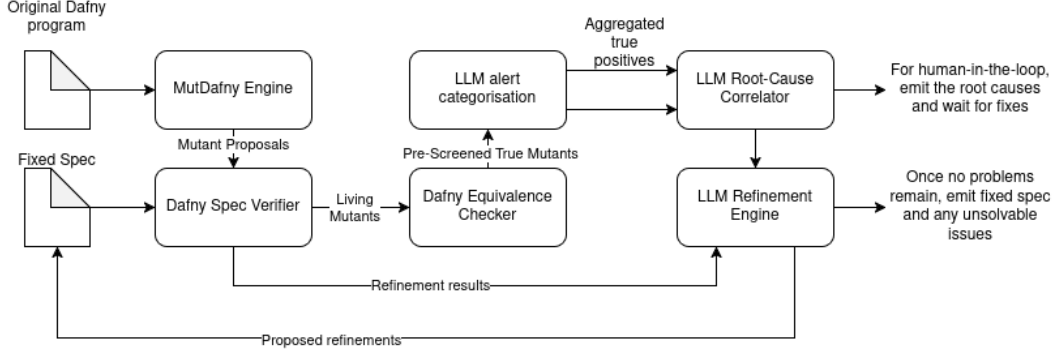


Figure 1: Block diagram of the MutRefine pipeline

what the intended behaviour here is, trying to infer the constraints *a priori* works only for the simplest problems where spec weaknesses are not semantic in nature.

This suggests the pipeline shown in Figure 1 for automating as much of this process as possible and ensuring the human operator has the highest quality information available to them when making manual refinements.

We discuss the implementation of each component:

1. **(Dafny Equiv Checker)** A common approach for symbolic checking of equivalent mutants is passing the two programs to an SMT solver to check equality. Fortunately, Dafny already has an SMT solver built-in to its verifier, and we can simply insert a `f_orig = f_mut` universal assertion and run the verifier. If it can be proven that the mutant does not change behaviour, it is discarded from the rest of the pipeline. This allows a large proportion of the false positives to be symbolically rejected before they hit any language model.

Many of the DafnyBench programs are written using methods, which enable loops, as opposed to functions which can be clearly compared extensionally. We simply introduce a shallow transpilation layer before EquivCheck that will localise the assertions and/or perform simple transformations of loopy methods into recursive helper functions. Whilst full coverage of extensionality checking would be optimal, it is a serious engineering effort, and the lightweight layer provides acceptable coverage for this tech demo.

2. **(LLM Categoriser)** A very small model (gemini-3.1-flash-lite) is then given the diff and asked to produce, with justification, a labelling of the mutant as EQUIV, WEAK_SPEC, or CONTEXTUAL. Contextual mutants will change behaviour, but not in a way that affects correctness of the function and can be specced for unambiguously, so are just surfaced to the user. Only WEAK_SPEC mutants continue in the automated pipeline.
3. **(LLM Correlator)** Once categories have been assigned and short semantic descriptions of the mutations generated in the previous round, they are grouped into batches of related mutants and dispatched back to the model. Now it is asked to partition them into "problems" which share a root cause, providing a description of what is causing this and where.
4. **(Refinement Engine)** For the human-in-the-loop version, this component is just an operator who reads the correlator report and makes decisions, inputting the patches back in to the pipeline. For the automated version, we now send the relevant sections of spec to a more advanced model (in our case, simply gemini-3.5-flash), the problem description, and a representative sample of the associated mutants, and ask it to either edit or insert a spec refinement that would solve this problem and catch the existing mutants.

On a failure of the fix to catch all the mutants or an invalidation of the reference implementation (which would indicate the proposed change now *overconstrains* the system) the LLM is given the results and queried to re-evaluate the problem, whether the cause was the analysis initially proposed, or whether this should be better split into two problems with two independent solutions. Once a max retry threshold is hit, the fixes are assumed to be unreliable and reverted, with the problem description surfaced to the user

```

SPEC PROBLEMS (4) · j/k select · Enter: view diffs
[C001] Missing functional correctness for insert (6 mut)
  The specification for insert and insertRecursion lacks postconditions that verify the structural integrity of the returned tree relative to the original. The current postconditions only cover value ranges, allowing arbitrary mutations to the tree structure or node linking. This allows for incorrect tree construction or invalid tree nodes.
Missing functional specification for GetMin/GetMax
  The methods GetMin and GetMax lack any specification (requires/ensures). As a result, any implementation that returns values of the correct type is considered valid, including ones that return the input tree itself or constants, completely ignoring the BST property.
[C003] Missing functional correctness for delete (18 mut) * unresolved
  The delete method has no specification. It does not ensure the resulting tree remains a valid BST, nor that the deleted value is absent, or that other elements are preserved. This permits the method to be entirely non-functional or logically broken while still verifying.
[C004] Missing specification for IO-only methods (1 mut)
  Methods like Postorder and Inorder are used for side-effect-based output and have no formal requirements or behavior specifications. Mutations that remove their logic remain verified because the spec does not demand any observable output or property preservation for these methods.
UNKILLABLE METHODS (126 mutants)
No postcondition ~ add ensures/decreases to enable spec fixes:
GetMin (23 mutant(s))
GetMax (21 mutant(s))
Inorder (18 mutant(s))
Postorder (46 mutant(s))
Main (18 mutant(s))

```

Figure 2: Problem Identification for BinaryTreeSearch

Future work would naturally consider these pipeline components augmented with tools from symbolic analysis grounded in theory to improve the LLM inference, but time constraints prevented this currently.

4 Results

Unfortunately, we have been unable to conduct a comprehensive evaluation of the framework. The time bottleneck is, as to be expected, in the Dafny verifier, both when generating the mutants and testing the proposed fixes - our choice of lightweight LLMs focused on speed reduces the time spend in inference to ~ 2 mins per file with very low token costs.

Instead, we focus on a case study: the `BinarySearchTree_tmp_tmp_bn2twp5_bst4copy.dfy` sample from DafnyBench, one of the five files uncovered manually by MutDafny to have a under-constraining spec. Of the 476 mutation targets for this file:

- 256 are killed and 25 are invalid (both detected by MutDafny)
- 30 are not killed by the spec, but are proven to be no-ops
- 59 are labelled by the LLM as EQUIV and 5 as CONTEXTUAL. An example of a valid contextual justification uncovered here is the following: This changes the logic of the main test setup; while it makes the program output incorrect relative to the original expected execution, it is a test-level mutation, not a fault in the verified methods themselves.
- 77 are from unkillable functions
- The final 24 survive scrutiny as WEAK_SPECS

The results are shown in Figure 2. Four problems are identified, with C001 being exactly the issue identified in the MutDafny paper, and the other three side effects of functions which have some specification of their functional correctness (e.g. *decreases*) so they are not skipped entirely, but the spec is still too underconstrained to propose a fix.

The LLM correctly concludes that the inserti

5 Discussion and Limitations

5.1 Discussion

We have promising initial results showing that this approach is feasible. However, due to time limitations, we weren’t able to perform the most comprehensive experiments.

5.2 Limitations and Future Work

We currently only evaluate our approach on DafnyBench, which consists of simple verified algorithmic code. We could evaluate our approach on more realistic and practical Dafny programs than

DafnyBench, such as IronFleet Hawblitzel et al. [2015] or the Dafny standard library. These larger programs will likely come with new challenges. Firstly, Constraint solving, which we use here for checking the equivalence of mutated programs to the original program, can be very slow as programs increase in complexity - it is relatively fast here as the programs in DafnyBench are quite small. For detecting equivalent mutants of larger programs, we can use approaches ranging from differential fuzzing Garcia et al. [2024], program normalisation Offutt and Craft [1994], property-based testing Claessen and Hughes [2000] or differential concolic execution Person et al. [2008]. Secondly, the larger programs may fill the context window of the LLMs or otherwise degrade their performance for specification refinement. Thus, we may evaluate the usage of techniques such as program slicing Weiser [1984] to reduce the amount of code which is shown to the LLM, while making sure to retain all the relevant code needed to decide how to refine the specification.

For the sake of time, we only evaluate a single LLM out of the box for specification refinement. We could perform rigorous experiments analysing different LLMs and prompting approaches. Additionally, we only evaluate our approach for a single verification language, Dafny. The same approach could in principle work with mutation verification of any verification language or framework, such as with model checking frameworks or protocol verification languages. Our approach may be less effective for such frameworks or languages where there may be less examples in the training data of the LLM, compared to Dafny which has an overall structure similar to many programming languages. Thus, we may need to think of approaches for enhancing our prompting-based approach. We could pursue a neurosymbolic approach, using feedback from symbolic tools to guide the LLM, and possibly using a small amount of finetuning to improve the performance of the LLM for our use-case.

6 Conclusion

We show that it is feasible to use mutation verification not just as an automated signal for when specifications are under-constrained, but to automatically refine specifications to improve upon these weaknesses. We hope to extend this work to more realistic benchmarks, making adaptations as needed, and enhance our approach by better leveraging available symbolic tools.

7 Code and Data

References

- Isabel Amaral, Alexandra Mendes, and José Campos. MutDafny: A Mutation-Based Approach to Assess Dafny Specifications. URL <http://arxiv.org/abs/2511.15403>.
- Certora. Certora prover documentation. <https://docs.certora.com/>, 2026.
- Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP 2000)*, pages 268–279, 2000. doi: 10.1145/351240.351266.
- Maxime Cordy, Sami Lazreg, Axel Legay, and Pierre Yves Schobbens. Towards Strengthening Formal Specifications with Mutation Model Checking. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 2102–2106. ACM. ISBN 979-8-4007-0327-0. doi: 10.1145/3611643.3613080. URL <https://dl.acm.org/doi/10.1145/3611643.3613080>.
- Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS’08/ETAPS’08*, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3540787992.
- R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978. doi: 10.1109/C-M.1978.218136.
- Bruno E. R. Garcia, Marcio E. Delamaro, and Simone R. S. Souza. Towards differential fuzzing to reduce manual efforts to identify equivalent mutants: A preliminary study. In *Proceedings of the XXXVIII Brazilian Symposium on Software Engineering (SBES 2024)*, pages 568–573, 2024. doi: 10.5753/sbes.2024.3557.

- Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP '15*, page 1–17, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338349. doi: 10.1145/2815400.2815428. URL <https://doi.org/10.1145/2815400.2815428>.
- Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997. ISSN 0098-5589. doi: 10.1109/32.588521. URL <https://doi.org/10.1109/32.588521>.
- K. Rustan M. Leino. Dafny: an automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR'10*, page 348–370, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3642175104.
- Chloe R Loughridge, Qinyi Sun, Seth Ahrenbach, Federico Cassano, Chuyue Sun, Ying Sheng, Anish Mudide, Md Rakib Hossain Misu, Nada Amin, and Max Tegmark. Dafnybench: A benchmark for formal software verification. *Transactions on Machine Learning Research*, 2025. ISSN 2835-8856. URL <https://openreview.net/forum?id=yBgTVWccIx>.
- A. Jefferson Offutt and W. Michael Craft. Using compiler optimization techniques to detect equivalent mutants. *Software Testing, Verification and Reliability*, 4(3):131–154, 1994. doi: <https://doi.org/10.1002/stvr.4370040303>. URL <https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.4370040303>.
- A. Jefferson Offutt and Jie Pan. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification and Reliability*, 7(3):165–192, 1997. doi: [https://doi.org/10.1002/\(SICI\)1099-1689\(199709\)7:3<165::AID-STVR143>3.0.CO;2-U](https://doi.org/10.1002/(SICI)1099-1689(199709)7:3<165::AID-STVR143>3.0.CO;2-U). URL <https://onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291099-1689%28199709%297%3A3%3C165%3A%3AAID-STVR143%3E3.0.CO%3B2-U>.
- Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2008)*, pages 226–237, 2008. doi: 10.1145/1453101.1453131.
- Siraphob Phipathananunth. Using Mutations to Analyze Formal Specifications. In *Companion Proceedings of the 2022 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, pages 81–83. ACM. ISBN 978-1-4503-9901-2. doi: 10.1145/3563768.3563960. URL <https://dl.acm.org/doi/10.1145/3563768.3563960>.
- Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science, SFCS '77*, page 46–57, USA, 1977. IEEE Computer Society. doi: 10.1109/SFCS.1977.32. URL <https://doi.org/10.1109/SFCS.1977.32>.
- Mark Weiser. Program slicing. *IEEE Trans. Softw. Eng.*, 10(4):352–357, July 1984. ISSN 0098-5589. doi: 10.1109/TSE.1984.5010248. URL <https://doi.org/10.1109/TSE.1984.5010248>.

A LLM Usage Statement

LLMs were used to assist in finding papers for understanding the related work and research gap, as well as to assist with the coding.