DSpec2Test: Specification-Driven Test Generation in Dafny

Sofia Vieira Pinto
INESC TEC, Faculdade de Engenharia, Universidade do Porto
Porto, Portugal
sofia.p.pt@gmail.com

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

João Pascoal Faria
INESC TEC, Faculdade de Engenharia, Universidade do Porto
Porto, Portugal
jpf@fe.up.pt

Alexandra Mendes*✉
INESC TEC, Faculdade de Engenharia, Universidade do Porto
Porto, Portugal
afmendes@fe.up.pt

Abstract

Verification-aware languages, such as Dafny, integrate logical constructs into code and enable automatic verification of program correctness. However, tests remain helpful in scenarios that verification alone does not address (e.g., to support test-driven development). Existing Dafny test generation tools are implementation-based, limiting their applicability in this context.

We present DSpec2Test, a specification-driven test generation tool for Dafny that automatically derives tests from formal specifications, without considering implementation details. Our tool extends Dafny's `generate-tests` command with a new black-box mode based on Disjunctive Normal Form (DNF) equivalence class partitioning and optional Boundary Value Analysis (BVA). DSpec2Test relies on the Z3 SMT solver to synthesize inputs and expected outputs that meet the specification-derived constraints.

We evaluate DSpec2Test on programs from DafnyBench mutated using MutDafny and compare it against Dafny's existing implementation-driven *Block* mode. DSpec2Test achieves a 93.9% mutation kill rate on a dataset of 131 mutants, outperforming *Block*'s 82.4%, and uniquely killing 17 mutants. These results suggest that specification-driven testing is an effective and complementary approach for testing Dafny programs.

Demo: https://youtu.be/mK1EeJfinRQ
Code: https://github.com/VeriFixer/DSpec2Test
Prebuilt docker image: https://doi.org/10.5281/zenodo.21191158

CCS Concepts

• **Software and its engineering → Formal software verification**; **Software testing and debugging**; *Software verification.*

Keywords

Dafny, Verification-aware languages, Test generation, Specification-driven test generation, Software verification, Formal methods

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

1 Introduction

Verification-aware languages, such as Dafny [7], integrate preconditions (*requires*), postconditions (*ensures*), and invariants (*invariant*) directly into the code, enabling automated detection of specification violations during development. However, when verification fails, error messages are often not sufficiently informative to identify the underlying fault [12], costing developers significant time and effort. Here, tests can serve as a complementary mechanism by providing concrete executions that help locate and understand the fault.

Within Dafny, test generation from implementation code is already supported by DTest [4], accessible via the `generate-tests` command. DTest targets code coverage to provide additional assurance that compilation of Dafny programs to target languages (e.g., C# or Java) preserves the properties verified in Dafny, attempting to mitigate the gap left by Dafny's unverified compilation pipeline. It is part of a broader toolkit that also includes DUnit (unit testing) and DMock (mock testing).

While DTest is useful for validating compiled implementations, there are scenarios in which generating tests directly from specifications is more suitable. Implementation-based tests are constrained by the structure and behaviour of the existing code, whereas tests based on the specification focus on the intended semantics of the program. This distinction is especially relevant during early stages of development, where specifications may already exist but implementations are still incomplete or evolving. In such settings, specification-driven tests support a test-driven development (TDD) style, where tests are derived from the specification first and guide the subsequent implementation. This can help to pinpoint errors as soon as implementations deviate from the intended behavior.

We present DSpec2Test, a tool for specification-driven test generation in Dafny. This tool is provided through an additional mode (*Spec*) for Dafny's `generate-tests` command. It analyses the preconditions and postconditions of the methods and functions to create Disjunctive Normal Form (DNF) clauses, producing a comprehensive pool of tests. 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.

Table 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`	$\{ A \wedge B,\ \neg A \wedge C \}$

Our main **contributions** are:

- ★ A specification-to-test pipeline (instantiated for Dafny) that combines (i) an *automatic Safe DNF* decomposition of arbitrary pre/postconditions, preserving short-circuit guard semantics, (ii) boundary value analysis on the partition-derived constraints, and (iii) an encoding of each test goal as an `assume`/`assert false` stub that delegates the search for concrete inputs and expected outputs to Z3.
- ★ DSpec2Test, the first tool for *specification-based conformance testing* in Dafny. It derives tests from `requires`/`ensures` clauses and can support test-driven development, specification validation, and fault localisation.
- ★ Evidence that specification-based and implementation-based testing reach complementary fault classes gathered through an empirical evaluation on 131 mutants from DafnyBench, in which DSpec2Test kills 93.9 %, outperforming DTest (82.4 %) and uniquely killing 17 mutants that DTest misses.

2 Formal Specification-Driven Test Generation

Formal specification-driven test generation is an approach that constructs test cases directly from formal specifications rather than relying on existing implementation code. DSpec2Test follows this approach by actively exercising all preconditions (*requires*) and postconditions (*ensures*), and by automatically applying the well-established black-box testing strategies of equivalence class partitioning (ECP) and boundary value analysis (BVA) [11]. DSpec2Test relies on an SMT-solver (Z3) to find concrete test data that satisfy the target test conditions produced by ECP and BVA.

2.1 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 [11]. We automate ECP by analysing the method's contract clauses and converting them into Disjunctive Normal Form (DNF). Alternative behaviours (depending on different input classes) are typically encoded in the method's postconditions, which is why our starting point is the conjunction of all preconditions (`requires`) and postconditions (`ensures`). Each DNF branch then constrains both inputs and outputs, allowing us to generate concrete values for the test inputs and expected outputs (when uniquely determined by the contract).

Listing 1: Example Dafny contract: `ConditionalGet`.

```
1 method ConditionalGet(s: seq<int>, x: int) returns (r:
  int)
2   requires |s| > 0
3   ensures x < 0 ==> r == -1          // Q1
4   ensures x >= |s| ==> r == 1        // 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 1 yields a branch where $r = s[x]$ is evaluated without the guard $0 \le x < |s|$, raising an out-of-bounds error. To avoid this, DSpec2Test's default *Safe DNF* mode applies the short-circuit-safe decomposition rules of Table 1, which produce mutually exclusive, guarded branches. Applied to `ConditionalGet`, each implication splits into two cases:

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

The cross product, conjoined with the precondition $|s| > 0$, yields eight candidate equivalence classes. Five are infeasible (Z3 returns UNSAT) due to mutually exclusive guards. The three feasible classes, after simplifying the redundant guard negations, are:

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

If exhaustive combination testing is required and can be safely applied, DSpec2Test also supports *Full DNF* (`-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 $\{A \wedge B,\ A \wedge \neg B,\ \neg A \wedge B\}$.

2.2 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. Therefore, when the `-bva` flag is provided, the tool complements the DNF phases with Boundary Value Analysis (BVA).

For every partition, the tool generates multiple test scenarios. The first test leaves all variables free, relying only on the base DNF constraints. 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 to an interval, the tool pins it to its exact valid edges. When a variable is unbounded, the tool uses default extremes (-100 and 100) for numeric variables. Sequences, sets, and maps are tested for lengths 0, 1, and > 1.

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

2.3 Concrete Test Data Generation

Like DTest, DSpec2Test relies on Z3 to find concrete test data that satisfy each DNF branch and BVA constraint. The relevant constraints are encoded as `assume` statements in the method's body, followed by an `assert false` that forces the verifier to emit a counterexample, providing concrete test inputs and expected outputs. For class C_3 of `ConditionalGet`, the instrumented body is:

Listing 2: Instrumented body for class C_3.

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

A counterexample such as `s=[3]`, `x=0`, `r=3` is materialised as a runnable Dafny test that sets up the input parameters, calls the method under test, and checks its outputs against concrete expected values produced by Z3 (when `-simplify` is set, and the postconditions uniquely constrain the outputs) or the original postconditions instantiated for the known inputs.

Listing 3: Generated test with `-simplify` flag for class C_3.

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

2.4 Implementation Pipeline

Figure 1 summarises how the components described above are integrated into Dafny's `generate-tests` command, in three stages.

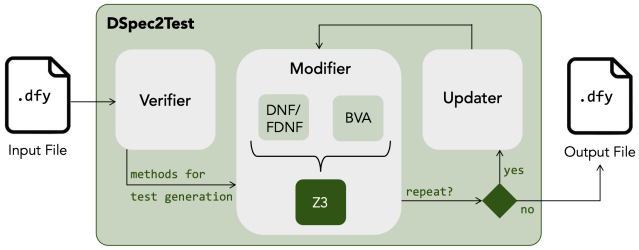

Figure 1: DSpec2Test pipeline. An input file is processed by the *Verifier*, where methods that are marked with `{:testEntry}` or fail verification are passed to the *Modifier* for DNF generation and optional BVA. The *Updater* iteratively refines the process by generating additional tests while ensuring 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, BVA boundary pins derived in the previous subsections, as illustrated in Listing 2.

The *Updater* drives iteration when `-repeat` is higher than one and `-bva` is not selected. After each generated test, it appends a fresh `assume` clause that excludes the previous input, so that the next solver call returns a different one. For instance, if `x` was assigned to `0`, an `assume x != 0;` statement is added before the next iteration.

3 Empirical Evaluation

3.1 Methodology

We compare two strategies of Dafny's `generate-tests`: ***Block***, the existing white-box mode provided by DTest, which targets path/branch coverage of the implementation; and ***Spec_bva***, the black-box mode introduced by DSpec2Test (*Spec*) with the `-bva` flag, which targets equivalence class and boundary value coverage of the specification.

We evaluate fault-detection capability on a corpus of 20 programs sampled from DafnyBench [8], selected to contain methods with formal specifications (pre/postconditions, loop invariants) amenable to automated test generation. Currently, neither of the tools (DTest and DSpec2Test) supports mutable state (e.g., arrays), class fields, and non-determinism (e.g., `:|` operator), so these constraints were also considered during the sampling process. Each program was annotated with {:testEntry} so tests could be generated from the original (unmutated) code.

We applied mutation testing, a well-established technique for evaluating test adequacy [13, 15], using MutDafny [1], a mutation tool that injects syntactic faults (relational/arithmetic operator replacement, statement deletion, conditional negation, etc.) into Dafny implementations. For each program, up to 10 mutants were generated, retaining only those that fail verification, yielding 200 mutants in total. A mutation that fails verification may preserve observable behaviour — it may simply break verification constructs. We mark such cases as *equivalent* mutants.

The evaluation pipeline, using Dafny 4.11.0 and Z3 4.12.1, (1) generates tests from the original program, (2) runs a safety check ensuring that the tests pass on the original (unmutated) code, and (3) runs each suite against every mutant to determine kill/survive/timeout status.

3.2 Results and Discussion

We base the comparison on the 17/20 programs supported by both strategies (three were not supported, crashing during test creation for at least one of them due to either exponential complexity or tests not meeting the preconditions), totaling 170 mutants.

Analysis of surviving mutants. We manually inspected the mutants that survived both strategies to determine if they could be realistically killed, and found 22 *non-compiling* mutants and 17 *equivalent* mutants (observationally equivalent to the original during execution), leaving *131 realistically-killable mutants*.

Analysis of timeouts. A small number of mutants timed out at the 300 sec kill-check budget, which is vastly longer than a standard Dafny test execution. A manual inspection confirmed that these were all caused by infinite loops introduced by the mutations, exposed when exercised by the generated test inputs. We therefore credit timeouts as kills.

Kill rates. Table 2 reports per-strategy kill rates and the overlap between the two tools on the 131 that can realistically be killed. *Spec_bva* kills 123 (93.9%), *Block* kills 108 (82.4%); the union covers 125 (95.4%), with *Spec_bva* uniquely killing 17, *Block* uniquely killing 2, and only 6 mutants killed by neither tool.

Test volume vs. kill yield. *Spec_bva* generates 6.6 tests per file on average (860 across the 131 files), versus ~1.6 for *Block*. However, this volume is heavily redundant: extracting the smallest failing test index per mutant from the runner output, we find 52 % of explicit-failure kills come from the first test, and 73 % from the first three.

Table 2: Mutation kill rates and kill overlap on 131 realistically-killable mutants. Timeouts (TO) are runtime infinite loops credited as kills. Killed = FAIL + TO.

	FAIL	TO	Killed	Surv.	Kill%
Block (WB)	99	9	108	23	82.4%
Spec_bva (BB)	106	17	123	8	**93.9%**
Union	—	—	125	6	95.4%
Killed by *Block* only	—	—	2	—	1.5%
Killed by *Spec_bva* only	—	—	17	—	13.0%

Complementarity. Of the 19 mutants killed by exactly one tool out of 131, 17 (89 %) are unique to *Spec_bva* and 2 are unique to *Block*, indicating that ECP and BVA on the specification reaches faults that path coverage on the implementation does not, and vice-versa, highlighting their complementarity.

Cost. *Spec_bva* is approximately 1.6× slower in wall-clock time (7889 s vs. 4867 s), reflecting the larger generated suite.

4 Related Work

In addition to DTest [4], Delfy [2] also generates tests for Dafny, using dynamic symbolic execution with Z3 to diagnose verification failures. Concolic tools such as PathCrawler [18] (for C, within Frama-C [6]) similarly generate inputs from executable code for structural coverage. Our approach differs from these as it is fully spec-driven and requires no implementation.

AutoTest [10] is a black-box testing framework for Eiffel that combines random sampling with existing contracts to generate tests for object-oriented code. We instead use SMT solver calls for a more directed exploration of the input space.

Other spec-driven test generators apply related techniques to formal contracts. Gil et al. [5] target SPARK/Ada (also a verification-aware language) with ECP guided by user-written `Contract_Cases` and Z3py called externally, but, unlike our approach, do not automatically derive DNF-based equivalence classes from arbitrary pre/postconditions or reuse the host verifier. Peña et al. [14] use Z3 to derive test inputs from preconditions in a custom specification language, but do not partition the whole contract via DNF for directed, human-readable tests. Rebello de Andrade et al. [16] apply Full DNF to axiomatic specifications of Java generics — a precursor to our Safe DNF for method contracts — and use Alloy Analyzer (a SAT solver) to find satisfying models for each minterm, but do not target a verification-aware language or apply boundary value analysis.

Property-based testing tools, such as QuickCheck [3] and Hypothesis [9], also generate tests from properties, but these are executable, partial, and input-centric, not full correctness specifications.

5 Future Work

We plan to refine DNF clauses with additional constraints so that every postcondition literal genuinely constrains the admissible outputs for the input Z3 picks. Consider finding the last occurrence of x in a sorted sequence a known to contain it, as shown in the example in Listing 4.

Listing 4: Last occurrence of x in a sorted sequence.

```
method LastIdx(a: seq<int>, x: int) returns (i: int)
 requires sorted(a) && x in a
 ensures 0 <= i < |a|      // Q1 (range guard)
 ensures a[i] == x          // Q2
 ensures x !in a[i+1..]     // Q3
```

Z3 picks the minimal model `a=[x]`, `i=0`, on which Q_2 and Q_3 hold vacuously; a buggy implementation returning `a[0]` passes. Exercising both non-vacuously requires multiple copies of x plus a distinct element (e.g., `a=[5,5,6]`, `x=5`). A *relevance check* forces this: for each output-referring Q_k (excluding range guards), search for an alternate $i_k \neq i$ satisfying the other literals but violating Q_k. Equivalently, satisfy the extended contract, as shown in Listing 5.

Listing 5: Extended contract for relevance checking.

```
method LastIdx'(a: seq<int>, x: int)
             returns(i: int, ghost i2:int, ghost i3:int)
 requires sorted(a) && x in a
 ensures 0 <= i < |a| && a[i] == x && x !in a[i+1..]
 ensures i2!=i && 0<=i2<|a| && a[i2]!=x && x !in a[i2+1..] //Q2
rel.
 ensures i3!=i && 0<=i3<|a| && a[i3]==x && x in a[i3+1..] //Q3
rel.
```

DSpec2Test also lacks *function inlining* in specifications and (like DTest) *support for mutable state* (arrays and class fields). Both are planned to broaden applicability.

We also plan to expand the evaluation to a broader sample of programs.

6 Conclusion

We presented DSpec2Test, a specification-driven test generation tool for Dafny that derives runnable tests from the `requires` and `ensures` clauses of a method, without examining the implementation. While specification-based testing is well-established for other contract languages (e.g., AutoTest for Eiffel, JMLUnit for Java), DSpec2Test is, to the best of our knowledge, the first Dafny tool that supports test generation prior to (or independently of) the implementation — a capability complementary to the existing implementation-driven DTest. As far as we are aware, it is also the first tool that performs automatic DNF partitioning of method contracts.

Our evaluation shows that DTest and DSpec2Test are complementary, with DSpec2Test uniquely killing 17 mutants that *Block* misses out of 131, indicating that specification- and implementation-based testing reach different fault classes. DSpec2Test is therefore useful both as a stand-alone tool for early-stage and TDD-style workflows and as a complement to existing white-box tooling such as DTest, useful for implementation coverage. However, for maximum fault localisation and rigor, both strategies should be employed.

Data Availability Statement

The data associated with this work are publicly available through a Zenodo repository [17]. The dataset can be accessed using the following DOI: https://doi.org/10.5281/zenodo.21191158

Acknowledgments

Sofia Vieira Pinto, João Pascoal Faria, and Alexandra Mendes were funded by National Funds through the FCT - Fundação para a Ciência e a Tecnologia, I.P. (Portuguese Foundation for Science and Technology) within the project VeriFixer, with reference 2023.15557.PEX (DOI: 10.54499/2023.15557.PEX).

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

References

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

[2] Maria Christakis, K. Rustan M. Leino, Peter Müller, and Valentin Wüstholz. 2016. Integrated Environment for Diagnosing Verification Errors. In *Tools and Algorithms for the Construction and Analysis of Systems*, Marsha Chechik and Jean-François Raskin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 424–441. doi:10.1007/978-3-662-49674-9_25

[3] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. *SIGPLAN Not.* 35, 9 (Sept. 2000), 268–279. doi:10.1145/357766.351266

[4] Aleksandr Fedchin, Tyler Dean, Jeffrey S. Foster, Eric Mercer, Zvonimir Rakamarić, Giles Reger, Neha Rungta, Robin Salkeld, Lucas Wagner, and Cassidy Waldrip. 2023. A Toolkit for Automated Testing of Dafny. In *NASA Formal Methods*, Kristin Yvonne Rozier and Swarat Chaudhuri (Eds.). Springer Nature Switzerland, Cham, 397–413. doi:10.1007/978-3-031-33170-1_24

[5] Samuel Jiménez Gil, Manuel I. Capel, and Gabriel Olea Olea. 2024. Automatic test cases generation from formal contracts. *Information and Software Technology* 172 (2024), 107467. doi:10.1016/j.infsof.2024.107467

[6] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A software analysis perspective. *Formal aspects of computing* 27, 3 (2015), 573–609. doi:10.1007/s00165-014-0326-7

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

[8] Chloe Loughridge, Qinyi Sun, Seth Ahrenbach, Federico Cassano, Chuyue Sun, Ying Sheng, Anish Mudide, Md Rakib Hossain Misu, Nada Amin, and Max Tegmark. 2024. Dafnybench: A benchmark for formal software verification. *arXiv preprint arXiv:2406.08467* (2024). doi:10.48550/arXiv.2406.08467

[9] David R. MacIver, Zac Hatfield-Dodds, and Many Other Contributors. 2019. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software* 4, 43 (2019), 1891. doi:10.21105/joss.01891

[10] Bertrand Meyer, Arno Fiva, Ilinca Ciupa, Andreas Leitner, Yi Wei, and Emmanuel Stapf. 2009. Programs That Test Themselves. *Computer* 42, 9 (2009). doi:10.1109/MC.2009.296

[11] Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. 2012. *The art of software testing*. Vol. 2. John Wiley and Sons, Ltd. doi:10.1002/9781119202486

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

[13] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Chapter Six - Mutation Testing Advances: An Analysis and Survey. Advances in Computers, Vol. 112. Elsevier, 275–378. doi:10.1016/bs.adcom.2018.03.015

[14] Ricardo Peña, Jaime Sánchez-Hernández, Miguel Garrido, and Javier Sagredo. 2023. *SMT-Based Test-Case Generation and Validation for Programs with Complex Specifications*. Springer Nature Switzerland, Cham, 188–205. doi:10.1007/978-3-031-31476-6_10

[15] Goran Petrović and Marko Ivanković. 2018. State of mutation testing at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice* (Gothenburg, Sweden) *(ICSE-SEIP '18)*. Association for Computing Machinery, New York, NY, USA, 163–171. doi:10.1145/3183519.3183521

[16] Francisco Rebello de Andrade, João P. Faria, Antónia Lopes, and Ana C. R. Paiva. 2012. Specification-Driven Unit Test Generation for Java Generic Classes. In *Integrated Formal Methods*, John Derrick, Stefania Gnesi, Diego Latella, and Helen Treharne (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 296–311. doi:10.1007/978-3-642-30729-4_21

[17] Sofia Vieira Pinto, Álvaro F. Silva, João Pascoal Faria, and Alexandra Mendes. 2026. *DSpec2Test: Specification-Driven Test Generation in Dafny (Artifact)*. doi:10.5281/zenodo.21191158

[18] Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. 2005. PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis. In *Dependable Computing - EDCC 5*, Mario Dal Cin, Mohamed Kaâniche, and András Pataricza (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 281–292. doi:10.1007/11408901_21

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