FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Specification-Driven Test Generation for Fault Localisation in Dafny

Sofia Vieira Pinto

Master in Informatics and Computing Engineering

Supervisor: Prof. Alexandra Mendes

July 10, 2026

Specification-Driven Test Generation for Fault Localisation in Dafny

Sofia Vieira Pinto

Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

President: Prof. Pedro Diniz
Examiner: Prof. Jorge Sousa Pinto
Supervisor: Prof. Alexandra Mendes

July 10, 2026

Resumo

À medida que o software se torna cada vez mais difundido, garantir a sua confiabilidade é uma preocupação crítica. Embora os testes de software continuem a ser a abordagem predominante, estes não garantem o comportamento correto em todos os cenários possíveis. A verificação de software, por outro lado, oferece fortes garantias matemáticas de correção, mas a depuração de falhas nestes ambientes é notoriamente complexa, dificultando a sua adoção. Esta dissertação preenche esta lacuna ao conciliar as garantias da verificação formal com a utilidade prática dos testes: utilizando testes automatizados para localizar erros quando a verificação falha. Apresentamos DSpec2Test, uma nova ferramenta de geração de testes baseados em especificações para Dafny, que utiliza a Forma Normal Disjuntiva (DNF) para realizar a Partição em Classes de Equivalência (ECP) e, opcionalmente, incorpora Análise de Valores de Fronteira (BVA). Avaliada em mais de 200 programas do DafnyBench, a nossa abordagem de caixa-preta (*SpecBva*) eliminou 91.5% dos mutantes, superando a abordagem tradicional de caixa-branca (DTest), embora a sua combninação tenha atingido o melhor resultado (91.95%). Para auxiliar a Localização de Falhas (FL), desenvolvemos um plugin de cobertura de código por teste. A aplicação de métricas Baseadas em Espectro (SBFL) revelou também que a combinação de testes de caixa branca e preta atinge o melhor resultado (*Top-1* de 46.5%). Contudo, a SBFL perde eficácia acentuadamente em programas maiores e na distinção de instruções sequenciais, visto que depende estritamente do rastreamento de execução. Consequentemente, exploramos uma abordagem semântica através do modelo `gpt-mini-5`. Avaliado no subconjunto de programas longos onde a SBFL demonstrou maiores limitações, o Modelo de Linguagem de Grande Escala (LLM) alcançou aproximadamente 75% de correspondência exata (*Exact Match*) e mais de 90% de correspondência parcial com a falha (*Any Match*). Ao contrário da SBFL, a profunda compreensão semântica do código permitiu ao modelo localizar erros com sucesso mesmo na ausência de testes a falhar. Em suma, este trabalho demonstra que aliar a geração orientada por especificações à depuração assistida por Inteligência Artificial (AI) fornece um caminho promissor para tornar as linguagens que incorporam a verificação mais acessíveis.

Palavras-chave: Geração de testes orientada por especificações • Localização de falhas • Verificação de software • Contratos • Dafny • Modelos de Linguagem de Grande Escala

Abstract

As software becomes increasingly widespread, ensuring its reliability is a critical concern. While software testing remains the mainstream approach, it cannot guarantee correct behaviour in every possible scenario. Software verification, on the other hand, offers strong mathematical guarantees of correctness, but debugging within these contexts is notoriously complex, hindering wider developer adoption. This dissertation bridges this gap by reconciling the robust guarantees of formal verification with the practical utility of testing: using automated tests to pinpoint errors when verification fails. We introduce DSpec2Test, a novel specification-based test generation tool for Dafny that employs Disjunctive Normal Form (DNF) to perform Equivalence Class Partitioning (ECP) and optionally incorporates Boundary Value Analysis (BVA). Evaluated on over 200 programs from DafnyBench, our black-box approach (*SpecBva*) killed 91.5% of mutants, outperforming the traditional white-box approach (DTest), although their combination yielded the best mutation score (91.95%). To support Fault Localisation (FL), we developed a custom per-test code coverage plugin. Applying Spectrum-Based Fault Localisation (SBFL) metrics also revealed that combining white and black box test suites yields the best results (46.5% *Top-1* score). However, SBFL struggles significantly with larger codebases and sequentially dependent statements due to its strict reliance on execution traces. Consequently, we explored a semantic-aware approach using the `gpt-mini-5` model. When evaluated on the specific subset of larger programs where SBFL degraded, the LLM achieved an *Exact Match* of approximately 75% and an *Any Match* exceeding 90%. Unlike SBFL, tthe model's deep semantic code comprehension allowed it to localise bugs even in the absence of failing tests. Ultimately, this work demonstrates that combining specification-driven testing with AI-assisted debugging provides a promising pathway to make verification-aware languages more accessible.

Keywords: Specification-driven test generation • Fault localisation • Software verification • Contracts • Dafny • Large Language Models

The United Nations Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) provide a global framework for achieving a better and more sustainable future for all. It includes 17 goals [1] to address the world's most pressing challenges, including poverty, inequality, climate change, environmental degradation, peace, and justice. We will examine this project in the light of these goals.

Software is the invisible infrastructure that powers our modern world. From the intelligent grids that distribute clean energy to the automated transport systems that cross our cities, society increasingly relies on critical software functioning flawlessly. By making verification-aware programming languages, like Dafny, more accessible and efficient through improved fault localisation, this work directly contributes to the reliability, safety, and resilience of these foundational technologies. Consequently, this dissertation aligns with the following goals and, particularly, these specific targets:

SDG 7 Ensure access to affordable, reliable, sustainable and modern energy for all.

- **7.1** By 2030, ensure universal access to affordable, reliable and modern energy services.

SDG 9 Build resilient infrastructure, promote inclusive and sustainable industrialisation and foster innovation.

- **9.1** Develop quality, reliable, sustainable and resilient infrastructure, including regional and transborder infrastructure, to support economic development and human well-being, with a focus on affordable and equitable access for all.
- **9.5** Enhance scientific research, upgrade the technological capabilities of industrial sectors in all countries, in particular developing countries, including, by 2030, encouraging innovation and substantially increasing the number of research and development workers per 1 million people and public and private research and development spending.

SDG 11 Make cities and human settlements inclusive, safe, resilient and sustainable.

- **11.2** By 2030, provide access to safe, affordable, accessible and sustainable transport systems for all, improving road safety, notably by expanding public transport, with special attention to the needs of those in vulnerable situations, women, children, persons with disabilities and older persons.

[1] United Nations SDGs webpage https://sdgs.un.org/goals, accessed 15 Jun 2026.

SDG	Target	Contribution	Performance Indicators and Metrics
7	7.1	By enhancing the robustness of the software governing modern energy services, this work helps to mitigate the risk of software-induced outages.	• Reduction in debugging time and software failure rates impacting energy systems.
9	9.1	By reducing the human effort required to debug verification-aware languages, it increases the reliability of critical infrastructures.	• Reduction in debugging time and software failure rates impacting critical infrastructures.
	9.5	Advances the state of the art in Formal Methods, by upgrading Dafny's current capabilities. Thus, it fosters technological innovation within the software engineering research community.	• Number of academic citations or publications derived from this work. • Adoption rate (e.g., repository clones/forks) of the developed tools.
11	11.2	Facilitates the creation of verified software for automated public transit systems, directly reducing the likelihood of accidents caused by programming errors.	• Reduction in debugging time and software failure rates impacting smart transport systems.

Acknowledgements

Nothing about this journey would be possible without the unconditional love, support, and values my parents instilled in me since birth to prepare me for my future. To them, I owe not only my life, but also every opportunity that has crossed my path.

Looking upwards in the bloodline, I must thank my grandmas (my guardian angels), and my grandpa, for acknowledging and passing forward to me that education is the most powerful tool one can have, and that we may never take it for granted. To the rest of my family and friends, I appreciate you all deeply for your comprehension, kindness, and amusement. Inês, particularly to you, thank you for never leaving my side.

Nevertheless, this thesis would not have come to life without the continuous guidance and support of my supervisor, Professor Alexandra Mendes. Thank you for believing in me since my first year at University, for introducing me to innumerable life-changing opportunities, and for lowering my stress levels in every single meeting. To Professor João Pascoal Faria, I thank you also for following this project up-close and providing helpful insights. I am equally grateful to Isabel and Álvaro for always making time to help me out with my countless questions.

Lastly, I want to express my deepest gratitude to the love of my life, Pedro. There are no words to describe your patience, thoughtfulness, love, and affection towards me. Thank you for always pushing me above my limits, and celebrating my accomplishments as your own. Most importantly, thank you for making me a better person, and the world a better place.

This work is financed 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 (https://doi.org/10.54499/2023.15557.PEX).

Sofia Vieira Pinto

Fundação
para a Ciência
e a Tecnologia

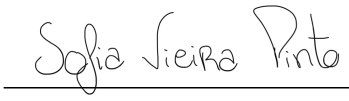
Declaration of honour

I declare that this dissertation is my own work and has not previously been submitted or assessed at this or any other institution. References to other authors (statements, ideas, thoughts), as well as the use of published works, strictly comply with the rules of attribution and are duly identified in the text and in the bibliographic references, in accordance with the applicable referencing standards. I am aware that the practice of plagiarism or self-plagiarism constitutes an academic offence, as established in the U.Porto Code of Ethics.

I further declare that portions of the text in this dissertation are adapted from a peer-reviewed paper titled: *"DSpec2Test: Specification-Driven Test Generation in Dafny"* [64]. This paper has been accepted for presentation at the International Conference on Automated Software Engineering (ASE) and is pending formal publication in the conference proceedings. This work, of which I am the first author, was developed in collaboration with three other researchers, including my supervisor. I state with transparency and rigor that all content, implementations, and evaluations included in this dissertation are exclusively my own original work. While the accepted paper contains additional collaborative evaluations, those have been entirely excluded from this document and replaced with independent evaluations conducted solely by me. The express agreement signed by all co-authors validating the independent nature of this dissertation is provided in Appendix A. Because the proceedings are pending publication, formal publisher authorisation is not applicable. However, the conference's standard author rights permit the inclusion of accepted manuscript content in academic theses.

I also declare that I have used Artificial Intelligence tools to aid in text refinement and in code development of the tools mentioned in this dissertation, which have both been carefully checked for errors, inaccuracies, unfounded attributions or other forms of bias in content and arguments, as well as determining authorship.

Sofia Vieira Pinto



*"**Luck** is what happens*
*when **preparation** meets **opportunity**."*

Seneca

Contents

List of Figures

List of Tables

List of Acronyms

AI	Artificial Intelligence
APR	Automated Program Repair
AST	Abstract Syntax Tree
BVA	Boundary Value Analysis
DbC	Design By Contract
DNF	Disjunctive Normal Form
ECP	Equivalence Class Partitioning
FL	Fault Localisation
IR	Information Retrieval
LLM	Large Language Model
MRR	Mean Reciprocal Rank
SAT	Boolean Satisfiability Problem
SBFL	Spectrum-Based Fault Localisation
SDG	Sustainable Development Goal
SMT	Satisfiability Modulo Theories

Chapter 1

Introduction

1.1 Context

Nowadays, software is present where it once was unthinkable: in a watch, on an oven, even in a door lock. With such abundance, failures become almost inevitable.

Testing is the most widespread approach to detect bugs in a system [5]. However, as Edsger Dijkstra once put it: *"testing can show the presence of errors but never their absence"*, since it only proves the correct behaviour of a system for a limited set of scenarios. Although in some cases testing might seem sufficient, such risk cannot be tolerated when it comes to critical systems. For instance, in an aeroplane whose software has simply been tested, the absence of a single test case may lead to a catastrophic outcome.

In contexts where failures are not tolerable, systems require stronger guarantees than testing can provide. Formal verification addresses this need by relying on mathematical proofs to show that the code behaves exactly as expected. A prominent methodology for achieving this is Design By Contract (DbC) [41], which specifies the intended behaviour of software through contracts, such as preconditions, postconditions, and invariants. Modern verification-aware languages, such as Dafny [34] and Verus [33], allow developers to embed these DbC contracts directly into their code so they can be automatically verified, ensuring the implementation strictly satisfies its specification.

Despite these strong guarantees, the adoption of verification-aware languages is still limited [48]. Writing correct specifications and code remains a complex task, and developers are still prone to introduce bugs. When a verifier fails, finding the root cause is crucial. However, poor error messages and feedback are known challenges that developers face when using verification-aware languages.

Traditionally, Automated Program Repair (APR) [45] automatically identifies and fixes a given bug while relying on weak correctness criteria, such as test suits which, as aforementioned, are limited. However, recent work uses formal specifications as correctness criteria for program repair [74]. Our work continues this line of research by automatically generating test-suites for Fault Localisation (FL) based on the specification (which we assume to be correct). This novel strategy

allows us to explore deterministic Spectrum-Based Fault Localisation (SBFL) techniques applied to the generated test-suite. We aim to automatically locate faults to improve verifier feedback and also support automated program repair.

This work was developed within the wider effort of the VeriFixer[1][2] project, whose goal is to develop methods for automatically fixing bugs in code written in verification-aware programming languages, focusing on Dafny, which is used both in teaching and in industry (e.g., Amazon and ConsenSys [6–8]).

1.2 Motivation

The goal of this project is to make verification-aware languages more accessible to programmers, in order to increase their usage and, in this way, encourage the development of software that is more reliable.

A recent study highlighted several difficulties that developers face when coding in verification-aware languages [48]. Amongst them lies the scarcity of documentation and the error messages' lack of clarity. The latter could be greatly improved with FL, by providing more precise clues on potential causes for the error.

Therefore, by adapting and innovating upon established FL techniques to support verification-aware languages, we are directly addressing a current challenge identified in the literature. This, in turn, may make these languages more accessible by reducing the current human effort needed to identify the source of errors. Thus, this work has the potential to contribute to a future where these languages are not expert-exclusive but rather within reach of a broader public. By increasing the popularity of verification-aware languages, we can move towards a culture in which correctness, safety, and reliability are integral parts of everyday development practices.

1.3 Problem

This work aims to address existing difficulties in detecting the exact location where a piece of code written in a verification-aware language is failing. Locating errors in verification-aware languages represents a barrier both in the learning process of this kind of programming languages and in their use in real-world applicatons, since the feedback offered by the verifier is often not useful enough, as mentioned above.

In examples like the one depicted in Listing 1.1, the feedback provided by the verifier on the second line — *"a postcondition could not be proved on this return path"* — is not indicative of where the fault is. In these cases, the programmer will most likely need to search throughout the whole implementation in order to find the bug.

Listing 1.1: Example of a Dafny program where the error message — *"a postcondition could not be proved on this return path"* — is not sufficiently helpful.

[1] https://verifixer.github.io/
[2] https://doi.org/10.54499/2023.15557.PEX

```
method Max(x: int, y: int) returns (m: int)
   ensures m >= x && m >= y
   ensures m == x  m == y
{
   if x < y {
      m := x; // BUG -> should be m:=y;
   } else {
      m := x;
   }
}
```

Hence, the need to develop increasingly helpful tools and mechanisms to aid developers in finding bugs is justified by examples such as the one shown in Listing 1.1, as well as by the other previously identified challenges faced by programmers when writing code in verification-aware languages.

1.4 Research Questions

In this section, we present the three research questions that guide this project.

Firstly, it is relevant to investigate the existing techniques for FL in verification-aware languages by analysing the state of the art.

RQ 1: Which fault localisation techniques are currently available for verification-aware programming languages?

Secondly, it is important to explore how these techniques can be applicable or adapted to different programming languages, particularly Dafny.

RQ 2: How can these fault localisation techniques be applicable to Dafny?

Lastly, it is essential to analyse and compare the performance of the deterministic (which, given the same input, always produce the same output) and non-deterministic (taking advantage of LLMs) approaches found for FL, in the context of Dafny.

RQ 3: How effective are these deterministic and non-deterministic approaches at detecting bugs in Dafny programs, and how do they compare to one another?

Guided by these questions, this research represents a significant step forward in FL for verification-aware programming languages, particularly Dafny.

1.5 Contributions

The main contributions of this work are:

- ★ DSpec2Test, the first tool for *specification-based testing* in Dafny. It is integrated as a new *Spec* mode of the existing `generate-tests` command (originally, DTest [20]) which generates tests for Dafny based on implementations. Conversely, DSpec2Test derives tests from the program's contract and can support test-driven development, specification validation, and fault localisation.

- ★ The first plugin for per-test code coverage for Dafny. By running this plugin together with the existing `dafny test` command — which executes Dafny methods marked as tests — and its `-no-verify` flag, it is possible to obtain the lines of the original (buggy or not) program covered by each of the tests in a file. It also gracefully handles infinite loops or recursive calls.

- ★ Evidence that specification-based testing is superior to implementation-based testing, in contexts where the specification is highly restrictive. These results were gathered through an empirical mutation testing evaluation on 2,085 realistically-killable mutants from DafnyBench, in which DSpec2Test kills 91.5% of the mutants, outperforming DTest (88.45%). However, the best results are actually achieved once white-box and black-box test generation strategies are combined (91.95% kill rate), evidencing their complementarity.

- ★ Evidence that LLM-based FL achieves much higher performance than SBFL. Our evaluation reveals that black-box testing using Boundary Value Analysis (BVA) (*SpecBva*), coupled with the Ochiai coefficient, achieves the highest standalone *Top-1* score of 46%. Furthermore, combining white-box and black-box strategies slightly improves the overall *EXAM* score to 63.77%. Comparatively, the LLM (`gpt-mini-5` [49]), with the tests provided by DSpec2Test, found a faulty line in 93.37% of the cases, in longer programs. Additionally, it was also able to detect at most 90% of the bugs (with *SpecBva*) in programs with no failing test cases, a result impossible to achieve by SBFL.

1.6 Dissertation Structure

This dissertation contains six chapters. In addition to the Introduction, Chapter 2 presents the fundamental concepts required to understand this research, and Chapter 3 describes the state of the art. Furthermore, Chapter 4 describes the proposed methodology, detailing the developed pipeline: from the automatic generation of test-suites based on formal specifications to the implementation of a coverage extraction plugin that will be crucial for SBFL. The evaluation of the test-generation tool through mutation testing, plus the comparision of the results obtained through deterministic (SBFL) and non-deterministic (Large Language Model (LLM)-based) FL techniques, is presented in Chapter 5. Finally, the conclusions and future work for this thesis can be found in Chapter 6.

Chapter 2

Background

This chapter provides an overview of core technical concepts central to this work: Software Testing, Software Verification, and Fault Localisation. This context is fundamental for understanding the research gap identified in the Literature Review and the subsequent solutions proposed.

2.1 Software Testing

With the increasing use and existence of technology, Software Testing [46] has become a constant necessity. Despite its growing application, it still presents itself as a difficult task to master, given that the state space is infinite. Therefore, test-case design becomes essential.

This can be achieved by either white-box testing, black-box testing, or, ideally, a combination of both. The former is more widespread and focuses on creating tests that cover the most statements, decisions, or conditions. In other words, it analyses a program's implementation and attempts to achieve the highest coverage, i.e., the greatest proportion of code that is executed by the test suite.

On the other hand, black-box testing is completely blind to a program's implementation, and focuses solely on its specification (the pre and post conditions that define a program's expected behaviour). This strategy encloses some methodologies that enable the creation of a more comprehensive test-suite, such as Equivalence Class Partitioning (ECP), Boundary Value Analysis (BVA), Cause-Effect Graphing, and Error Guessing.

ECP partitions the domain into two or more groups that define sets of inputs with similar characteristics. This means that if a value from one partition raises an error, it is expected that all values in that same partition would flag the same bug. Nevertheless, as real-world inputs often possess multiple attributes, equivalence classes derived from different conditions can overlap. Consequently, two values from the same equivalence class could potentially result in different test outcomes, if one of them simultaneously belongs to another overlapping class that behaves differently.

While ECP effectively reduces the test space, it often disregards extreme or boundary values. BVA addresses this by targeting edge cases that sit precisely on, below, or above the boundaries of input or output equivalence classes.

Additionally, Cause-Effect Graphing demands the translation of a program's specification into a formal Boolean logic network. It maps input conditions (causes) to output actions (effects) and applies constraints to eliminate impossible combinations. This graph is subsequently converted into a decision table, ensuring that all significant logical interactions are tested without resulting in combinatorial explosion.

Lastly, black-box testing also encompasses Error Guessing, an heuristic that cannot be easily automated, as it relies on the intuition of an experienced developer to spot likely fallible scenarios.

Ultimately, the best approach would be to combine all of these strategies and methodologies to achieve the highest program fidelity. However, no amount of tests will ever warrant the correct behaviour of a program, nor provide the same guarantees as Software Verification.

2.2 Software Verification

While testing is heavily reliant on heuristic approaches to uncover faults, Software Verification employs formal mathematical methods to prove the correct behaviour of a program [72]. This process requires a program to include not only an implementation but also a formal specification.

Consequently, successful verification proves that the implementation strictly complies with its defined specification. It is important to note that verification does not guarantee the absolute absence of bugs, but rather the absence of faults regarding the provided specification. If the latter accurately reflects the real-world requirements, the resulting software achieves a level of reliability unachievable by testing alone.

The mathematical foundation for this paradigm is rooted in Hoare Logic [26]. Hoare introduced the formal system of reasoning using triples in the form of {P}C{Q}, where P represents the preconditions, C corresponds to the program's commands or implementation, and Q represents the postconditions. This logic establishes that if a program C executes in a state satisfying P, it will terminate in a state satisfying Q.

This mathematical formalism heavily inspired the software engineering methodology known as Design By Contract (DbC) [41]. Under DbC, developers define a rigid contract (the specification) before writing the corresponding implementation. This pivots the development mindset from how the program is implemented to what the program is formally required to do.

To bridge the gap between abstract mathematical proofs and practical software engineering, several verification-aware programming languages have been developed (e.g., Verus [33] and Why3 [21]). These languages natively integrate specifications and code, automating the proof process for the developer. Dafny is a prominent example of such a language and is the focus of our work.

2.2.1 Dafny

Dafny [34] is an open-source verification-aware programming language that natively supports formal specification. To facilitate integration with existing codebases, it is designed to compile into several major programming languages, including C# [29], Java [24], and Python [63]. Listing 2.1 illustrates a standard Dafny method. The code is divided into its specification (lines 2 to 5) and its implementation (lines 6 to 16). The specification defines the method's contract: line 2 establishes a precondition (`requires`), strictly defining the valid input domain, while lines 3 to 5 define the postconditions (`ensures`), declaring the expected output behaviour of the program.

Listing 2.1: Example of a Dafny method.

```
1  method {:testEntry} Classify(x: int) returns (r: int)
2    requires x >= -100 && x <= 100
3    ensures x < 0 ==> r == -1
4    ensures x == 0 ==> r == 0
5    ensures x > 0 ==> r == 1
6  {
7    if x > 0 { //Buggy line: Should be x < 0
8      r := -1;
9    }
10   else if x == 0 {
11      r := 0;
12   }
13   else {
14      r := 1;
15   }
16 }
```

To enforce these contracts, Dafny relies on the Z3 SMT solver [17] for continuous static verification. The solver analyses the code continuously during development. In this example, because the implementation contains a logical bug (line 7) that contradicts the specified behavior, the SMT solver will flag a verification error, showing that the code does not satisfy its own contract.

2.2.1.1 DTest: `generate-tests` command

Beyond static verification, Dafny provides built-in support for automated white-box testing via the experimental `generate-test` command, which originates from the DTest framework [20]. This tool generates unit tests based on the program's implementation and offers two primary modes: *Block* coverage and *Path* coverage. A variation of the former, *InlinedBlock*, allows for function inlining to achieve a more in-depth analysis. However, as the tool is still experimental, it has some limitations. Currently, target methods must be encapsulated within a module and explicitly annotated with the `{:testEntry}` attribute, visible in the first line of Listing 2.1. Additionally, the generator does not yet support mutable state objects, such as classes and arrays, or any form of non-determinism (e.g. `:|` operator).

An example of the tests produced by the `generate-tests` command in *Block* mode for the program in Listing 2.1, assuming it is contained in a module named `Module`, inside file `Example.dfy`, can be found in Listing 2.2.

Listing 2.2: Output produced by the `generate-tests` command in *Block* mode for the `Classify` method. `Classify` was initially written in an `Example.dfy` file, inside a module named `Module`.

```
include "Example.dfy"
module ExampledfyUnitTests {
import Module
 method {:test} Test0() {
   expect -1 >= -100 && -1 <= 100, "If this check fails at runtime, the test does not meet
the preconditions";
   var r0 := Module.Classify(-1);
   expect -1 < 0 ==> r0 == -1;
   expect -1 == 0 ==> r0 == 0;
   expect -1 > 0 ==> r0 == 1;
 }
 method {:test} Test1() {
   expect 0 >= -100 && 0 <= 100, "If this check fails at runtime, the test does not meet the
preconditions";
   var r0 := Module.Classify(0);
   expect 0 < 0 ==> r0 == -1;
   expect 0 == 0 ==> r0 == 0;
   expect 0 > 0 ==> r0 == 1;
 }
 method {:test} Test2() {
   expect 1 >= -100 && 1 <= 100, "If this check fails at runtime, the test does not meet the
preconditions";
   var r0 := Module.Classify(1);
   expect 1 < 0 ==> r0 == -1;
   expect 1 == 0 ==> r0 == 0;
   expect 1 > 0 ==> r0 == 1;
 }
}
```

The generated tests cover all the conditional blocks present in the original method's implementation. Consequently, if a block is mistakenly omitted from the implementation, the corresponding test will also be absent. For instance, if the `else` branch (lines 13 to 15, from Listing 2.1) was missing from `Classify`'s implementation, `Test0` would not be generated.

Once saved to a Dafny file, these methods are recognised as executable tests due to the `{:test}` attribute and can be executed using the `dafny test` command. When testing faulty implementations, it is necessary to append the `-no-verify` flag, otherwise, the verifier will halt execution upon detecting the original program's contract violations, preventing the test results from being output.

Because the original `Classify` implementation in Listing 2.1 contains a logical error, running `dafny test ExampleTests.dfy -no-verify` produces the output shown in Listing 2.3.

Listing 2.3: Output produced by the `test` command with the `-no-verify` flag, assuming the tests were stored in an `ExampleTests.dfy` file

```
Dafny program verifier did not attempt verification
ExampledfyUnitTests.Test0: FAILED
      ExampleTests.dfy(7,0): expectation violation
ExampledfyUnitTests.Test1: PASSED
ExampledfyUnitTests.Test2: FAILED
```

```
6       ExampleTests.dfy(23,0): expectation violation
7 [Program halted] ExampleTests.dfy(1,0): Test failures occurred: see above.
```

In conclusion, while still experimental, Dafny's automated testing capabilities provide a valuable complement to its static verification.

2.3 Fault Localisation

Software fault localisation is known to be one of the most time-consuming tasks during software development. It is also just as crucial. For these reasons, it has gained increasing popularity over the past years. In fact, multiple automated FL techniques have been developed to direct developers to the exact location of a bug with minimal human intervention [70]. They can be broadly classified into distinct families based on their underlying mechanisms.

Slice-based [66] techniques isolate faults by extracting the specific subset of program statements that directly influences variables at the point of a manifested failure. Statistics-based [35] techniques compute suspiciousness scores for individual statements by applying statistical formulas to the execution traces of both passing and failing test cases. Similarly, Spectrum-Based Fault Localisation (SBFL) [57] approaches apply similarity coefficients to code coverage to detect faults. These have served as a foundational baseline in the literature [50], constituting the largest portion in the distribution of papers retrieved by Wong et al. [70]. Another paradigm includes program state-based [14] techniques, which alter internal states during execution (such as mutating variable values) to observe which modifications transition a failing run into a passing one. Furthermore, machine learning and data mining-based approaches [76] take historical data as input, including past bug reports and code metrics, to train predictive models for fault detection. Finally, model-based [56] techniques pinpoint discrepancies by comparing a program's actual execution against a formal and predefined mathematical representation of its expected behaviour.

Because most of these techniques output a ranked list of suspicious elements, their performance is typically evaluated using well-established ranking metrics [67, 70]. *Top-N* measures the percentage of faults successfully located within the top N positions of the ranked candidate list (commonly *Top-1*, *Top-3*, and *Top-5*). The *Mean Reciprocal Rank (MRR)* averages the reciprocal of the rank at which the first faulty element is found across all evaluated programs. The *Average Rank* represents the mean position at which the ground-truth faults are ranked across the entire dataset. Finally, the *EXAM* score evaluates the percentage of code that a developer must inspect before finding the fault, with lower scores indicating a more effective tool.

A common criticism of statement-level metrics is their apparent inability to detect omission faults (i.e., missing statements), as there is no specific line of code to assign a score to. However, in practice, the absence of a required statement often forces the execution down an unexpected or incorrect control-flow branch. FL techniques can subsequently assign high suspiciousness scores to these incorrectly executed branches, providing developers with a highly useful starting point for their debugging activities [70].

Nevertheless, the utility of a FL tool should not be assessed solely based on these metrics. There are other factors that must be taken into account, such as time, space, human effort, and computational cost.

Because no single approach is undoubtedly superior to others in all fronts, combined strategies have also been considered. By incorporating the strengths of various methods and limiting the drawbacks, researchers aim to move closer to a perfect automated debugging solution, although such a system has yet to be found [70].

However, given the established prominence of SBFL as a traditional benchmark in automated debugging, coupled with the recent advancements of LLMs in software engineering tasks, the subsequent sections will further explore the mechanics of these two highly relevant approaches.

2.3.1 Spectrum-Based Fault Localisation

Spectrum-Based Fault Localisation (SBFL) [57] is amongst the most widely adopted deterministic techniques due to its lightweight and language-agnostic nature [70]. SBFL evaluates the execution spectra of passing and failing test cases to compute a suspiciousness score for each statement in the code. To calculate this score, SBFL formulas rely on the following fundamental variables:

- N_F: the number of failing tests.
- N_{CF}: the number of failing tests that execute a statement.
- N_{UF}: the number of failing tests that do not execute a statement.
- N_S: the number of passing tests.
- N_{CS}: the number of passing tests that execute a statement.
- N_{US}: the number of passing tests that do not execute a statement.

Based on these parameters, various similarity coefficients have been proposed. The Tarantula [28] approach ranks statements based on the proportion of failing tests that execute them relative to the proportion of passing tests that do so. Its suspiciousness score is calculated as:

$$S_{\text{Tarantula}} = \frac{\frac{N_{CF}}{N_F}}{\frac{N_{CF}}{N_F} + \frac{N_{CS}}{N_S}} \tag{2.1}$$

Ochiai [1], originally borrowed from the molecular biology domain, is particularly successful at single-fault identification. It often outperforms Tarantula by penalising statements that are frequently executed by passing tests, using the geometric mean of the total number of failing tests and the total number of tests executing the statement:

$$S_{\text{Ochiai}} = \frac{N_{CF}}{\sqrt{N_F \times (N_{CF} + N_{CS})}} \tag{2.2}$$

Another notable metric is DStar (D^*) [71], which isolates faulty elements by exponentially weighting the number of failed executions. The exponent $*$ represents a user-defined power (typically 2, denoted as D^2), allowing the metric to heavily reward statements executed by many failing tests:

$$S_{D^*} = \frac{N_{CF}^*}{N_{UF} + N_{CS}} \tag{2.3}$$

While empirical studies often demonstrate that D^* is more effective than traditional similarity coefficients across a variety of scenarios, no single metric guarantees optimal FL universally. Because different codebase structures and fault types can skew spectral data differently, employing and comparing multiple metrics remains a standard and necessary practice in FL research.

2.3.2 LLM-Based Fault Localisation

More recently, researchers have increasingly started to compare deterministic FL metrics (which consistently produce the same results for a given input) with outputs generated by LLMs [75, 77], which are non-deterministic. Instead of relying on execution spectra, LLMs exploit vast amounts of pre-trained data to semantically analyse source code and test cases to identify bugs.

The efficacy of LLMs in these tasks, however, is highly dependent on how the context and instructions are formulated (i.e., prompt engineering). Recent literature demonstrates that the structure, length, and order of a prompt significantly impact a model's performance. For instance, providing excessive codebase context can overwhelm the model and dilute the bug's signal. Sepidband et al. [58] highlight that targeted, element-level or file-level contexts typically yield much better results than feeding entire code repositories.

Furthermore, LLMs are known to exhibit an input order bias during FL tasks. Models tend to prioritise earlier information in the prompt, meaning that breaking the prompt down into smaller and segmented contexts can drastically reduce hallucination and improve localisation [54]. Consequently, structuring a prompt for non-deterministic FL is not a trivial task. It requires balancing context granularity and extensiveness with logical and sequential constraints to guide the LLM's reasoning effectively.

Finally, the shift from deterministic ranking to generative prediction introduces a duality in how FL performance can be evaluated. When LLMs are configured to output a ranked list of suspicious lines or files, traditional Information Retrieval (IR) ranking metrics such as *MRR* and *Top-N* remain the standard [47]. However, when developers task LLMs with directly extracting the specific faulty text span or line range, the process transforms into an extractive IR problem. In this generative context, FL performance is evaluated using other metrics: *Exact Match* (the percentage of cases where the predicted lines perfectly align with the ground-truth fault), alongside *Precision*, *Recall*, and the *F1-Score* to measure the exactness and completeness of the retrieved code snippet [61]. These metrics provide an assessment of a model's performance while avoiding the artificial inflation that *Accuracy* would produce due to the heavy class imbalance known to FL (i.e., the vast majority of source code being non-faulty).

Chapter 3

Literature Review

Our literature review addresses **RQ 1** by exploring the available techniques for both FL and automated testing in verification-aware languages.

3.1 Fault Localisation Approaches in Verification-Aware Languages

Our investigation began by targeting Fault Localisation (FL), through an initial exploratory search with a highly restrictive query that combined Dafny with FL:

> (“fault localisation” OR “fault localization” OR “fault discovery”) AND “Dafny”

This preliminary search produced 29 results across four different data sources — ACM Digital Library[1], arXiv[2], IEEE Xplore[3], ScienceDirect[4], and Scopus[5] — but only two were actually relevant. This indicates a gap in the existing literature, which this work aims to address.

Given the limited amount of relevant results obtained, we broadened the scope of the query, by including other verification-aware programming languages and other closely related terms:

> (“fault localisation” OR “fault localization” OR “fault discovery”) AND (“verification-aware” OR “verifiable programming language” OR “verifiable programming languages” OR “contract programming” OR “design by contract” OR “contract-based” OR “programming by contract” OR “Agda” OR “Bandera” OR “Dafny” OR “Escher C Verifier” OR “Eiffel” OR “Frama-C” OR “Idris” OR “Java PathFinder” OR “JML” OR “mbeddr” OR “Spark Ada” OR “VeriFast” OR “Verus” OR “Whiley” OR “Why3”)

[1] ACM Digital Library webpage https://dl.acm.org, accessed 01 Jan 2026.
[2] arXiv webpage https://arxiv.org, accessed 01 Jan 2026.
[3] IEEE Xplore webpage https://ieeexplore.ieee.org, accessed 01 Jan 2026.
[4] ScienceDirect webpage https://www.sciencedirect.com, accessed 01 Jan 2026.
[5] Scopus webpage https://www.scopus.com, accessed 03 Jan 2026.

The programming languages listed in this query were extracted from the Formal Methods Europe (FME) Industry Committee's[6] website, since it is maintained by researchers focusing on Formal Methods and verification and, therefore, reflects commonly recognised languages in the Formal Methods community. Nevertheless, KeY [2], P [18], Spec# [4], TLA+ [32], and VCC [15], were excluded from the search query, because these are highly frequent expressions in other fields which, therefore, gathered numerous non-relevant results. This occurred because some search engines are case-insensitive or ignore symbols like + and #, causing these short terms or acronyms to match unrelated content. Additionally, Verus was included in the query due to its ability to verify Rust [31] code.

This search was carried out across the aforementioned data sources with no filters applied. However, due to connector limits in some data sources, the query was partitioned into four segments, as described below:

S1: ("fault localisation" OR "fault localization" OR "fault discovery") AND ("verification-aware" OR "verifiable programming language" OR "verifiable programming languages" OR "contract programming" OR "design by contract" OR "contract-based")

S2: ("fault localisation" OR "fault localization" OR "fault discovery") AND ("programming by contract" OR "Agda" OR "Bandera" OR "Dafny" OR "Escher C Verifier" OR "Eiffel")

S3: ("fault localisation" OR "fault localization" OR "fault discovery") AND ("Frama-C" OR "Idris" OR "Java PathFinder" OR "JML" OR "mbeddr")

S4: ("fault localisation" OR "fault localization" OR "fault discovery") AND ("Spark Ada" OR "VeriFast" OR "Verus" OR "Whiley" OR "Why3")

The results were then aggregated. The amount of articles retrieved from each data source, and per query segment, is identified in Table 3.1.

Table 3.1: Number of documents retrieved per data source, and per segment

Data source	S1	S2	S3	S4	Number of documents
ACM Digital Library	71	76	102	16	265
arXiv	1	2	0	0	3
IEEE Xplore	2	3	3	0	8
ScienceDirect	13	19	34	1	67
Scopus	105	25	116	5	251
Total	192	125	255	22	**594**

[6]Formal Methods Europe Industry Committee webpage https://fme-industry.github.io/#verifiable-programming-languages, accessed 22 Oct 2025.

The total number of documents retrieved was 594. After removing the duplicates, using Zotero[7], 401 documents remained. Next, the inclusion and exclusion criteria identified in Table 3.2 and Table 3.3, respectively, were applied. There were no retracted publications in this set, although there were 10 papers in a language other than English, which reduced the dataset to 391 documents. However, after removing all the documents related to proceedings, newsletters, and lecture notes, the number of documents decreased to 301.

Table 3.2: Inclusion criteria considered in this study

ID	Inclusion Criteria
IC1	Published in the English language
IC2	Published in the area of fault localisation in verification-aware languages
IC3	Full-text availability

Table 3.3: Exclusion criteria considered in this study

ID	Exclusion Criteria
EC1	Duplicated studies
EC2	Retracted publications
EC3	Lecture notes, newsletters, or proceedings

Afterwards, the remaining documents were analysed based on their title and abstract, and their full-text accessibility was assessed. Thus, the final number of documents identified via databases was six. However, during the full-text analysis of one of these studies [51], an additional relevant paper [44] was identified via citation searching. This record was retrieved, assessed, and deemed eligible. Consequently, a total of seven studies was included in the final review. These are presented in Table 3.4.

Table 3.4: Title of documents included for review

Reference	Title
[10]	Angelic debugging
[51]	Automated fixing of programs with contracts
[53]	How testing helps to diagnose proof failures
[23]	Improving the effectiveness of spectra-based fault localization using specifications
[59]	Inferring multiple helper Dafny assertions with LLMs
[44]	Programs That Test Themselves
[74]	Specification-Guided Repair of Arithmetic Errors in Dafny Programs using LLMs

The complete study selection process is shown in Figure 3.1, following the PRISMA[8] (Preferred Reporting Items for Systematic Reviews and Meta-Analyses) 2020 flow diagram.

[7]Zotero webpage https://www.zotero.org, accessed 12 Nov 2025.
[8]PRISMA webpage https://www.prisma-statement.org, accessed 12 Nov 2025.

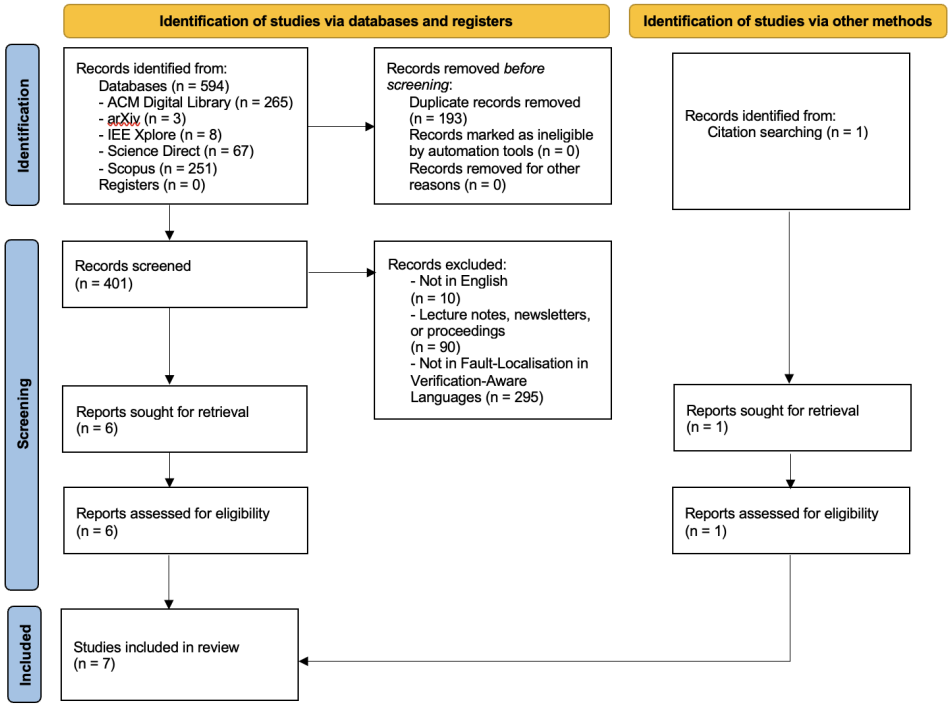


Figure 3.1: PRISMA 2020 flow diagram.

3.1.1 Relevant Literature Analysis

When coding in any programming language, verification-aware or not, it is common for developers to test their software, find bugs, correct them, and then neglect these tests since the defect has been corrected. However, AutoTest [44] shows how crucial it can be to store these tests for future regression testing. Additionally, and most importantly, they present contracts as valuable tools for fault detection. Their work focuses on the automatic generation of test cases by executing programs annotated with contracts, using the latter as runtime oracles. These tests are created from class instances, by calling their routines multiple times with various arguments. A minimised version of the failing ones is then stored for future regression testing. AutoTest was integrated into EiffelStudio [43].

Continuing this line of work, the same authors created AutoFix-E [51], which works on Eifell [42] classes containing contracts. AutoFix-E takes the tests generated by AutoTest as input and executes them to detect contract violations. From both passing and failing executions, the tool creates snapshots, which are abstract representations of program states.

FL in AutoFix-E is then performed by identifying suspicious snapshots. Each snapshot is assigned a suspiciousness score computed from both static (e.g., distance between program statements) and dynamic criteria (e.g., program's runtime behaviour). The following steps related to the fixing of the fault only take into consideration snapshots that ranked high in the previous phase.

AutoTest and AutoFix-E show that contracts can detect failures dynamically, but they do not

distinguish whether failures originate from implementations, specifications, or verification limitations.

This distinction is addressed by a later study [53], which proposes a combined verification and testing methodology to classify proof failures. In this approach, failed proofs are analysed using targeted test generation to search for both the motive behind the failure and concrete counterexamples. Depending on the outcome, proof failures are diagnosed as "non-compliance between code and specification", "weaknesses in subcontracts", "prover incapacity", "likely prover incapacity", or "unknown". By automating this process, this work also aims to make verification-aware languages accessible to non-expert users.

Similarly exploring the intersection of Formal Methods and debugging, "Angelic Debugging" [10] was introduced, using the Java PathFinder model checker [25]. This method employs symbolic execution to identify "flexible" expressions that can be altered to fix a failing test without breaking any passing ones. However, the analysis does not cover the entire implementation. The first step consists on defining a search scope (set of lines) for the bug, which can be achieved manually, or from automated tools powered, for instance, through SBFL. Then, by replacing concrete values within this scope with angelically non-deterministic statements, the system explores the execution space to locate faults, and later, fix them.

While the previous approaches focus on detecting and diagnosing single faults, others [23] have proposed a combination of spectra-based fault localisation and specification-based analysis, that allow for the identification of multiple faults. Contrary to purely spectra-based methods that rely solely on passing and failing test runs, their SAT-TAR framework leverages Boolean Satisfiability Problem (SAT) technology to perform an unsatisfiability analysis on violated specifications. By computing unsatisfiable cores, the tool accurately isolates likely faulty statements in Java programs, instrumented with JFSL [78] or JML [11] specifications, showing how formal contracts can help refining test-driven localisation.

The former methodologies largely assume that the necessary specifications are already present. However, in practice, proof failures in verification-aware languages frequently arise from multiple missing or insufficient annotations, such as loop invariants or intermediate assertions. Recent work [59] tackles this issue by comparing and combining heuristic-based and LLM-based methodologies for FL, when one, two, or multiple assertions are missing. They concluded that LLMs are equally and sometimes more capable of identifying the position of missing assertions in Dafny code. Nevertheless, the best results were actually achieved once the combination of both methods (heuristic and LLM-based) was applied.

Complementary to assertion inference, recent work on APR for Dafny [74] also combines deterministic and non-deterministic methods to localise and correct faults in Dafny code. From the specification, their tool applies Hoare Logic [26] to evaluate how the program state changes after each statement. This FL step outputs a ranking of the suspicious lines, which is then used by the LLM to guide the repair phase. The generated candidate fixes are evaluated by re-running the Dafny verifier, ensuring that corrected programs satisfy all previously specified contracts. Limitations of this work involve programs that repeatedly update the same variable and the termination

verification of while loops.

In short, existing approaches show that contracts can serve as effective fault detectors, that testing can help diagnose the causes of failed proofs, and that faults can be addressed through automated techniques. Each generation of tools has incrementally reduced the manual burden of FL. However, as these techniques grow in complexity, distinct limitations emerge regarding their scalability and determinism. A critical discussion of these limitations and the gaps remaining in the current state of the art follows.

3.1.2 Critical Discussion

This subsection critically analyses the findings described previously.

Firstly, a fundamental challenge in verification-aware FL is determining what constitutes the "ground truth". Most seen approaches operate on the assumption that the specification is correct and the implementation is at fault. By taking the specification as a rigid oracle, one is discarding the possibility that the error might actually be in the contract. Petiot et al. [53] contrast with this idea, by acknowledging that the failure is often due to a "weak specification" or "prover incapacity" rather than a code bug. Although it was not retrieved by our systematic search, recent work by Amaral et al. on MutDafny [3] explicitly addresses this gap. MutDafny applies mutation testing to Dafny programs specifically to evaluate the quality of the contracts, allowing developers to detect weak specifications that fail to catch injected faults. However, even these diagnostic and mutational classifications rely on the existence of a specification to test against. There is a lack of robust methodologies for scenarios where the specification is fundamentally contradictory or mathematically impossible to satisfy.

Secondly, approaches that may depend on SBFL [10, 23] are heavily reliant on the quality of the underlying test suite. If the tests do not adequately cover the execution paths, even the most rigorous specification-based unsatisfiability analysis or angelic value will struggle to locate the fault accurately.

Thirdly, verification-aware languages differ from standard programming languages because they require additional annotations (i.e., loop invariants or assertions). This explains the recent pivot towards LLMs [59], since heuristic methods have a harder time guessing the location of a missing invariant from scratch, whereas Generative Artificial Intelligence (AI) can propose candidates. A threat, however, is that there is the risk of data leakage when working with open-source datasets, which might positively affect the LLMs' performance.

Lastly, current tools still require significant human input. Most approaches are limited to the identification/correction of single errors in the code, which is not realistic for a novice programmer.

Because FL techniques in verification-aware languages are so sparse, it became evident that answering **RQ 1** required a shift in perspective. Testing is historically the primary strategy for deterministic fault discovery. Yet, even within the verification context, most identified approaches disregard or at least do not take full advantage of the specifications for test generation. Therefore, to address the gaps identified in our systematic search, we conducted a targeted review of specification-driven test generation tools.

3.2 Specification-Driven Test Generation[9]

Specification-driven test generation is not a recent invention. In 1993, Dick and Faivre [19] proposed Disjunctive Normal Form (DNF) decomposition for model-based specifications, in order to systematically isolate testing sub-domains. Their work demonstrated that expanding logical specifications into mutually exclusive and purely conjunctive disjunctions naturally yields distinct equivalence classes. However, when generating executable tests, this logical expansion must respect the underlying execution semantics of the language. As highlighted by Chalin [9], assertion languages must preserve short-circuit evaluation and enforce "strong validity" to prevent runtime crashes when partial functions, such as array accesses, are evaluated outside their valid domains.

More recently, and besides DTest [20], described in Chapter 2, Delfy [12] also generates tests for Dafny based on the implementation. It uses dynamic symbolic execution with Z3 to diagnose verification failures. When a verification condition cannot be proven statically, Delfy uses the Z3 solver to generate a concrete input value that drives the execution until that point. Other concolic tools, such as PathCrawler [69] for C (within the Frama-C [30] framework), also systematically generate test inputs but are designed for structural coverage.

Furthermore, as seen in the systematic review, there are black-box testing frameworks like AutoTest [44] that combine random sampling with existing contracts to generate tests for object-oriented code.

Other specification-driven test generators apply related techniques to formal contracts. Peña et al. [52] use Z3 to derive test inputs from preconditions in a custom specification language. Later, Gil et al. [22] target SPARK/Ada [40] using ECP guided by user-written `Contract_Cases` and external calls to Z3py.

Additionally, Rebello de Andrade et al. [55] apply Full DNF to axiomatic specifications of Java generics and use Alloy Analyzer (a SAT solver) to find satisfying models for each minterm, but do not actually target a verification-aware language.

Finally, property-based testing tools (e.g., QuickCheck [13], Hypothesis [37]) also generate tests from properties, but these are executable, partial, and input-centric, not full correctness specifications, meaning that they constrain observable behavior over sampled executions rather than fully defining system-wide correctness.

In summary, while automated testing techniques are gaining traction, they primarily fall into two categories: implementation-dependent approaches (such as dynamic symbolic execution) or random, black-box strategies. Even tools that use contracts or properties often target non-verification-aware languages or rely on partial specifications rather than full correctness constraints. This leaves a significant gap in the context of Dafny. Currently, there is no native framework that derives test suites purely from formal specifications, nor an automated approach for FL in the context of Dafny.

[9] Portions of the text in this section have been adapted from the tool paper: Pinto, S. V., et al., "DSpec2Test: Specification-Driven Test Generation in Dafny" [64], accepted at the IEEE/ACM International Conference on Automated Software Engineering (ASE) 2026.

3.3 Answer to RQ 1: Which fault localisation techniques are currently available for verification-aware programming languages?

RQ 1: The literature demonstrates that FL in verification-aware programming languages is emerging but remains scarce. The retrieved studies broadly fall into three categories: dynamic testing, FL via symbolic execution and satisfiability solvers, and LLM-assisted inference. Firstly, dynamic approaches like AutoTest [44] and AutoFix-E [51] generate contract-based tests with random inputs to score the suspiciousness of program snapshots based on runtime behavior. Additionally, Petiot et al. [53] use targeted testing to classify proof failures into different categories. Later, strategies using symbolic execution and satisfiability solvers to isolate defects without relying solely on concrete executions emerged. For instance, Angelic Debugging [10] uses symbolic execution to identify expressions that mathematically fix failing paths, while Gopinath et al. [23] compute unsatisfiable cores on violated specifications to refine conventional spectra-based FL. More recently, LLMs have been applied to address missing annotations. Silva et al. [59] combine generative AI with heuristics to infer missing assertions in Dafny, while Wu et al. [74] pair Hoare Logic state evaluation with LLMs to guide the repair of arithmetic errors while respecting contractual requirements.

Given the sparsity of FL techniques in this domain, and since existing methods, like AutoFix-E, already demonstrate the value of testing for FL, addressing this research question requires a deeper exploration of automated test generation. Testing approaches generally split into implementation-dependent and specification-driven strategies. White-box tools, such as Delfy [12] and DTest [20], use dynamic symbolic execution to diagnose verification failures based on source code. Conversely, specification-driven tools prioritise formal contracts to derive test inputs, ranging from black-box random samplers like AutoTest to methodologies applying logic decomposition, such as DNF expansion.

Despite this variety, current FL techniques are still heavily reliant on pre-existing and implementation-derived test suites. Concurrently, available automated testing techniques frequently prioritise structural over specification coverage or target languages entirely outside the verification-aware ecosystem. Ultimately, this review reveals a lack of an automated pipeline for Dafny that generates test suites purely from formal contracts to drive FL.

Chapter 4

Methodology

This chapter outlines our approach that will ultimately lead to localising faults in Dafny. We start by detailing DSpec2Test, a tool that automatically generates tests from the specification of Dafny programs. We then describe the code coverage plugin for Dafny that was developed so that FL strategies for ranking and identifying the most probable faulty line could be employed.

4.1 Test Generation: DSpec2Test[1]

DSpec2Test is a tool for specification-driven test generation in Dafny. It is provided through an additional mode (*Spec*) for Dafny's `generate-tests` command, detailed in Section 2.2.1.1. It analyses the preconditions (`requires`) and postconditions (`ensures`) of the methods and functions to apply the well-established black-box testing strategies of ECP (through DNF), and, optionally, BVA [46]. To the best of our knowledge, there are no other Dafny tools that allow the automatic creation of tests prior to the program's implementation being written. DSpec2Test relies on an SMT solver (Z3) to find concrete test data that satisfy the target test conditions produced by ECP and BVA. The tool's source code can be found in its own GitHub repository, which is an adapted version of Dafny's v4.11.0 release.

4.1.1 DNF-Based Equivalence Class Partitioning

ECP consists of dividing input data into partitions, called equivalence classes, where all values in a class are expected to behave in the same way [46]. We automate ECP by analysing the method's contract clauses and converting them into DNF. This transformation takes place because expanding a logical specification into a disjunction of conjunctions systematically isolates the formula into mutually exclusive and purely conjunctive sub-domains. This naturally yields distinct equivalence classes and ensures that nested conditions are fully exposed, rather than masked, guaranteeing logic coverage [19].

[1] Portions of the text in this section have been adapted from the tool paper: Pinto, S. V., et al., "DSpec2Test: Specification-Driven Test Generation in Dafny" [64], accepted at the IEEE/ACM International Conference on Automated Software Engineering (ASE) 2026.

Alternative behaviours (depending on different input classes) are typically encoded in the method's postconditions. Algorithmically, DSpec2Test isolates the preconditions (`requires`) and postconditions (`ensures`), applying the DNF expansion to each set independently. The tool then calculates the cross-product of these expanded pre and postcondition branches. Each resulting combination constrains both inputs and outputs, allowing us to generate concrete values for the test inputs.

Concrete expected outputs may also be presented when uniquely determined by the contract, by applying the `-simplify` flag. This option replaces the methods postconditions with concrete instantiations of the output values, simplifying the tests' output, and making them more easily interpretable. A specific example of this behaviour can be found in Section 4.1.4.

Listing 4.1: Example of a Dafny contract for the `ConditionalGet` method. This contract requires a not empty input sequence, and guarantees that the output variable `r` will always be defined by, at least, the input parameter `x`: if `x` is a negative integer, then `r` will be -1; if `x` is higher than the size of sequence `s`, then `r` will be -2; if x is within the bound of `s`, then `r` will be the value of `s` at position `x`

```
1 method {:testEntry} ConditionalGet(s: seq<nat>, x: int) returns (r: int)
2   requires |s| > 0
3   ensures x < 0 ==> r == -1     // Q1
4   ensures x >= |s| ==> r == -2  // Q2
5   ensures 0 <= x < |s| ==> r == s[x] // Q3
```

Conversion to DNF must be done carefully: a naive expansion of an implication $A \Rightarrow B$ as $\neg A \vee B$ produces a branch in which B is evaluated without the guard A, which may cause runtime crashes. For example, naively expanding Q_3 in Listing 4.1 yields a branch where $r = s[x]$ is evaluated without the guard $0 \leq x < |s|$, raising an out-of-bounds error. To avoid this, DSpec2Test defaults to *Safe DNF*, inspired in Chalin's principle of "strong validity" [9]. By enforcing the short-circuit-safe decomposition rules of Table 4.1, the conversion guarantees that partial functions remain protected by their guards, producing mutually exclusive branches.

Table 4.1: Safe DNF decomposition rules.

Operator	**Branches (mutually exclusive)**
$A \Rightarrow B$	$\{\neg A,\ A \wedge B\}$
$A \vee B$	$\{A,\ \neg A \wedge B\}$
$A \Leftrightarrow B$	$\{A \wedge B,\ \neg A \wedge \neg B\}$
`if A then B else C`	$\{A \wedge B,\ \neg A \wedge C\}$

Applied to `ConditionalGet`, in Listing 4.1, each implication splits into two cases:

- Q_1: $\{x \geq 0,\ x < 0 \wedge r = -1\}$
- Q_2: $\{x < |s|,\ x \geq |s| \wedge r = -2\}$
- Q_3: $\{\neg(0 \leq x < |s|),\ 0 \leq x < |s| \wedge r = s[x]\}$

The cross product, conjoined with the precondition $|s| > 0$, yields eight candidate equivalence classes. Five are infeasible due to mutually exclusive guards. To optimise performance and prevent sending trivially impossible combinations to the verifier, DSpec2Test employs a preliminary contradiction check during the cross-product generation. If an equivalence class is identified as logically contradictory at this stage, it is immediately pruned. Any remaining hidden infeasibilities are ultimately caught when Z3 evaluates the instrumented code and returns UNSAT.

For the `ConditionalGet` example, the three feasible classes, after simplifying the redundant guard negations, are:

- C_1: $|s| > 0 \quad \wedge \quad x < 0 \quad \wedge \quad r = -1$ // below range
- C_2: $|s| > 0 \quad \wedge \quad x \geq |s| \quad \wedge \quad r = -2$ // above range
- C_3: $|s| > 0 \quad \wedge \quad 0 \leq x < |s| \quad \wedge \quad r = s[x]$ // in range

If exhaustive combination testing is required and can be safely applied, DSpec2Test also supports *Full DNF* (with the `-fdnf` flag), which drops short-circuit safety and forces every n-ary disjunctive operator to produce all $2^n - 1$ non-empty subsets of branch satisfaction. For instance, $A \vee B$ expands to $\{A \wedge B, A \wedge \neg B, \neg A \wedge B\}$. Although this exercises all combinations of independent disjuncts, it requires caution: it may generate branches that attempt to evaluate subexpressions outside their valid domain. For example, if applied to Q_3, this option bypasses the $0 \leq x < |s|$ guard, meaning $r = s[x]$ could be evaluated with any arbitrary value of x, potentially raising an out-of-bounds error. Nevertheless, Z3 fails gracefully. Therefore, it does not visibly raise any error, simply skipping that test scenario.

4.1.2 Boundary Value Analysis

While equivalence class partitioning provides broad coverage, defects (such as off-by-one errors) often hide at the edges of these partitions. For that reason, we added support for Boundary Value Analysis (BVA) to DSpec2Test, which complements the DNF phases when the `-bva` flag is provided.

For every partition, the tool generates multiple test scenarios. The first test leaves all variables free, relying only on the base DNF constraints (behaving exactly as if `-bva` were omitted). Subsequent tests pin exactly one variable to a boundary value at a time, leaving all others free. This prevents conflicting constraints while exploring extreme edge cases.

The pinned boundary values are determined by each partition's constraints. If the partition restricts a variable, the tool pins it to its valid edges. For discrete integer types with exclusive bounds (e.g., $-1 < x \leq 50$), the tool calculates the nearest valid integer (attempting $x = 0$ and $x = 50$, in this case). For continuous numeric types (reals and floats), where a "nearest" value is infinitely small, DSpec2Test approximates the exclusive boundary by applying a minimal offset of 0.0001 from the strict limit. If a variable lacks a lower or upper bound (e.g., $x \leq 0$), the tool substitutes the missing edge with an extreme value. By default, the tool evaluates to a symmetric interval

$[-N,N]$, where N is the maximum 32-bit signed integer (2,147,483,647). A symmetric lower bound (-2,147,483,647) is deliberately chosen over the true 32-bit minimum (-2,147,483,648) to simplify scaling: when a user overrides the default by passing an integer to the `-bva` flag, the tool simply replaces N with the provided value to construct the new extremes.

While this symmetric interval serves as the default for unbounded integers and reals, the tool adapts its fallback boundaries based on the variable's underlying data type. For instance, Boolean variables are tested for both `true` and `false` states. Similarly, unsigned BitVectors (`bv`) lack negative bounds. Therefore, their missing lower bounds always defaults to 0, and for bit-widths under 64, their upper bounds default to their maximum bitwise capacity (i.e., $2^{bits}-1$).

Notably, the boundary generation engine respects explicit inequality constraints (e.g., $x \neq 0$). During the DNF cross-product phase, DSpec2Test records all exclusionary conditions into a constraint tracker. If a calculated boundary value collides with an explicit exclusion, the tool automatically discards that specific boundary test. This guarantees that generated edge cases do not trivially violate the partition's underlying domain.

Additionally, sequences, sets, and maps are tested for lengths 0, 1, and > 1. If a partition already restricts a numeric variable or collection size to an exact value, the tool does not generate additional boundary cases for it.

For instance, applied to class C_3 of `ConditionalGet`, BVA emits four additional scenarios for each partition, with the following constraints: $x = 0$ (minimum), $x = |s|-1$ (maximum), $|s| = 1$, and $|s| > 1$.

4.1.3 Additional Remarks

DSpec2Test also addresses some limitations of the original DTest tool presented in Section 2.2.1.1. Firstly, it eliminates the requirement of encapsulating the methods or functions to be tested in a module. Secondly, it does not only generate tests for methods/functions with the `{:testEntry}` attribute, but also for those that fail verification, directly facilitating the debugging process.

Furthermore, DSpec2Test provides additional useful flags for test generation. Besides `-bva` and `-simplify`, which are specific to the *Spec* mode, DSpec2Test features the `-repeat` and `-time` flags, available for any strategy. The `-repeat` flag enables the generation of multiple rounds of tests, each time with distinct input values. The selected test generation strategy is executed as many times as specified by this parameter, while guaranteeing the uniqueness of the newly generated tests. If, for any reason, new input values cannot be created, test generation terminates gracefully to avoid redundant tests.

Finally, the purpose of the `-time` is purely informative. It prints comments in between the different rounds of generated tests, making it easier to identify which tests were produced by each repetition while also highlighting the cumulative time taken to generate them.

4.1.4 Concrete test data generation

Like DTest, DSpec2Test relies on Z3 SMT solver (via Dafny/Boogie) to find concrete test data. The core mechanism is an iterative process: DSpec2Test dynamically instruments the method's body with specific `assume` statements to guide the solver, followed by an `assert false` to force the verifier to emit a counterexample containing the concrete inputs and expected outputs.

Before the dynamic instrumentation even begins, DSpec2Test strips the target method of its original implementation, reducing it to an empty block. This guarantees that the test generation is driven entirely by the specification, preventing the solver from being constrained or confused by the program's actual (and potentially faulty) implementation.

Additionally, prior to applying the DNF expansion, DSpec2Test filters the extracted contract clauses to isolate the actual user-defined specifications. Because Dafny compiles high-level code into Boogie's intermediate representation, methods often contain system-generated constraints regarding memory allocation and state (e.g., variables like `$Heap` and `$Tick`). DSpec2Test explicitly ignores these internal variables, as well as trivial boolean literals, ensuring that the test generation is driven purely by the developer's logical constraints.

Then, the actual test generation cycle consists in: for every valid class in the DNF decomposition, the method body is freshly instrumented with an `assume` and `assert false` statements. Once a test is generated for a class, this instrumentation is discarded to provide a clean slate for the next iteration.

However, for this process to work, an additional `assume` statement referencing a new state (of the form `{:captureState_<Method>_<N> "SpecComb_<Method>_<N>"}`,where `N` is the test's index in that particular repetition) is injected prior to the DNF instrumentation. Because the native `generate-tests` command is strictly designed for block/path testing, this annotation tricks the pipeline into treating our synthetic constraints as a distinct and reportable execution state. Without it, the solver would generate the test data internally but refuse to print it.

For example, to generate a test for class C_3 of `ConditionalGet`, without any boundary constraints, the tool instruments the body to strictly enforce the DNF branch conditions (Listing 4.2).

Listing 4.2: Instrumented body for class C_3. The original method implementation is replaced with a state-capturing assumption, the logical constraints defining the C_3 equivalence class, and a failing assertion to force counterexample generation.

```
method {:testEntry} ConditionalGet(s: seq<nat>, x: int) returns (r: int)
  // contract unchanged
{
  assume {:captureState_ConditionalGet_0 "SpecComb_ConditionalGet_0"} true;
  assume |s| > 0 && 0 <= x < |s| && r == s[x];
  assert false;
}
```

In Listing 4.2, Z3 is constrained to the valid domain of C_3, but it is free to pick any valid array and any index within bounds. When BVA is enabled, this per-class cycle expands into an inner loop. For every boundary value uncovered in a class, DSpec2Test builds upon this baseline by

iteratively appending strict equality constraints to the `assume` statement. This process "pins" the DNF branch to a specific boundary. For instance, in C_3, the constraint $0 \leq x < |s|$ establishes 0 as the lower bound for x. To test this exact edge, DSpec2Test injects the explicit constraint `x == 0`, as shown in Listing 4.3.

Listing 4.3: Instrumented body for class C_3, with BVA. An explicit equality constraint (`x == 0`) is appended to the equivalence class constraints to force the solver to generate test data precisely at the calculated lower boundary.

```
method {:testEntry} ConditionalGet(s: seq<nat>, x: int) returns (r: int)
  // contract unchanged
{
  assume {:captureState_ConditionalGet_0 "SpecComb_ConditionalGet_0"} true;
  assume |s| > 0 && 0 <= x < |s| && r == s[x] && x == 0;
  assert false;
}
```

By appending this pin, DSpec2Test forces Z3 to yield a counterexample that specifically exercises the boundary condition (e.g., `s=[3]`, `x=0`, `r=3`). As shown in Listing 4.4 for C_3, when the assumptions are feasible, this counterexample is then materialised into a runnable Dafny test. If the `-simplify` flag is set, and the postconditions uniquely constrain the outputs, DSpec2Test hardcodes the concrete expected values produced by Z3 directly into the assertions.

Listing 4.4: Generated test for class C_3 using the `-simplify` flag. The tool hardcodes the concrete expected outputs produced by Z3 (e.g., `expect r == 3`) directly into the test assertions rather than relying on postconditions.

```
method {:test} Test_ConditionalGet_C3() {
  var seqint0 : seq<int> := [3];
  var r := ConditionalGet(seqint0, 0);
  expect r == 3;
}
```

Alternatively, the default behaviour relies on the original postconditions, instantiated for the known inputs, to evaluate the method's correctness, as shown in Listing 4.5.

Listing 4.5: Generated test for class C_3. The test evaluates correctness by asserting the original method postconditions, dynamically instantiated with the generated concrete inputs.

```
method {:test} Test_ConditionalGet_C3() {
  var seqint0 : seq<int> := [3];
  expect |seqint0| > 0;
  var r := ConditionalGet(seqint0, 0);
  expect 0 < 0 ==> r0 == -1;
  expect 0 >= |seqint0| ==> r0 == -2;
  expect 0 <= 0 < |seqint0| ==> r0 == seqint0[0];
}
```

Finally, this iterative manipulation of `assume` statements is also how DSpec2Test generates multiple tests with distinct inputs for the same method when using the `-repeat` flag. Instead of appending boundary pins, the tool re-runs the pipeline while appending `assume` constraints

that explicitly negate the inputs generated in every test of all previous iterations, as illustrated in Listing 4.6.

Listing 4.6: Iterative instrumentation for `ConditionalGet` using the `-repeat` flag. Exclusions (e.g., `assume x != 0`) regarding every input value are prepended to the body to prevent Z3 from yielding previously generated inputs, followed by the base class constraints.

```
1  method {:testEntry} ConditionalGet(s: seq<nat>, x: int) returns (r: int)
2    // contract unchanged
3  {
4    assume x != 0;
5    assume s != [3];
6    assume x != 1;
7    assume s != [0];
8    assume x != -1;
9    assume s != [0];
10   assert true;
11   assume {:captureState_ConditionalGet_0 "SpecComb_ConditionalGet_0"} true;
12   assume |s| > 0 && 0 <= x < |s| && r == s[x];
13   assert false;
14 }
```

In this scenario, the `assume` statements preventing the generation of previous input values (lines 4 to 9) are placed at the top of the body, immediately followed by the baseline `assume` (lines 11 and 12), and `assert false` (line 13) constraints of the target class (in this case, class C_3). With every subsequent run, DSpec2Test chains additional exclusions to the top of the body, iteratively forcing Z3 to discover novel test cases until the requested number of repetitions is reached, or no distinct input value can be found. Note that BVA boundary pins are not calculated or inserted during these repeat cycles, as all boundary values are exhaustively exercised during the first iteration.

While this explicit negation works seamlessly for primitive types (integers, booleans) and collections of primitives, DSpec2Test employs a specialised heuristic for some User-Defined Types (complex objects and class instances, except natural numbers and tuples), as neither DTest nor DSpec2Test fully supports them. Because these object instances cannot be trivially compared by value, DSpec2Test filters them during the repeat cycle. If an input parameter is a singular object instance, it is excluded from the negation constraints to prevent solver crashes. If the parameter is a collection of objects (e.g., a sequence or set), DSpec2Test negates the collection's cardinality (e.g., $|s| \neq 1$) rather than its literal contents. This ensures the solver continues to explore novel program states safely without violating Boogie's memory model.

4.1.5 Implementation Pipeline

Figure 4.1 summarises how the components of DSpec2Test, which make up the newly created *Spec* mode, are integrated into Dafny's `generate-tests` command. The process is done in three stages.

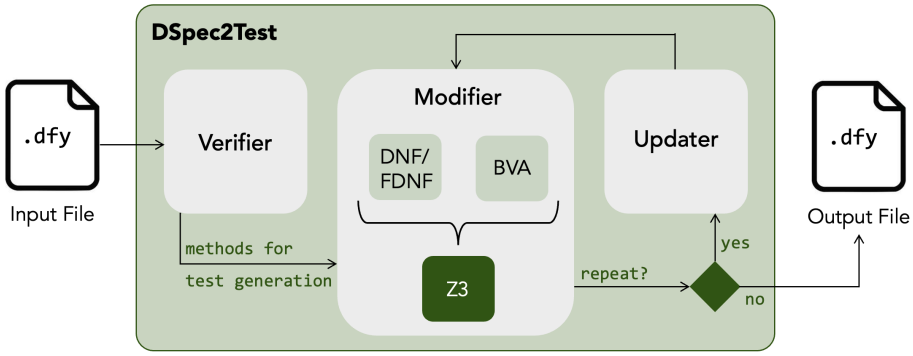


Figure 4.1: DSpec2Test pipeline. An input file is processed by the *Verifier*, where methods that are marked with `{:testEntry}` or fail verification are passed to the *Modifier* for DNF generation and optional BVA. The *Updater* iteratively refines the process by generating additional tests while ensuring novelty through added `assume` clauses.

The *Verifier* flags for test generation every method that fails verification or is annotated with `{:testEntry}`. The implementations of these methods are then discarded, since they are not used for specification-based test generation.

The *Modifier* instruments each flagged method with the DNF classes and, when `-bva` is selected, adds additional BVA boundary pins described in Section 4.1.2, as illustrated in Listing 4.3.

The *Updater* drives iteration when `-repeat` is higher than one. After each generated test, it appends a fresh `assume` clause for each previously generated input, so that the next solver call returns different ones and, therefore, distinct tests. For instance, if `x` was assigned to `0`, an `assume x != 0;` statement is added before the next iteration. Regardless of `-bva` being selected, BVA will not be applied to the subsequent calls to `-repeat`, since all boundary values have already been exercised during the first iteration.

4.2 Per-Test Code Coverage Plugin

To achieve SBFL through tests, we first need to obtain the code coverage of each individual test. Although Dafny compiles to languages like C# and Java, which both have standard test-coverage tools (like Coverlet[2], and Cobertura[3], respectively), this compiled-language coverage does not easily map back to the exact lines in the original Dafny source code. Moreover, the existing Dafny's test coverage functionality associated with the `generate-tests` command is still experimental, and it produces the combined coverage of all tests in a file, which is not the information we need for FL. For these reasons, we created a custom per-test code-coverage plugin for Dafny, stored in the `coverage` folder of the dissertation's repository.

This tool receives a Dafny file as input. This file should contain both the tests and the program to be tested, or include the target file at the top. In our case, since the previous stage of the pipeline

[2]Coverlet's webpage https://github.com/coverlet-coverage/coverlet, accessed 01 Jun 2026.
[3]Cobertura's webpage https://cobertura.github.io/cobertura/, accessed 01 Jun 2026.

already combined the programs and their tests into a single file, we use these combined files, as illustrated in Listing 4.7.

Listing 4.7: Example of a dafny file combining the program to be tested (`Classify`) and the corresponding tests generated by DSpec2Test.

```
method {:testEntry} Classify(x: int) returns (r: int)
  requires x >= -100 && x <= 100
  ensures x < 0 ==> r == -1
  ensures x == 0 ==> r == 0
  ensures x > 0 ==> r == 1
{
  if x > 0 { //Buggy line: Should be x < 0
    r := -1;
  }
  else if x == 0 {
     r := 0;
  }
  else {
     r := 1;
  }
}

method {:test} Test0() {
  var r0 := Example.Classify(1);
  expect 100 < 0 ==> r0 == -1;
  expect 100 == 0 ==> r0 == 0;
  expect 100 > 0 ==> r0 == 1;
}
method {:test} Test1() {
  var r0 := Example.Classify(0);
  expect 0 < 0 ==> r0 == -1;
  expect 0 == 0 ==> r0 == 0;
  expect 0 > 0 ==> r0 == 1;
}
method {:test} Test2() {
  var r0 := Example.Classify(-1);
  expect -1 < 0 ==> r0 == -1;
  expect -1 == 0 ==> r0 == 0;
  expect -1 > 0 ==> r0 == 1;
}
```

The methods, such as `Classify` (Listing 2.1), are instrumented with temporary print statements, as depicted in Listing 4.8, that allow us to track the execution line by line.

Listing 4.8: Logical representation of the AST after the coverage plugin instruments the `Classify` method.

```
method {:testEntry} Classify(x: int) returns (r: int)
  requires x >= -100 && x <= 100
  ensures x < 0 ==> r == -1
  ensures x == 0 ==> r == 0
  ensures x > 0 ==> r == 1
{
  print "COVERAGE_LINE: 6\n";
  if x > 0 { //Buggy line: Should be x < 0
    print "COVERAGE_LINE: 8\n";
    r := -1;
```

```
  }
  else {
    print "COVERAGE_LINE: 10\n";
    if x == 0 {
      print "COVERAGE_LINE: 11\n";
      r := 0;
    }
    else {
      print "COVERAGE_LINE: 13\n";
      print "COVERAGE_LINE: 14\n";
      r := 1;
    }
  }
}
```

Note how the `else if` is transformed into a nested block to capture precise branch execution. Furthermore, the `else` block is instrumented with two print statements to allow for the identification of the `else` statement itself. However, the proposed coverage instrumentation is strictly restricted to methods. In Dafny, functions and predicates are mathematically pure and cannot accommodate imperative side effects. Consequently, injecting print statements into these pure contexts would violate Dafny's static semantics, triggering immediate compiler errors.

Additionally, to prevent infinite `while`-loops or recursive calls from halting test execution, the plugin allows for a configurable input parameter (`max_iter`) that restricts the maximum recursion depth and amount of loop iterations (defaulting to 1,000).

In the Dafny ecosystem, plugins serve as a mechanism to extend the compiler's behavior without requiring modifications to the core source code. They allow developers to intercept the compilation pipeline at specific stages [62]. In our case, the plugin is integrated directly into Dafny's testing pipeline via the command-line interface (e.g., `dafny test <file> -plugin <path> -no-verify`). By implementing Dafny's `PluginConfiguration` interface, the tool hooks into the compiler during the Abstract Syntax Tree (AST) resolution phase to execute the instrumentation. This allows the plugin to traverse the resolved syntax tree and inject the required print statements prior to compilation. Furthermore, the inclusion of the `-no-verify` flag allows the pipeline to bypass Dafny's formal verification engine, which not only reduces the execution overhead, but also allows for the generation of test coverage for buggy programs.

We then developed a Python wrapper script that executes this plugin across a directory of Dafny files. This script outputs the aggregated coverage data as a JSON dictionary. At the root level, the keys are strings representing the file names. Each file name maps to a nested object where the keys are test identifiers (e.g., "Test0"). Finally, these test identifiers map to an object containing two fields: `passed`, a boolean indicating whether the test executed successfully or not, and `coverage`, an array of integers representing the specific lines executed by that test. Listing 4.9 depicts this structure.

Listing 4.9: Structure of the output JSON file generated by the script running the coverage plugin.

```
{
  "filename1.dfy": {
    "Test0": {
```

```
      "passed": true,
      "coverage": [9, 10, 15]
    },
    "Test1": {
      "passed": false,
      "coverage": [9, 10, 17, 18, 19]
    }
  },
  "filename2.dfy": {
    "Test0": {
      "passed": true,
      "coverage": [6, 9, 10, 12]
    }
  }
}
```


Our plugin effectively solves the challenge of mapping compiled-language coverage back to Dafny source code. By generating reliable execution traces for every test in a file, it outputs the precise coverage data required to enable our subsequent SBFL pipeline.

4.3 Answer to RQ 2: How can these fault localisation techniques be applicable to Dafny?

RQ 2: To make existing FL techniques applicable to Dafny, it is essential to create an automated pipeline that generates both specification-based test suites and dynamic execution matrices.

First, test-suites can be generated from formal specifications by translating a method's contract into distinct, mathematically verifiable constraints. The process isolates user-defined preconditions and postconditions, expanding them into mutually exclusive sub-domains using a short-circuit-safe DNF. This ECP is then optionally augmented with BVA to exercise extreme edge cases. After pruning trivially infeasible combinations, the remaining valid domains are fed to an Satisfiability Modulo Theories (SMT) solver (Z3) to resolve the constraints into concrete test data. To this end, we implemented DSpec2Test as a new mode (*Spec*) for Dafny's native `generate-tests` command. To guarantee that test generation is strictly specification-driven, DSpec2Test strips the target method of its original implementation. It then dynamically instruments the clean AST) with `assume` statements enforcing the DNF constraints, followed by a failing assertion. This forces the verifier to emit a counterexample, which is finally extracted and parsed into a runnable Dafny test method.

To bridge the final gap to some FL techniques, a custom per-test coverage plugin is needed. This plugin enables the extraction of execution coverage by instrumenting the Dafny AST before compilation, producing per-test coverage information suitable for FL.

This pipeline provides the exact execution data required to successfully apply FL techniques directly to Dafny programs.

Chapter 5

Evaluation

Our evaluation is divided into three main phases. Firstly, DSpec2Test's performance is assessed through mutation testing. Secondly, SBFL metrics are employed to compare the performance of black and white box testing techniques in the process of detecting faults. Lastly, an LLM-based approach is applied to a subset of the original dataset to enable a direct comparison with traditional SBFL.

All files, scripts, and instructions needed to reproduce these experiments are publicly available. The test generation evaluation can be accessed inside the *Evaluation* folder, in DSpec2Test's repository. Meanwhile, the evaluations with the SBFL and LLM metrics are hosted in the dissertation's repository, which also includes the DSpec2Test code as a submodule.

5.1 Setup

All evaluations were conducted on an Apple MacBook Air running macOS Tahoe 26.5, equipped with an M1 processor (8-core CPU) and 16 GB of unified memory.

Furthermore, the software environment relied on the latest stable release of Dafny (v4.11.0), alongside its corresponding Z3 theorem prover (v4.12.1).

5.2 Dataset

For our evaluation, we use DafnyBench [36], currently the largest benchmark suite for Dafny, with over 750 programs. As our methodology relies on the assumption that the specifications are correct, but no currently existing process can ensure specification correctness, we measure the strength of the specifications and only retain as part of our dataset the ones that are above a certain threshold.

To objectively measure specification strength, we use MutDafny's [3] results. In the context of formal verification, a specification's quality can be measured by its ability to reject faulty implementations. We established a strict inclusion threshold of a 90% mutation score, calculated as:

$$\text{Mutation Score} = \left(\frac{\text{Killed}}{\text{Killed} + \text{Survived}} \right) \times 100 \tag{5.1}$$

While a high mutation score does not guarantee correctness, as it only tests against artificially introduced faults and cannot validate the programmer's original intent, it indicates that the specification is sufficiently rigorous to detect almost all mutations. Applying this filter narrowed the DafnyBench dataset to 430 programs with highly restrictive specifications.

It should also be noted that the current implementation of MutDafny does not eliminate equivalent programs, which means that, in some cases, the mutation score may be below 100% in cases where the specification is correct. This is due to the possibility that survived mutants are equivalent to the original implementation and, therefore, correctly verifies.

Due to the known limitations of the `generate-tests` command, we applied a secondary filter with regular expressions to exclude programs that lacked methods (four cases), as well as those containing arrays (including array2), classes, arrow types, or the non-deterministic `:|` operator (212 cases). This refinement resulted in a final evaluation set of 214 programs.

To enable the test generation process, we automatically injected the `{:testEntry}` attribute into every method across the selected programs. Finally, we used MutDafny to generate up to 10 mutants per program, while making sure to discard mutants equivalent to the original programs. This process yielded a total of 2,085 mutants, slightly below the theoretical maximum of 2,140, because some programs lacked sufficient statements to produce 10 distinct, valid mutants. As the mutant generation took place, a `ground_truth.json` file was being constructed to store which lines contained the mutation(s). Because MutDafny's internal processing slightly alters file formatting, we simultaneously processed the original programs without mutations. This ensured their formatting matched the mutants perfectly, so that lines with different indentation or syntax were not incorrectly flagged as a mutation.

5.3 DSpec2Test Mutation Testing Evaluation

To evaluate DSpec2Test, we employ mutation testing, a well-established fault-injection technique used to assess the quality and robustness of a test suite [27, 73]. Using this technique, we compared our approach against the existing modes available in Dafny's `generate-tests` command: *Block* and *Path*. Because the *Spec* mode supports an optional `-bva` flag, we generated tests both with and without it to isolate and evaluate its performance impact. Hereafter, we refer to the *Spec* mode executed with the `-bva` flag as the *SpecBva* mode, for simplicity. Consequently, our evaluation compares two white-box test generation modes (*Block* and *Path*) against two black-box modes (*Spec* and *SpecBva*). Additionally, we applied the newly implemented `-repeat` flag across all four modes, with values ranging from one to ten, to understand its impact on test generation and mutant kill rates.

5.3.1 Test Generation

Starting with the 214-program dataset described in the previous section, we ran the `generate-tests` command for each of the four modes using the configuration flags detailed in Table 5.1.

Table 5.1: Flags used for test generation across all modes.

Flag	Value	Description
`-repeat`	10	Attempts to generate 10 repetitions of tests with different input values
`-length-limit`	30	Maximum length of generated sets, maps, and sequences
`-time`	N/A	Prints elapsed time for each repetition
`-ignore-warnings`	N/A	Prevent termination due to non-critical warnings
`-solver-option`	`O:memory_max_size=3072MB`	Limits the underlying Z3 solver's memory allocation to 3GB

The `-length-limit` parameter dictates the maximum size of generated sequences, sets, and maps. While it was initially set to 50, the underlying Z3 solver occasionally threw non-fatal "Prover Error" messages due to the expanded state space. Lowering the threshold to 30 avoided these solver errors across the dataset while maintaining sufficiently rigorous test inputs. For the single program where this limitation still triggered prover errors, an inspection confirmed that the quality and validity of the generated tests were unaffected, so we removed these errors in a post-processing step. Furthermore, the `memory_max_size` value was calculated strategically based on the hardware specifications of the machine used for these experiments. We allocated memory using the formula:

$$\texttt{memory\_max\_size} = \frac{\text{Total RAM} \times 75\%}{\text{CPU Count} \div 2} \tag{5.2}$$

On our setup, this memory allocation allowed four jobs to run concurrently with 3 GB of RAM each, which was sufficient for Dafny and Z3 to operate reliably while maximizing parallelism. For the *SpecBva* mode, we set the `-bva` limit parameter to 100 (overriding the default of 2,147,483,647). This adjustment was made to constrain the input sizes in scenarios involving deep recursion or exponential complexity.

Despite our initial dataset filtration, some behaviours were impossible to exclude using regular expressions. This led to the failure of seven additional programs across all modes due to malformed code that only manifested at runtime (as shown in Listing 5.1), as well as non-deterministic branching (e.g., `if *{}`) or assignments (e.g., `var a := *;`) caused by the use of the `*` operator.

Listing 5.1: Predicate `sortedbad()` from `dafny-duck_tmp_tmplawbgxjo_ex3.dfy` program. The first `==>` raises a verifier warning regarding unusual indentation and suggests missing parentheses. However, the syntax error only causes a fatal failure at runtime.

```
predicate sortedbad(s: string)
{
  forall i, j :: 0 <= i <= j < |s| && s[i] == 'b' && s[j] != 'b' ==> i < j &&
  forall i, j :: 0 <= i <= j < |s| && s[i] != 'd' && s[j] == 'd' ==> i < j
}
```

Additionally, *Spec* and *SpecBva* modes failed in two other programs, presumably due to resource exhaustion. Nevertheless, failures were more prevalent in white-box modes: four programs failed in *Block* mode, and 38 failed in *Path* mode. The failures in *Block* mode originated from methods lacking implementations, the presence of `assert false` statements in the program's body, and Z3's known limitations with map comprehensions, which trigger quantifiers that overwhelm the solver (demonstrated in Listing 5.2).

Listing 5.2: `Clover_update_map.dfy` program example, where the heavy use of map comprehensions triggers quantifiers that overwhelm Z3.

```
method {:testEntry} update_map<K(!new), V>(m1: map<K, V>, m2: map<K, V>) returns (r: map<K,
V>)
// contract unchanged
{
  r:= map k | k in (m1.Keys + m2.Keys) :: if k in m2 then m2[k] else m1[k];
}
```

In addition to these four programs, *Path* mode failed on 28 others. These failures fall into two categories: time exhaustion and unexpected crashes. Twenty-one programs were terminated because they could not conclude test generation within a one-hour time limit, which we consider impractical for test generation. The remaining seven programs failed unexpectedly during execution, indicating a fatal abort likely caused by resource exhaustion.

Considering the partial overlap in failing programs across the different testing strategies (which will be detailed subsequently in Table 5.2), we excluded all programs that failed in any of the four modes. Therefore, we will consider 175 programs that successfully generated tests across all four modes in the upcoming steps of the pipeline.

5.3.2 Safety Check

The next step in our pipeline was the safety check. The safety check is a `dafny test` run, designed to verify whether the generated tests are well-formed and valid. Because this check is run against the original, unmutated programs, every generated test is expected to pass. If at least one test fails during this phase, the program is marked as failing the safety check and is discarded.

To optimise resources, the generated tests were split according to their repetitions, by taking advantage of the output logs produced by the `-time` flag, with the following structure: `// REPEAT 1 - TIME: 0.0s`. Therefore, if a test from an early repetition failed, execution was halted, sparing the need to evaluate the remaining repetitions for that program. This approach inevitably impacts the total safety check time metric, because it demands the call to `dafny test` at most ten times per program.

During this phase, the same 24 programs generated by the *Spec* and *SpecBva* modes were flagged as failing. While some timed-out due to massive branching tree of redundant function calls, others produced tests that violated method preconditions. The latter occurred particularly when the methods required nested collections as input parameters (e.g., `seq<seq<int>>`, or `seq<string>`, which is equivalent to a `seq<seq<char>>`). *Block* and *Path* modes failed the safety check on 15 and 17 programs, respectively, for similar reasons.

Notably, with the exception of a single program during the test generation phase, the failures observed in *Block*, *Spec*, and *SpecBva* (across both phases) are subsets of the programs that failed in *Path* mode. Table 5.2 summarises the program failures at each pipeline stage for all evaluated strategies.

Table 5.2: Program failures per pipeline stage across all modes.

Strategy	**Test Generation Failures**	**Safety Check Failures**	**Valid Programs**
Block	11	15	188
Path	38	17	159
Spec	9	24	181
SpecBva	9	24	181
All	39	17	158
All but *Path*	13	22	179

Therefore, a total of 158 programs (or 1541 mutants) out of the initial 214 pool advanced to the mutation kill check phase. This discrepancy is predominantly driven by the limitations of the *Path* mode. As officially documented in the Dafny Reference Manual [62], *Path* mode attempts to exhaustively cover all execution paths, which frequently results in immense execution times [62, Section 13.6.1.14]. In our evaluation, it was indeed solely responsible for 21 failures. Consequently, if *Path* mode was excluded from the comparative analysis, the pool of valid programs would expand to 179.

5.3.3 Mutation Kill Check

Following the safety check, we executed the generated tests from the valid programs against their corresponding mutants. This kill check step was performed using the `dafny test` command, together with the `-no-verify` flag (otherwise mutations would prevent compilation) across all ten repetitions to evaluate whether each mutant was killed, survived, timed out, or raised an error.

The execution followed the same sequential optimisation logic applied during the safety check, although now applied to mutants, instead of programs. This means that mutants were evaluated repetition by repetition. Because tests from one repetition build upon the tests generated in the preceding ones, if a mutant was killed or caused a timeout in an early repetition, it was marked as successfully killed, and the subsequent repetitions were skipped. This short-circuiting significantly reduces redundant computation and execution time.

5.3.4 Results and Discussion

For the following discussion, we define the Kill Rate as the percentage of mutants that did not survive the generated tests. While similar to the mutation score used for dataset filtering, this metric specifically encompasses both explicitly killed mutants and those that resulted in a timeout. It is calculated as follows:

$$\text{Kill Rate} = \left(\frac{\text{Killed} + \text{Timed Out}}{\text{Killed} + \text{Timed Out} + \text{Survived}} \right) \times 100 \tag{5.3}$$

To understand the impact of the `-repeat` flag, we tracked the evolution of the kill rate over the ten test generation repetitions. Table 5.3 details this progression for the 158 valid programs.

Table 5.3: Mutant kill rate percentages across ten repetitions for the 158 valid programs (1,541 mutants). Values in bold indicate the peak kill rate achieved by each testing strategy. The combined modes (*Block + SpecBva* and *Path + SpecBva*) highlight the potential of hybrid testing.

Strategy	***Block***	***Path***	***Spec***	***SpecBva***	***Block + SpecBva***	***Path + SpecBva***
Rep=1	70.02	64.57	64.89	88.06	89.81	89.42
Rep=2	85.01	84.75	84.82	90.66	91.50	91.43
Rep=3	86.44	87.02	87.15	90.91	91.56	91.82
Rep=4	87.35	87.61	87.8	91.17	91.69	**91.95**
Rep=5	87.35	87.93	88.06	91.43	91.76	91.95
Rep=6	88.06	88.12	88.19	**91.5**	91.76	91.95
Rep=7	88.19	88.25	88.25	91.5	91.76	91.95
Rep=8	88.19	88.25	88.38	91.5	91.76	91.95
Rep=9	88.38	**88.32**	88.38	91.5	**91.82**	91.95
Rep=10	**88.45**	88.32	**88.45**	91.5	91.82	91.95

Figure 5.1 highlights the superior behaviour of *SpecBva*, regardless of the target repetition. As shown in Table 5.3, it achieves an 88.06% Kill Rate at the very first repetition, a level that the other modes require at least four additional repetitions to match. This early top-performance indicates that BVA efficiently targets edge cases where mutations are most likely to alter program behaviour.

From repetition six onwards, the tests become highly redundant, since there is no noticeable improvement in the kill rate, with *SpecBva* reaching a peak of 91.50%, and the other modes plateauing near 88%. Moreover, although *Spec* begins as the worst performing strategy, it outperforms its white-box counterparts by repetition three. In addition, despite the immense computational cost of the *Path* mode, it actually yields results that are marginally worse than *Block*, in the last two repetitions.

Because *Path* heavily restricted our dataset, we wanted to analyse whether the 21 additional programs that worked for the other three modes would depict a different scenario. However, Figure 5.2 corroborates the previous findings, with *SpecBva* consistently outperforming the remaining strategies.

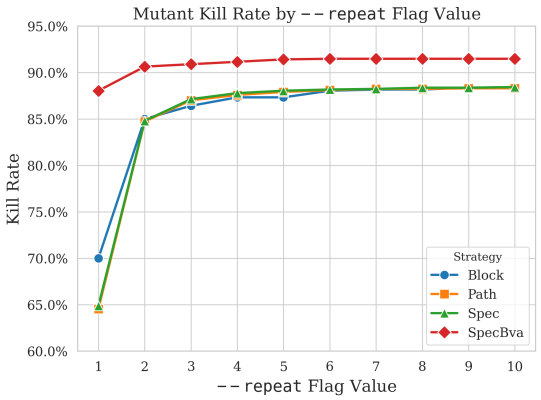


Figure 5.1: Mutant kill rate evolution for each strategy across ten repetitions (1,541 mutants, 158 programs). *SpecBva* consistently outperforms the others from the first repetition, while the remaining strategies plateau at similar, lower kill rates.

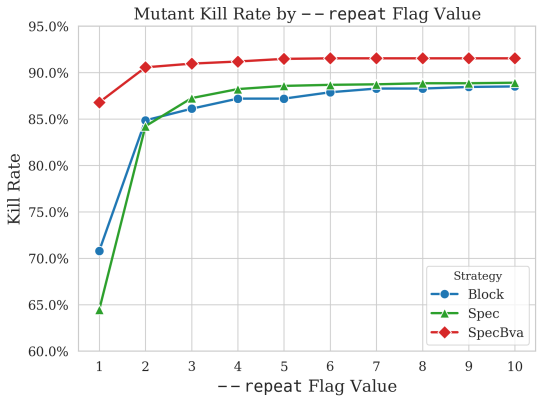


Figure 5.2: Mutant kill rate evolution excluding the *Path* strategy (1,751 mutants, 179 programs). It corroborates the main dataset, with *SpecBva* maintaining a clear advantage over both *Block* and *Spec* modes.

We also examined the relationship between the kill rate and the volume of generated tests. Figure 5.3 plots the kill rate against the average number of tests per mutant. While *SpecBva* ultimately produces a higher average number of tests per mutant by the tenth repetition, its efficiency is unmatched: by the sixth repetition, with an average of 12 tests per mutant, it achieves a kill rate that the other strategies can never, even with an average of 15 tests. This highlights the quality and focus of the tests generated via BVA.

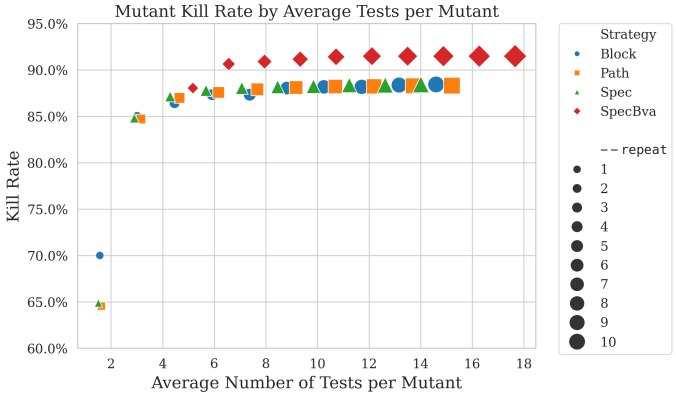


Figure 5.3: Kill rate evolution across the average number of tests generated per mutant. *SpecBva* demonstrates superior efficiency, reaching a kill rate by the sixth repetition (averaging 12 tests per mutant) that other strategies fail to achieve even with higher test volumes.

Additionally, we evaluated the complementarity of these approaches. As evidenced in Table 5.3, for this dataset of highly restrictive specifications, the optimal configuration is the combination of the *Path* and *SpecBva* modes. Impressively, this hybrid approach achieves its peak performance of 91.95% as early as repetition four. This outcome highlights a profound synergy:

while black-box testing (*SpecBva*) is highly efficient at covering expected contractual edge cases, white-box testing (*Path*) is necessary to expose structural faults hidden deep within the implementation logic. Ultimately, this suggests that developers can achieve near-optimal fault detection with significantly less computational overhead by combining BVA with a modest number of structural testing repetitions, avoiding the severe diminishing returns seen in later iterations.

Finally, we observed the execution times for each of the three major pipeline steps: test generation, safety check, and kill check.

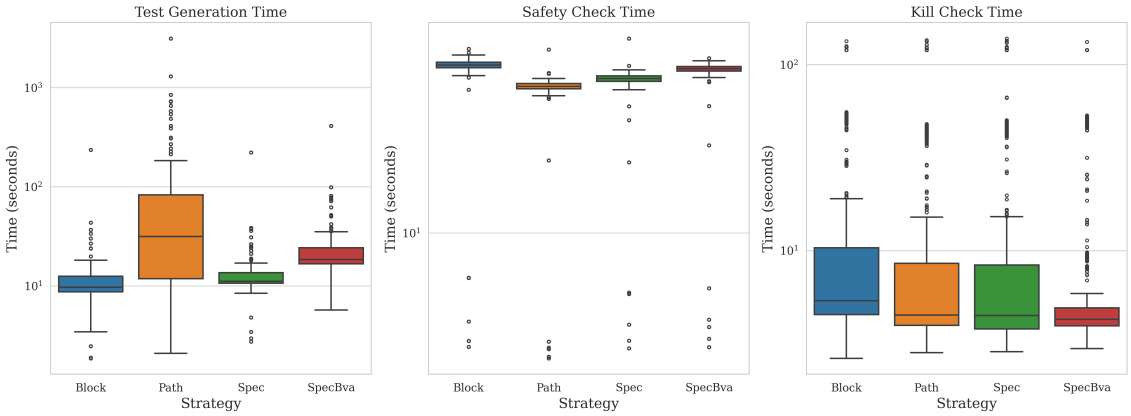


Figure 5.4: Log-scale execution times by strategy across the three pipeline phases: test generation, safety check, and kill check. As expected, *Path* experiences significantly higher test generation times. Outliers in the safety and kill checks reflect the short-circuiting evaluation logic, where failures in early repetitions cease time accumulation.

It is important to note a technical limitation regarding time measurement: while the `-time` flag provides the exact execution time for each repetition during test generation, the `dafny test` command does not.

To overcome this, we measured the safety and kill checks cumulatively. For example, the time recorded for repetition two represents the time to run tests from repetition one plus repetition two. To optimise this process, the kill check was conditionally applied: a mutant was only evaluated in the current repetition if it had not already been killed or timed out in a previous one. Similarly, the safety check would not be run for a certain program if it had already failed in a previous repetition.

Figure 5.4 clearly illustrates these limitations. The outliers in the safety check represent the few programs that crashed early during this step, and therefore did not calculate safety check time for the following repetitions. On the other hand, the outliers in the kill check time plots, represent the mutants that either survived, or were killed in later repetitions. For instance, since *SpecBva* achieved its highest kill rate early (around repetition six), its mean is lower when compared with the other modes.

Furthermore, the data demonstrates that test generation is by far the most time-consuming phase of the pipeline, with the *Path* mode exhibiting the longest test generation time of all, as expected. While the higher outliers correspond to highly complex programs that take longer to

generate tests for, the lower ones match programs for whom more tests were impossible to be generated, given the lack of valid inputs.

From this evaluation, we conclude that *SpecBva* is undoubtedly the superior strategy in mutation kill testing, while the significant time investment in the *Path* mode together with its average results, raises questions about its practical utility in automated debugging. To fully understand the value of these generated test suites regarding FL, we must look beyond isolated mutation coverage.

5.3.5 Concrete Example

To illustrate the practical implications of specification strength on test effectiveness, we examine a concrete example from our dataset. The tests for this analysis were generated using the *SpecBva* mode, not only because it consistently demonstrated the highest fault-detection capabilities throughout our evaluation, but also because it constitutes one of the core contribution of this dissertation.

Consider the program in Listing 5.3, which calculates the n-th triangular number using a while loop, constrained by a loose post-condition establishing a lower-bound property ($k+i+j \geq 2 \times n$).

Listing 5.3: `Dafny_Verify_tmp_tmphq7j0row_Fine_Tune_Examples_50_examples_41.dfy` program example.

```
1  method {:testEntry} main(n: int, k: int) returns (i: int, j: int)
2    requires n >= 0
3    requires k == 1 || k >= 0
4    ensures k + i + j >= 2 * n
5  {
6    i := 0;
7    j := 0;
8    while i < n
9      invariant 0 <= i <= n
10     invariant j == i * (i + 1) / 2
11   {
12     i := i + 1;
13     j := j + i;
14   }
15 }
```

When subjecting this program to mutation testing, a discrepancy emerges in the test suite's ability to detect different faults. For instance, the following mutation in line 6 of Listing 5.4 introduces an off-by-one error by altering the loop condition from $i < n$ to $i \leq n$.

Listing 5.4: `Dafny_Verify_tmp_tmphq7j0row_Fine_Tune_Examples_50_examples_41__185_ROR_Le.dfy` program mutant example.

```
1  method {:testEntry} main(n: int, k: int) returns (i: int, j: int)
2    // contract unchanged
3  {
4    i := 0;
5    j := 0;
6    while i <= n
7      invariant 0 <= i <= n
8      invariant j == i * (i + 1) / 2
```

```
9   {
10    i := i + 1;
11    j := j + i;
12  }
13 }
```

This mutation forces the while loop to execute for one additional iteration. Because the loop body strictly increases the values of the variables in each iteration (with i incrementing by one and j accumulating i), this extra pass produces terminal values for i and j that are strictly greater than those produced by the original program. Consequently, the final sum safely satisfies the loose inequality $k+i+j \geq 2 \times n$. Since the *SpecBva*-generated test suite is produced based on the decomposition of the loose contract, it only verifies this lower bound rather than the exact expected state of the variables. As a result, every produced test passes, and the mutation is not caught. In this particular case, two extra post-conditions would be needed to produce tests that actually caught the mutation: `ensures i == n` and `ensures j == n * (n + 1) / 2`.

Conversely, consider a second mutation from this dataset that alters the loop condition to $i == n$, again in line 6, this time in Listing 5.5.

Listing 5.5: `Dafny_Verify_tmp_tmphq7j0row_Fine_Tune_Examples_50_examples_41__185_ROR_Eq.dfy` program mutant example.

```
1  method {:testEntry} main(n: int, k: int) returns (i: int, j: int)
2    // contract unchanged
3  {
4    i := 0;
5    j := 0;
6    while i == n
7      invariant 0 <= i <= n
8      invariant j == i * (i + 1) / 2
9    {
10     i := i + 1;
11     j := j + i;
12   }
13 }
```

For any boundary test case where $n > 0$, the loop condition immediately evaluates to false upon initialisation, bypassing the loop body entirely and leaving i and j at zero. In this example, when the tests evaluate the post-condition, the resulting sum drastically fails to meet the required $2 \times n$ threshold for larger values of n. Because this mutation strictly violates the property codified in the test assertions, the same test suite that failed to spot any mutation in the previous example, now successfully detects the faulty state and kills the mutation. Ultimately, this demonstrates that an apparent failure to detect a bug might happen not because the tests produced or the methods to produce them are weak, but because the specification is not strong enough to ensure the correct behaviour of the program.

5.4 Spectrum-Based Fault Localisation Evaluation

Having established that DSpec2Test's specification-driven testing, particularly *SpecBva*, efficiently generates high-quality tests capable of killing mutants, the next step is to evaluate whether these test suites can accurately localise those faults. To achieve this, we applied SBFL techniques to the all four test suites generated in the previous phase.

This evaluation utilised the 1,541 mutants derived from the 158 programs that successfully passed the test generation and safety check phases in all four strategies. To compute the execution spectra required by SBFL, we used the custom per-test code coverage plugin described in Section 4.2, with the `max_iter` parameter set to 5,000. The resulting JSON files, which mapped each test to its pass/fail status and the specific lines it executed, served as the foundational data for our FL algorithms.

During this process, there was an additional precaution that we had to acknowledge. Although the plugin is equipped to handle infinite loops or recursive calls, calling it with the `-no-verify` flag makes it vulnerable to compilation errors of the defective mutants. Therefore, we adjusted our pipeline to include a mechanism that allows to handle these issues, without compromising the performance of the whole. Usually, `dafny test` is called once per file. However, when a crash takes place, each test of the program is individually cared for, allowing for them to crash, but still execute the following tests and programs. Evidently, if a test leads to a crash we classify it as failing.

Moreover, the ground truth was extracted from the mutant generation phase described in Section 5.2. Because standard code coverage tools trace execution at the level of abstract syntax tree nodes or basic blocks, they inherently ignore non-executable structural lines, such as standalone braces ({ or }) or comments. To prevent valid programs from being incorrectly penalised or dropped during evaluation, we mapped the ground truth annotations against the original formatted source code. If an annotated fault pointed to a non-executable line, we applied a dynamic mapping heuristic to search the immediate adjacent code block and include the nearest executable statement. This adjustment ensures that physical line numbers accurately reflect the logical execution of the faulty basic block, an alignment necessary for the valid empirical assessment of coverage-based fault locators [70].

However, before evaluating the FL capabilities of each strategy, it is crucial to filter out mutants that do not trigger any test failures, as these cannot be localised by SBFL techniques. From the mutation kill rates achieved in the previous section, we could derive the amount of mutants where this happens. Nevertheless, since previously a timeout occurred through a time limit, and now we have a parameter that limits recursion, there might be some disalignment. Therefore, Table 5.4 reproduces these values. *Spec*, *Block*, and *Path* failed to trigger test failures in roughly 10.5% of the generated mutants. In contrast, *SpecBva* left only 7.6% of mutants undetected, therefore, the SBFL evaluation over *SpecBva* will address a higher range of mutants than the remaining strategies. We can see that there is indeed a mismatch of less than 1% between the data shown in Table 5.3 and Table 5.4 .

Table 5.4: Zero-Failure Mutants per Test Generation Strategy (Total = 1,541)

Strategy	Zero-Failure Mutants	Percentage (%)	Valid Mutants
Block	165	10.7	1376
Path	163	10.6	1378
Spec	162	10.5	1379
SpecBva	117	7.6	1424

To process the execution spectra generated by the custom coverage plugin, we evaluated the test suites using three SBFL similarity coefficients: Tarantula, Ochiai, and DStar. For D^*, we utilised an exponent of 2 (i.e., D^2), as it is the standard in literature. As detailed in Section 2.3.1, these specific metrics were selected because they represent fundamentally different mathematical approaches to fault isolation. To evaluate their effectiveness across the different test generation modes, we employed the standard ranking metrics previously established in Section 2.3: *Top-N* Rank, *Mean Reciprocal Rank* (MRR), *Average Rank*, and the *EXAM* Score.

In scenarios where multiple statements shared the exact same suspiciousness score (a common occurrence in SBFL, known as a *tie*), we employed a tie-breaking strategy. Rather than optimistically assuming the developer inspects the faulty statement first, or pessimistically assuming they inspect it last, the most recently adopted approach calculates the arithmetic mean of the ranks within the tied block [50]. This provides the most mathematically sound and realistic representation of a developer's average debugging effort when faced with equally suspicious statements.

5.4.1 Results and Discussion

After applying all three coefficients to the tests generated by the different modes, we soon concluded that Ochiai outperforms the alternatives. Specifically, *SpecBva* achieves the best performance across every single metric, even when accounting for a larger pool of valid mutants than the other strategies. Figure 5.5 illustrates this comparison for the *Top-1* accuracy, while Figure 5.6 shows the *EXAM* score performance.

To understand why Ochiai consistently outperformed Tarantula and D^* , we must examine their underlying mathematical formulations. The effectiveness of an SBFL metric depends on how it balances the penalty for a statement being executed in passing tests (N_{CS}) against the reward for being executed in failing tests (N_{CF}). Tarantula normalises the failure and passing ratios against the total number of failing (N_F) and passing (N_S) tests. However, in automated test generation scenarios like *SpecBva*, the suite often produces a vast number of passing tests compared to failing ones. This massive imbalance dilutes Tarantula's passing ratio, making it overly sensitive to coincidental correctness, a phenomenon where a test case executes a faulty statement but still produces the correct output because the error fails to propagate to an observable failure [39].

Unlike Tarantula, Ochiai completely ignores N_{FUS} (statements not executed in passing tests), making it highly resilient to the vast volume of a test suite. Furthermore, the geometric mean in the denominator heavily penalises statements that are frequently executed in passing tests (N_{CS}) without relying on the global passing test count (N_S). Because *SpecBva* generates dense boundary

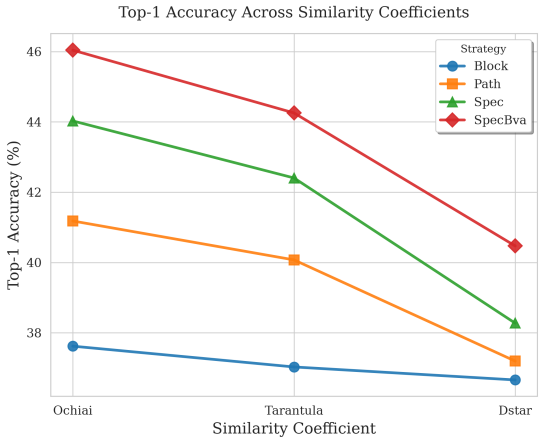


Figure 5.5: Top-1 score for each strategy across the three coefficients. All strategies performed best when using Ochiai. *SpecBva* showcases the best results across all metrics.

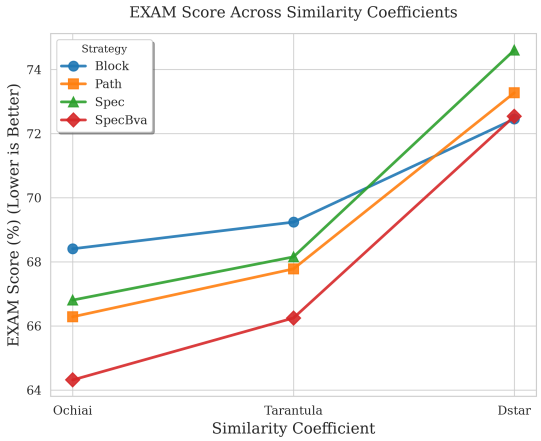


Figure 5.6: Exam score for each strategy across the three coefficients (lower is better). All strategies performed best when using Ochiai. *SpecBva* showcases the best results across all metrics.

tests that often execute the same paths with slightly different values (some passing, some failing), Ochiai successfully filters out the spectral noise, pinpointing the actual fault with greater precision than D^*'s purely polynomial approach.

Given the superior results achieved by Ochiai, Table 5.5 summarises the FL performance of this metric across the distinct test generation modes, as well as their combinations.

Table 5.5: Spectrum-Based Fault Localisation results using Ochiai across individual and combined test generation strategies. *SpecBva* dominates standalone performance, although combining it with the white-box strategies yield slight improvements in every metric.

Strategy	**Top-1 (%)**	**Top-3 (%)**	**Top-5 (%)**	**MRR**	**Avg Rank**	**EXAM (%)**
Block	37.6	91.1	97.9	0.652	1.96	68.41
Path	41.2	91.4	98.0	0.677	1.89	66.29
Spec	44.0	90.9	97.1	0.690	1.94	66.81
SpecBva	46.0	91.4	97.2	0.707	1.84	64.32
Combined Strategies						
Spec + Block	44.5	91.6	97.6	0.698	1.85	65.25
Spec + Path	45.2	91.5	97.6	0.702	1.84	65.16
SpecBva + Block	46.4	91.8	97.7	0.712	1.80	63.72
SpecBva + Path	46.5	91.6	97.7	0.712	1.80	63.77

An analysis of Table 5.5 reveals several critical insights regarding the relationship between test generation techniques and FL efficacy. First, *SpecBva* unequivocally dominates the standalone strategies, achieving the highest *Top-1* accuracy (46%) and *MRR* (0.707), while requiring developers to inspect the smallest portion of the codebase (*EXAM* score of 64.32%). Its superiority comes from the precision of boundary tests: by constraining test generation to the boundaries of input domains, *SpecBva* produces failing execution spectra that are highly localised, which drastically

reduces the occurrence of coincidental correctness.

Mathematically, examining the relationship between the evaluation metrics reveals the precise behaviour of *SpecBva*. *Top-1* accuracy represents a binary success rate, whereas *MRR* acts as a harmonic mean that disproportionately rewards high ranks (a rank of 1 yields 1.0, a rank of 2 yields 0.5, a rank of 3 yields 0.33, etc.). Given *SpecBva*'s *Top-1* of 46%, those top-ranked faults contribute 0.46 to the total *MRR*. Subtracting this from the total *MRR* of 0.707 leaves a contribution of 0.247 spread across the remaining 54% of the faults. This results in an average reciprocal rank of approximately 0.457 (0.247 divided by 54%) for the faults that miss the top spot. Because a rank of 2 yields exactly 0.5, this mathematical relationship dictates that when *SpecBva* fails to rank the fault first, it almost certainly ranks it second, which is corroborated by the 1.84 *Average Rank* value.

However, the data exposes a clear performance ceiling, as not even the combined strategies reach a *Top-1* score of 50%. This exposes a limitation of spectrum-based techniques: because SBFL is purely a statistical approximation of causality, it cannot distinguish between a genuinely faulty statement and a non-faulty statement that shares the exact same execution trace. In fact, the data shows that, on average, the faulty line is ranked second, across all strategies. This suggests that semantic-aware techniques are required to achieve true exact-match isolation.

5.4.2 Concrete Example

To illustrate the capabilities and inherent limitations of SBFL, we examine two distinct mutants of the triangular number program introduced in Section 5.3.5. Both mutants were evaluated using the *SpecBva*-generated test suite. By analysing their execution spectra, we can understand why SBFL metrics successfully isolate certain faults while struggling with others.

Consider the first mutant, which introduces an unexpected `break` statement inside the `while` loop at line 10, in Listing 5.6.

Listing 5.6: `Dafny_Verify_tmp_tmphq7j0row_Fine_Tune_Examples_50_examples__41__177-309_LBI.dfy` program mutant example.

```
1  method {:testEntry} main(n: int, k: int) returns (i: int, j: int)
2    // contract unchanged
3  {
4    i := 0;
5    j := 0;
6    while i < n
7      invariant 0 <= i <= n
8      invariant j == i * (i + 1) / 2
9    {
10     break;
11     i := i + 1;
12     j := j + i;
13   }
14 }
```

For this mutant, all three evaluated metrics (Tarantula, Ochiai, and D^2) perfectly ranked the buggy line as the number one most suspicious statement. To understand why, we must look at the execution spectra produced by the test suite.

Boundary tests where $n = 0$ evaluate the loop condition to false and bypass the loop body entirely, successfully passing while only covering the variable initialisations and condition clause (lines 4, 5, and 6). Conversely, tests where $n > 0$ enter the loop, immediately execute the `break` statement on line 10, and subsequently fail because the postcondition is violated. Crucially, the `break` instruction prevents the execution of the subsequent statements within the loop.

Mathematically, line 10 is executed in every single failing test ($N_{CF} > 0$) and in absolutely zero passing tests ($N_{CS} = 0$). Because no other statement in the program shares this exact, purely fatal execution signature, all three SBFL formulas yield their maximum possible suspiciousness score exclusively for line 10. This allows the metrics to unambiguously isolate the fault, resulting in a perfect *Top-1* rank.

However, SBFL techniques are strictly bound by the granularity of the coverage data they receive. Consider a second mutant, where the increment operation is altered from an addition to a multiplication, also at line 10, in Listing 5.7.

Listing 5.7: `Dafny_Verify_tmp_tmphq7j0row_Fine_Tune_Examples_50_examples_41__280_AOR_Mul.dfy` program mutant example.

```
1  method {:testEntry} main(n: int, k: int) returns (i: int, j: int)
2    // contract unchanged
3  {
4    i := 0;
5    j := 0;
6    while i < n
7      invariant 0 <= i <= n
8      invariant j == i * (i + 1) / 2
9    {
10     i := i * 1;
11     j := j + i;
12   }
13 }
```

Unlike the previous example, this fault was not perfectly isolated. Ochiai and Tarantula ranked the correct line in second place, while D^2 ranked it third. This degradation in ranking performance is a classic manifestation of the "basic block" limitation in coverage-based FL.

When analysing the coverage traces for this mutant, tests with $n = 0$ pass and cover lines 4, 5, and 6, exactly as before. Tests with $n > 0$ enter the loop and execute the mutated line 10. Because the assignment `i := i * 1` does not interrupt control flow, execution seamlessly proceeds to line 10. The program then either traps itself in an infinite loop (which the plugin handles smoothly via a parametrised limited amount of loop iterations) or fails the postcondition validation.

As a result, lines 10 and 11 share the exact same execution spectrum: both are executed in all failing tests and zero passing tests. From the perspective of the SBFL formulas, these two statements are mathematically indistinguishable. They receive the exact same maximum suspiciousness score, creating a tie. Because SBFL metrics lack syntactic awareness and cannot deduce

that `i := i * 1` is inherently more suspicious than `j := j + i`, the tie must be resolved arithmetically. This example demonstrates that while SBFL is highly effective at narrowing down the location of a fault to a specific execution path or basic block, it is fundamentally incapable of differentiating between sequentially dependent statements that are always executed together.

5.4.3 Impact of Failing Tests Rate on Fault Localisation

Given that the SBFL coefficients were evaluated on a robust number of tests per strategy (averaging 18 for *SpecBva*, 14 for *Spec*, and 15 for the white-box approaches), it is necessary to investigate whether the ratio of passing to failing tests within a suite impacts localisation accuracy.

To determine this, we analysed the correlation between the failing test ratio of each mutant and its resulting localisation performance with Ochiai. Since our data distribution is non-parametric we employed Spearman's Rank Correlation Coefficient (ρ) [60] for the continuous variables (*MRR*, *EXAM*). Because *Top-1* accuracy is a binary outcome (Success or Failure), we evaluated its relationship with the failing test ratio using the Mann-Whitney U test [38, 68] to determine statistical significance. However, given our large sample size (N>1,350), even trivial differences can produce highly significant p-values. Therefore, we also calculated the Rank-Biserial Correlation [16] to quantify the actual magnitude (effect size) of the relationship.

Table 5.6 presents the results of our statistical analysis that revealed two distinct behaviours regarding how test suite composition impacts FL. Note that for the correlation analysis, 23 additional mutants were excluded across all strategies because their unexecuted faults yielded no definable rank in the SBFL output. This happened on some Statement Deletion (SDL) mutations from MutDafny that kept no executable statements.

Table 5.6: Statistical correlation between the failing test ratio and fault localisation metrics. Values represent the Rank-Biserial effect size for *Top-1* accuracy (derived via Mann-Whitney U test) and Spearman's ρ for *MRR* and *EXAM* scores. The results indicate that while the failing test ratio has negligible impact on top-tier ranking metrics (*Top-1* and *MRR*), a higher ratio significantly degrades the overall *EXAM* score.

Strategy	*N*	**Top-1 Effect**	**MRR** (ρ)	**EXAM** (ρ)
Block	1353	-0.107***	0.055*	0.363***
Path	1355	-0.112***	0.051	0.413***
Spec	1356	-0.036	0.007	0.379***
SpecBva	1401	-0.081**	0.049	0.402***

*$p < 0.05$, **$p < 0.01$, ***$p < 0.001$

Firstly, the data indicates that the failing test ratio has no practical impact on the ability to rank the fault at or near the very top. Across all strategies, the correlation with *MRR* was exceptionally weak ($\rho \approx 0.05$). While the Mann-Whitney U test for *Top-1* accuracy yielded statistically significant p-values ($p < 0.01$) due to our large sample size (N>1,350), the actual effect sizes were

negligible (ranging from -0.036 to -0.112). This demonstrates that if an SBFL formula is capable of accurately pinpointing a bug to the first rank, it will generally do so regardless of whether the test suite contains a low or high density of failing tests.

Conversely, we observed a statistically significant, moderate positive correlation ($\rho \approx 0.36$ to 0.41, $p < 0.001$) between the failing test ratio and the *EXAM* score across all strategies. Because a higher *EXAM* score represents worse performance (requiring the developer to inspect a larger percentage of the codebase), this indicates that test suites heavily skewed toward failing tests degrade the overall ranking spectrum. In traditional SBFL, diagnostic formulas rely heavily on the contrast between execution traces of passing and failing tests. When a high volume of tests fail, this execution contrast is diluted by collateral execution paths, making it harder for the formula to separate the root cause from the noise. Consequently, if the tool fails to rank the bug at the very top, a high failing test ratio ensures the true fault is buried significantly further down the ranked list.

In summary, while the ratio of failing tests has no meaningful correlation with *MRR* and *Top-1* results, a higher concentration of failing tests degrades the overall *EXAM* score.

5.4.4 Impact of Program Length

To investigate whether our findings are skewed by trivial implementations or not, we conducted a secondary evaluation isolating the top 25% longest programs in our dataset (third quartile). As program size increases, the search space for faults naturally expands, introducing more execution paths and, consequently, more spectral noise into the SBFL computation. Figure 5.7 shows that the median implementation size of the programs in our mutated dataset is six. The third quartile indicates that only 25% of the mutants actually have an implementation with eight or more lines of code (discarding trivial, non-executable lines).

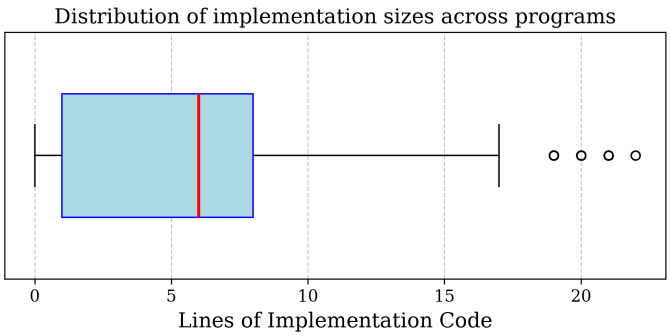


Figure 5.7: Distribution of implementation sizes across mutated programs. The median implementation size is six, while only 25% of the programs have eight or more lines of code.

Therefore, this section will target the evaluation of the 498 programs within the third quartile, using the same metrics as above. Just as before, programs with no failing tests will be discarded from this evaluation, since SBFL cannot detect faulty lines in these scenarios. Thus, Table 5.7 shows the actual amount of mutants that will be evaluated for each strategy.

Table 5.7: Zero-Failure mutants per test generation strategy on programs with 8 or more lines of code (Total = 498).

Strategy	Zero-Failure Mutants	Percentage (%)	Valid Mutants
Block	67	13.5	431
Path	62	12.4	436
Spec	71	14.3	427
SpecBva	60	12.0	438

As expected, FL accuracy experienced degradation across all strategies when restricted to these longer programs. However, Ochiai remained the metric that yielded the best results. Therefore, Table 5.8 presents the results obtained for this coefficient.

Table 5.8: Spectrum-Based Fault Localisation results using Ochiai across individual and combined test generation strategies for mutated programs with eight or more lines of code. *SpecBva* continues to dominate standalone performance, while combined strategies yield slight improvements, particularly, in *Top-1* accuracy.

Strategy	Top-1 (%)	Top-3 (%)	Top-5 (%)	MRR	Avg Rank	EXAM (%)
Block	5.8	75.1	93.5	0.429	2.82	46.90
Path	10.1	75.3	93.8	0.463	2.70	44.00
Spec	11.8	72.7	90.8	0.459	2.99	45.32
SpecBva	15.6	75.0	91.1	0.501	2.71	40.65
Combined Strategies						
Spec + Block	13.9	75.4	92.5	0.485	2.69	43.11
Spec + Path	15.2	75.2	92.5	0.493	2.67	42.91
SpecBva + Block	16.6	76.2	92.7	0.511	2.59	40.17
SpecBva + Path	17.2	75.8	92.8	0.513	2.59	40.25

We can see that the comparative hierarchy remained identical: *SpecBva* maintained its position as the most effective standalone strategy, while its combination with either *Path* or *Block* continued to produce the absolute best results. Therefore, although the performance decreases across every metric in this restricted dataset, the same relative conclusions can be drawn: the complementarity between black-box and white-box testing techniques produces unmatched test suites for FL.

However, while the relative hierarchy of the test generation strategies remains intact, the absolute metrics demonstrate severe limitations regarding SBFL's viability in practice. Isolating the longest programs (which, in this dataset, merely means implementations with 8 or more lines of code) causes the peak *Top-1* accuracy to drop sharply from 46.5% to 17.2%. If a search space expansion of just a few lines of code is sufficient to introduce enough spectral noise to mask the exact fault over 80% of the time, it indicates that the statistical nature of SBFL lacks robustness. This sharp degradation reinforces our earlier conclusion: relying solely on execution frequencies is fundamentally inadequate for precise fault isolation. If SBFL struggles to pinpoint faults in implementations of this modest size, its applicability to real-world, complex codebases is highly questionable, reaffirming the need for deeper, semantic-aware FL techniques.

5.5 LLM-based Fault Localisation Evaluation

LLMs are increasingly being adopted to aid in software engineering tasks, demonstrating strong capabilities in code generation, analysis, and debugging [75]. Given these advancements, we conducted a targeted experiment to compare LLM-driven FL against the traditional SBFL approach.

To optimise computational, time, and cost resources while maintaining a rigorous comparison, we restricted our focus to the best-performing test-generation strategies from our earlier evaluation: *SpecBva* for black-box testing and *Path* for white-box testing. Furthermore, because previous results indicated that SBFL struggles significantly with larger codebases, we filtered the dataset to include only the 498 mutants containing eight or more lines of code, roughly equivalent to 25% of the original dataset. This specifically tests the LLM against the exact scenario where SBFL experienced a severe degradation in performance. The experiment was executed using the `gpt-5-mini` model [49], selected for its highly efficient balance of reasoning capability and cost-effectiveness.

Inspired by the findings outlined in Section 2.3.2, we designed a concise and straightforward prompt for this evaluation. To ensure a fair comparison, the LLM was provided with the same information available to the SBFL algorithm: the complete program and the suite of tests generated by the corresponding strategy. To avoid further increasing the already considerable length of the prompt, the per-test code coverage was not included in it. We explicitly removed all natural language comments from the source code, as SBFL does not acknowledge them, but leaving them could affect the LLM's performance. Particularly, the first line comments containing the name of the program and the applied mutation, could give clues to the LLM, especially due to the potential of data leakage. Listing 5.8 shows the overview structure of our prompt.

Listing 5.8: Structure of the LLM prompt for fault localisation.

````
You are an expert formal verification engineer specializing in Dafny.
Your task is to perform precise fault localization. You will be provided with a specific
Dafny program, and, occasionally, its test cases. Do not hallucinate external context. Your
output MUST be a valid JSON object with exactly two keys: 'lines' (a list of integers
representing the buggy line numbers) and 'reason' (a single-sentence string explaining the
root cause). Do not output any other text or explanation.

[TASK INSTRUCTION]
Analyze the code and the following tests. Locate the bug.

[DAFNY PROGRAM]
{method_code}

[TESTS]
{all_tests}

[OUTPUT FORMAT]
Provide ONLY a valid JSON object in the following format:
```json
{{
   "lines": [<int>, ...],
   "reason": "<one sentence reasoning>"
}}
```
````

Our prompt instructs the model to provide a reasoning trace before outputting the final localised fault. While our evaluation focuses on the accuracy of the localised lines, rather than the quality of the explanation, enforcing this intermediate step is a deliberate design choice. Research [65] demonstrates that prompting models to output a reasoning chain, forces them to sequentially process complex logic, drastically reducing systematic failures and hallucination. By requiring an explanation, we hope to improve the precision of the final localisation while simultaneously generating a trace that aids in debugging.

5.5.1 Results and Discussion

Before assessing LLM-based FL performance, it is important to highlight the operational efficiency of the chosen model. As detailed in Table 5.9, the entire experiment was highly resource-efficient when processing the 498 files for each testing strategy. With the `gpt-5-mini` model, the total execution time was approximately 4.8 hours, incurring a negligible total API cost of $2.36 USD. This confirms that LLM-driven FL can be deployed at scale, offering a great balance of reasoning capability and cost-effectiveness without prohibitive overhead.

Table 5.9: Operational cost and execution time for the `gpt-5-mini` evaluation across 498 programs in each strategy.

Strategy	Input Tokens	Output Tokens	Time (s)	Cost (USD)
Path	734,808	33,179	8,706	$1.20
SpecBva	801,614	32,988	8,848	$1.16
Total	**1,536,422**	**66,167**	**17,554**	**$2.36**

The evaluation results clearly demonstrate the efficacy of LLMs for FL. As detailed in Table 5.10, both *Path* and *SpecBva* excelled at FL across all metrics, representing a massive improvement compared to SBFL (Table 5.8). While the metrics are not perfectly synonymous, a connection can still be established: because LLMs typically output unranked sets of lines rather than an exhaustive ranked list, the LLM's *Any Match* metric (indicating that at least one flagged line is faulty) serves as the closest analogue to a successful top-tier SBFL ranking (such as *Top-1* or *Top-5*). Both metrics measure the tool's ability to successfully guide the developer to genuine faulty code in a single interaction. This *Any Match* metric was explicitly included to maintain evaluation parity, as a successful localisation in standard practice is ultimately defined by whether the developer is pointed to the ground-truth fault. Here, we see a striking discrepancy, increasing from 10–15% in SBFL to over 90% with the LLM. The overall *Precision* (89%) and *Recall* (87%) further indicate that when the model identifies a fault, it rarely flags innocent code and successfully captures the vast majority of the faulty logic.

Furthermore, this parity is strengthened by the qualitative differences in the tools' outputs. While an SBFL ranking provides only a numerical suspicion score, LLMs accompany their flagged lines with a natural language justification. Manual observation of the results revealed that even when the LLM flagged slightly larger scopes (e.g., an entire conditional block rather than a single faulty expression), the accompanying "reason" immediately clarified the exact nature of the

bug. This contextualisation has the potential to significantly reduce the developer's cognitive load, making the *Any Match* of a multi-line prediction functionally equivalent, and perhaps superior, to inspecting a single isolated line flagged by traditional SBFL without context.

Table 5.10: LLM-based FL results. Both strategies exhibit great performance across all datasets, much superior than the one seen in SBFL.

	All programs		**Programs with failing tests**		**Programs with no failing tests**	
Strategy	*Path*	*SpecBva*	*Path*	*SpecBva*	*Path*	*SpecBva*
Number of Programs	498	498	436	438	62	60
Precision (%)	88.75	89.09	89.41	89.08	84.14	89.17
Recall (%)	86.95	87.85	87.84	88.47	80.65	83.33
F1 Score (%)	85.98	86.72	86.67	86.96	81.11	85.00
Exact Match (%)	73.09	74.50	72.94	74.43	74.19	75.00
Any Match (%)	92.97	93.37	94.04	93.84	85.48	90.00

When comparing the test-generation strategies, the results reveal a nuanced relationship. When evaluating strictly on programs with failing tests, the white-box *Path* strategy slightly outperformed the black-box *SpecBva* in Precision (89.41% vs. 89.08%) and *Any Match* (94.04% vs. 93.84%). However, *SpecBva* maintained higher *Recall* and *F1-Scores* across all categories, proving it provides a more comprehensive behavioral context for the model to interpret.

It is also important to note that the 75% *Exact Match* score may actually underestimate the model's true precision. As previously detailed (Section 5.4), the ground-truth fault locations were heuristically mapped to the nearest executable statements to accommodate the AST-based tracking of the code coverage plugin. Consequently, if the LLM semantically identified a relevant line that is not the nearest to where the code was omitted, it would be strictly penalised under the *Exact Match* metric, despite providing a logical reasoning.

This behavioral context becomes critical when examining programs with no failing tests. Since these programs were discarded from the SBFL evaluation (as SBFL relies entirely on the mathematical divergence between passing and failing execution traces) we analysed them separately. Remarkably, the LLM maintained high performance even on this strictly passing subset, achieving an *Any Match* score of 85.48% with *Path*, and 90% with *SpecBva*. Because the LLM relies on deep semantic code comprehension rather than execution paths, it can detect logical inconsistencies and localise bugs even without a failing test acting as a trigger.

5.6 Final Discussion

From this chapter, we can conclude that different approaches to similar problems can produce very distinct outcomes.

Our initial test generation evaluation demonstrated that mutation testing yields strong results, frequently achieving kill rates approaching or exceeding 90%. Within this context, *SpecBva* outperformed its counterparts. Despite producing a higher volume of tests, it generated superior

results within a comparable timeframe. Conversely, the white-box *Path* strategy was computationally expensive, taking the longest to conclude test generation, and was solely responsible for the failure to process 21 programs within the one-hour timeout limit.

Moving towards localising faults, SBFL produced much unsatisfactory outcomes. After discarding the mutants with no failing tests, we were left with 1,424 mutants to evaluate on *SpecBva*, and around 1,380 mutants for the remaining strategies. None of them (even when combined) could reach a 50% *Top-1* rate, and all required a developer to analyse at least 64% of the code to find the bug (*EXAM* score). However, when filtering the dataset to contain only mutants with eight or more lines in the implementation, the results were even more unsettling. The *Top-1* score failed to reach even 20%. While the EXAM score improved to around 40%, this might be an artificial inflation due to the proportional increase in average program size rather than an improvement in the algorithm's precision.

When comparing these results to semantic-aware LLM-based FL, we conclude that LLMs are vastly superior at identifying and extracting faulty logic. The results showed a stark contrast: while SBFL struggled to immediately highlight the bug, the LLM achieved an *Any Match* rate exceeding 90% and an *Exact Match* rate of approximately 75% (that might be negatively biased due to the heuristic adjustment of ground-truth lines). Furthermore, unlike SBFL, which relies entirely on the mathematical divergence between passing and failing execution traces, the LLM's deep semantic comprehension allowed it to successfully localise faults even in programs with strictly passing test suites. It accomplished this with remarkable operational efficiency, proving that generative models can bridge the gap where traditional spectrum-based methods fail to produce good-quality results.

In conclusion, this research highlights that while robust test generation strategies like *SpecBva* provide an excellent foundation for identifying the presence of bugs, relying on SBFL to locate them is practically inefficient for developers. By using generative LLMs, we can drastically reduce the manual search space, maintain high precision in larger codebases, and successfully decouple the localisation process from the strict necessity of failing tests. In summary, integrating semantic-aware LLMs into the debugging pipeline represents a significant step toward truly automated FL.

5.7 Answer to RQ 3: How effective are these deterministic and non-deterministic approaches at detecting bugs in Dafny programs, and how do they compare to one another?

RQ 3: Deterministic SBFL proves to be moderately effective at isolating bugs in Dafny, peaking at a 46.5% *Top-1* accuracy when combining specification-generated test suites (*SpecBva*) with white-box strategies. However, its reliance on statistical execution spectra introduces severe limitations: it cannot mathematically distinguish between sequentially dependent statements within the same execution block, and its performance degrades drastically on larger codebases, descending to a 17.2% *Top-1* rate for implementations of eight or more lines.

In contrast, non-deterministic LLM-based FL significantly outperforms traditional SBFL by understanding code semantics rather than relying purely on execution traces. When evaluated on the exact subset of larger programs where SBFL's performance deteriorated, the LLM achieved an *Exact Match* of approximately 75% and an *Any Match* exceeding 90%. Furthermore, this semantic awareness allowed the LLM to successfully localise faults even in programs without failing tests — a scenario where SBFL is entirely blind. Ultimately, while deterministic execution traces provide a baseline, non-deterministic generative models offer a vastly superior approach to FL.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Guaranteeing software reliability is a critical challenge in modern computing, and verification-aware languages like Dafny offer robust mathematical proofs of correctness to address this need. However, debugging verification failures remains a major hurdle that limits the widespread adoption of these languages. This dissertation bridges the gap between formal verification and practical testing by introducing DSpec2Test, a novel specification-driven test generation tool that uses DNF as ECP and BVA to generate tests directly from formal contracts. Furthermore, to enable execution tracing in Dafny, we developed a custom per-test code coverage plugin capable of handling complex control flows, including infinite loops and recursive calls.

Our empirical evaluation via mutation testing on programs from the DafnyBench dataset revealed that specification-based testing with BVA (*SpecBva*) drastically outperforms traditional white-box testing, achieving an impressive 91.5% mutation kill rate. By applying SBFL metrics to these generated test-suites, we demonstrated that *SpecBva*, combined with the Ochiai coefficient, provides the best standalone FL (46% *Top-1* accuracy). The absolute best results were achieved by combining black-box and white-box test suites, highlighting a strong complementarity between testing structural logic and contractual boundaries.

Additionally, an LLM-based FL approach (with `gpt-mini-5`) was also employed against a smaller set of longer programs, where SBFL failed to meet the 20% *Top-1* mark. The generative model drastically outperformed the latter, achieving an *Exact Match* of approximately 75% and an *Any Match* exceeding 90%. Furthermore, by relying on deep code comprehension rather than execution traces, the LLM successfully localised bugs even in programs with strictly passing test suites.

In summary, these findings indicate that while specification-driven test generation produces high-quality test suites, relying purely on SBFL can be insufficient for precise bug localisation. Instead, the results suggest that integrating generative LLMs offers a promising approach to enhance the debugging of verification-aware languages. Together with the two novel tools created

to facilitate FL, these contributions have the potential to support a wider adoption of verification-aware languages.

6.2 Future Work

Future research should first address the current technical limitations of DSpec2Test. Specifically, integrating function inlining in specifications, similarly to the existent *InlinedBlock* mode of Dafny's `generate-tests` command, and adding support for mutable state elements, such as arrays and class fields across all test-generation strategies. These improvements will significantly broaden the tool's real-world applicability.

Furthermore, given the strict limitations observed with purely statistical SBFL metrics, future work must explore semantic-aware FL techniques even further. The current experiments could be complemented with scenarios where no tests are given to the LLM, or when only one failing test case is provided.

Finally, integrating methodologies like Angelic Debugging could elevate the current statement-level ranking to an expression-level ranking, offering developers highly precise debugging suggestions and ultimately paving the way for full Automated Program Repair in verification-aware languages, particularly, Dafny.

References

[1] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. "An evaluation of similarity coefficients for software fault localization". In: *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. IEEE. 2006, pp. 39–46.

[2] Wolfgang Ahrendt et al. *Deductive software verification-the key book*. Springer, 2016.

[3] Isabel Amaral, Alexandra Mendes, and José Campos. "MutDafny: A Mutation-Based Approach to Assess Dafny Specifications". In: *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE)*. 2026.

[4] Mike Barnett, K Rustan M Leino, and Wolfram Schulte. "The Spec# programming system: An overview". In: *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*. Springer. 2004, pp. 49–69.

[5] Antonia Bertolino. "Software testing research: Achievements, challenges, dreams". In: *Future of Software Engineering (FOSE'07)*. IEEE. 2007, pp. 85–103.

[6] Franck Cassez. "Verification of the incremental Merkle tree algorithm with Dafny". In: *International Symposium on Formal Methods*. Springer. 2021, pp. 445–462.

[7] Franck Cassez, Joanne Fuller, and Horacio Mijail Antón Quiles. "Deductive verification of smart contracts with Dafny". In: *International Conference on Formal Methods for Industrial Critical Systems*. Springer. 2022, pp. 50–66.

[8] Franck Cassez et al. "Formal and executable semantics of the ethereum virtual machine in dafny". In: *International Symposium on Formal Methods*. Springer. 2023, pp. 571–583.

[9] Patrice Chalin. "A sound assertion semantics for the dependable systems evolution verifying compiler". In: *29th International Conference on Software Engineering (ICSE'07)*. IEEE. 2007, pp. 23–33.

[10] Satish Chandra et al. "Angelic debugging". In: *Proceedings of the 33rd International Conference on Software Engineering*. 2011, pp. 121–130.

[11] Yoonsik Cheon and Gary T Leavens. "A simple and practical approach to unit testing: The JML and JUnit way". In: *European Conference on Object-Oriented Programming*. Springer. 2002, pp. 231–255.

[12] Maria Christakis et al. "Integrated environment for diagnosing verification errors". In: *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2016, pp. 424–441.

[13] Koen Claessen and John Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. 2000, pp. 268–279.

[14] Holger Cleve and Andreas Zeller. "Locating causes of program failures". In: *Proceedings of the 27th international conference on Software engineering*. 2005, pp. 342–351.

[15] Ernie Cohen et al. “VCC: A practical system for verifying concurrent C”. In: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 2009, pp. 23–42.

[16] Edward E Cureton. “Rank-biserial correlation”. In: *Psychometrika* 21.3 (1956), pp. 287–290.

[17] 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*. Springer. 2008, pp. 337–340.

[18] Ankush Desai et al. “P: safe asynchronous event-driven programming”. In: *ACM SIGPLAN Notices* 48.6 (2013), pp. 321–332.

[19] Jeremy Dick and Alain Faivre. “Automating the generation and sequencing of test cases from model-based specifications”. In: *International Symposium of Formal Methods Europe*. Springer. 1993, pp. 268–284.

[20] Aleksandr Fedchin et al. “A toolkit for automated testing of Dafny”. In: *NASA Formal Methods Symposium*. Springer. 2023, pp. 397–413.

[21] Jean-Christophe Filliâtre and Andrei Paskevich. “Why3 — Where Programs Meet Provers”. In: *Programming Languages and Systems*. Ed. by Matthias Felleisen and Philippa Gardner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 125–128. ISBN: 978-3-642-37036-6.

[22] Samuel Jimenez Gil, Manuel I Capel, and Gabriel Olea Olea. “Automatic test cases generation from formal contracts”. In: *Information and Software Technology* 172 (2024), p. 107467.

[23] Divya Gopinath, Razieh Nokhbeh Zaeem, and Sarfraz Khurshid. “Improving the effectiveness of spectra-based fault localization using specifications”. In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 2012, pp. 40–49.

[24] James Gosling. *The Java language specification*. Addison-Wesley Professional, 2000.

[25] Klaus Havelund and Thomas Pressburger. “Model checking java programs using java pathfinder”. In: *International Journal on Software Tools for Technology Transfer* 2.4 (2000), pp. 366–381.

[26] Charles Antony Richard Hoare. “An axiomatic basis for computer programming”. In: *Communications of the ACM* 12.10 (1969), pp. 576–580.

[27] Yue Jia and Mark Harman. “An analysis and survey of the development of mutation testing”. In: *IEEE transactions on software engineering* 37.5 (2010), pp. 649–678.

[28] James A Jones, Mary Jean Harrold, and John T Stasko. “Visualization for fault localization”. In: *in Proceedings of ICSE 2001 Workshop on Software Visualization*. 2001.

[29] Brian W Kernighan and Dennis M Ritchie. *The C programming language*. Pearson Educación, 1988.

[30] Florent Kirchner et al. “Frama-C: A software analysis perspective”. In: *Formal aspects of computing* 27.3 (2015), pp. 573–609.

[31] Steve Klabnik and Carol Nichols. *The Rust programming language*. No Starch Press, 2023.

[32] Leslie Lamport. *Specifying systems*. Vol. 388. Addison-Wesley Boston, 2002.

[33] Andrea Lattuada et al. “Verus: A practical foundation for systems verification”. In: *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. 2024, pp. 438–454.

[34] K Rustan M Leino. "Dafny: An automatic program verifier for functional correctness". In: *International conference on logic for programming artificial intelligence and reasoning*. Springer. 2010, pp. 348–370.

[35] Chao Liu et al. "Statistical debugging: A hypothesis testing-based approach". In: *IEEE Transactions on software engineering* 32.10 (2006), pp. 831–848.

[36] Chloe Loughridge et al. "Dafnybench: A benchmark for formal software verification". In: *arXiv preprint arXiv:2406.08467* (2024).

[37] David R MacIver, Zac Hatfield-Dodds, et al. "Hypothesis: A new approach to property-based testing". In: *Journal of Open Source Software* 4.43 (2019), p. 1891.

[38] Henry B Mann and Donald R Whitney. "On a test of whether one of two random variables is stochastically larger than the other". In: *The annals of mathematical statistics* (1947), pp. 50–60.

[39] Wes Masri and Rana Abou Assi. "Prevalence of coincidental correctness and mitigation of its impact on fault localization". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23.3 (2014), pp. 1–28.

[40] John W McCormick and Peter C Chapin. *Building high integrity applications with SPARK*. Cambridge University Press, 2015.

[41] Bertrand Meyer. "Applying'design by contract'". In: *Computer* 25.10 (1992), pp. 40–51.

[42] Bertrand Meyer. "Eiffel: A language and environment for software engineering". In: *Journal of Systems and Software* 8.3 (1988), pp. 199–246.

[43] Bertrand Meyer. "Using the EiffelStudio environment". In: *Touch of Class: Learning to Program Well with Objects and Contracts*. Springer, 2009, pp. 843–846.

[44] Bertrand Meyer et al. "Programs that test themselves". In: *Computer* 42.9 (2009), pp. 46–55.

[45] Martin Monperrus. "The living review on automated program repair". PhD thesis. HAL Archives Ouvertes, 2018.

[46] Glenford J Myers et al. *The art of software testing*. Vol. 2. Wiley Online Library, 2004.

[47] Gyumin Nam and Geunseok Yang. "LLMLoc: A Structure-Aware Retrieval System for Zero-Shot Bug Localization". In: *Electronics* 14.21 (2025), p. 4343.

[48] Francisco Oliveira, Alexandra Mendes, and Carolina Carreira. "What Challenges Do Developers Face When Using Verification-Aware Programming Languages?" In: *arXiv preprint arXiv:2506.23696* (2025).

[49] OpenAI. *Models - OpenAI API Documentation*. Accessed: 2026-07-03. 2026. URL: https://developers.openai.com/api/docs/models/gpt-5-mini.

[50] Spencer Pearson et al. "Evaluating and improving fault localization". In: *Proceedings of the 39th International Conference on Software Engineering (ICSE)*. IEEE. 2017, pp. 609–620.

[51] Yu Pei et al. "Automated Fixing of Programs with Contracts". In: *IEEE Transactions on Software Engineering* 40.5 (2014), pp. 427–449. DOI: 10.1109/TSE.2014.2312918.

[52] Ricardo Peña et al. "SMT-based test-case generation and validation for programs with complex specifications". In: *Analysis, Verification and Transformation for Declarative Programming and Intelligent Systems: Essays Dedicated to Manuel Hermenegildo on the Occasion of His 60th Birthday*. Springer, 2023, pp. 188–205.

[53] Guillaume Petiot et al. “How testing helps to diagnose proof failures”. In: *Formal Aspects of Computing* 30.6 (2018), pp. 629–657.

[54] Md Nakhla Rafi et al. “The impact of input order bias on large language models for software fault localization”. In: *arXiv preprint arXiv:2412.18750* 14 (2024).

[55] Francisco Rebello de Andrade et al. “Specification-driven unit test generation for Java generic classes”. In: *International Conference on Integrated Formal Methods*. Springer. 2012, pp. 296–311.

[56] Raymond Reiter. “A theory of diagnosis from first principles”. In: *Artificial intelligence* 32.1 (1987), pp. 57–95.

[57] Raul Santelices et al. “Lightweight fault-localization using multiple coverage types”. In: *2009 IEEE 31st International conference on software engineering*. IEEE. 2009, pp. 56–66.

[58] Melika Sepidband, Hung Viet Pham, and Hadi Hemmati. “On the Role of Fault Localization Context for LLM-Based Program Repair”. In: *arXiv preprint arXiv:2604.05481* (2026).

[59] Álvaro Silva, Alexandra Mendes, and Ruben Martins. “Inferring multiple helper Dafny assertions with LLMs”. In: *arXiv preprint arXiv:2511.00125* (2025).

[60] Charles Spearman. “The proof and measurement of association between two things.” In: *American Journal of Psychology* (1961).

[61] Hovhannes Tamoyan et al. “SHERLOC: Structured Diagnostic Localization for Code Repair Agents”. In: *arXiv preprint arXiv:2606.24820* (2026).

[62] The dafny-lang community. *Dafny Reference Manual*. Accessed: 2026-06-18. 2026. URL: https://dafny.org/dafny/DafnyRef/DafnyRef.

[63] Guido Van Rossum, Fred L Drake, et al. *Python reference manual*. Vol. 111. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[64] Sofia Vieira Pinto et al. “DSpec2Test: Specification-Driven Test Generation in Dafny”. In: *2026 IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2026.

[65] Jason Wei et al. “Chain-of-thought prompting elicits reasoning in large language models”. In: *Advances in neural information processing systems* 35 (2022), pp. 24824–24837.

[66] Mark Weiser. “Program slicing”. In: *IEEE Transactions on software engineering* 4 (1984), pp. 352–357.

[67] Ming Wen et al. “Historical spectrum based fault localization”. In: *IEEE Transactions on Software Engineering* 47.11 (2019), pp. 2348–2368.

[68] Frank Wilcoxon. “Individual comparisons by ranking methods”. In: *Biometrics bulletin* 1.6 (1945), pp. 80–83.

[69] Nicky Williams et al. “Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis”. In: *European Dependable Computing Conference*. Springer. 2005, pp. 281–292.

[70] W Eric Wong et al. “A survey on software fault localization”. In: *IEEE Transactions on Software Engineering* 42.8 (2016), pp. 707–740.

[71] W Eric Wong et al. “The DStar method for effective software fault localization”. In: *IEEE Transactions on Reliability* 63.1 (2013), pp. 290–308.

[72] Jim Woodcock et al. “Formal methods: Practice and experience”. In: *ACM computing surveys (CSUR)* 41.4 (2009), pp. 1–36.

[73] Martin R Woodward. “Mutation testing—its origin and evolution”. In: *Information and Software Technology* 35.3 (1993), pp. 163–169.

[74] Valentina Wu, Alexandra Mendes, and Alexandre Abreu. “Specification-guided repair of arithmetic errors in Dafny programs using LLMs”. In: *arXiv preprint arXiv:2507.03659* (2025).

[75] Yonghao Wu et al. “Large language models in fault localisation”. In: *arXiv preprint arXiv:2308.15276* (2023).

[76] Jifeng Xuan and Martin Monperrus. “Learning to combine multiple ranking metrics for fault localization”. In: *2014 IEEE international conference on software maintenance and evolution*. IEEE. 2014, pp. 191–200.

[77] Aidan ZH Yang et al. “Large language models for test-free fault localization”. In: *Proceedings of the 46th IEEE/ACM international conference on software engineering*. 2024, pp. 1–12.

[78] Kuat T Yessenov. “A lightweight specification language for bounded program verification”. PhD thesis. Massachusetts Institute of Technology, 2009.

Appendix A

Authorisation

This appendix contains a visual reproduction of the formal authorisation from co-authors confirming the independent nature of the work presented in this dissertation. The original document, digitally signed by all involved parties, has been submitted as a separate supplementary file.

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Co-Author Declaration and Authorisation

We, the undersigned, co-authors of the manuscript titled *"DSpec2Test: Specification-Driven Test Generation in Dafny"*, hereby declare and confirm that Sofia Vieira Pinto is the first and primary author of the aforementioned manuscript. We expressly authorise her to include and adapt text from this manuscript within her dissertation.

Furthermore, we confirm that the collaborative evaluations and results present in the original manuscript have been entirely excluded from her dissertation. We acknowledge that all implementations, dataset testing, and evaluations presented in the dissertation were conducted independently and represent the exclusive work of Sofia Vieira Pinto.

By signing below, we agree to these terms and formally validate the independent nature of the work presented in her dissertation.

Assinado por: **Álvaro Manuel Festas Pereira da Silva** Num. de Identificação: 14290583 Data: 2026.06.12 15:50:04 -0400	Assinado por: **Alexandra Sofia Ferreira Mendes** Num. de Identificação: 11995167 Data: 2026.06.14 22:37:53 +0100	Assinado por: **João Carlos Pascoal de Faria** Num. de Identificação: 03703791 Data: 2026.06.14 22:40:18+01'00'
Álvaro Silva	Alexandra Mendes (Supervisor)	João Pascoal Faria
Date: _____	Date: _____	Date: _____