DafnyFix: Single-Transformation Automated Repair in Dafny

Isabel Amaral
INESC TEC, Faculty of Engineering, University of Porto
Porto, Portugal
up202006677@fe.up.pt

Álvaro F. Silva
INESC TEC, Faculty of Engineering, University of Porto
Porto, Portugal
up201603524@fe.up.pt

João F. Ferreira
INESC-ID and Faculty of Engineering, University of Porto
Porto, Portugal
joao@joaoff.com

Alexandra Mendes*✉
INESC TEC, Faculty of Engineering, University of Porto
Porto, Portugal
afmendes@fe.up.pt

Abstract

Formal verification using verification-aware languages, like Dafny, ensures a program's conformance to its specification. Assuming a correct and complete specification, a verification failure may indicate an implementation fault that the developer must identify and correct. However, debugging verification failures is particularly challenging, as verification diagnostics do not directly hint at the implementation changes required for verification to succeed. Automated Program Repair has long been explored for traditional programming languages to reduce manual debugging effort, yet it remains largely underexplored in verification-aware contexts.

We propose DafnyFix, a repair framework for Dafny that searches over a space of single-transformation candidate patches generated through three complementary heuristics: existing mutation testing operators, newly introduced repair-oriented mutation operators, and state-guided repair templates inspired by AutoFix. We evaluate DafnyFix's effectiveness on a dataset of implementation faults in LLM-generated Dafny programs curated from the *dafny-synthesis* benchmark, and successfully repair 62.5% of the faults, showing that small deterministic transformations can repair a substantial number of faulty LLM-generated implementations. Our results further show that the proposed repair heuristics are complementary, targeting different categories of implementation faults. Additionally, we identify six fault patterns of LLM-generated programs in our dataset that can be corrected by small program transformations.

CCS Concepts

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

Keywords

Dafny, Verification-aware languages, Automated Program Repair, Search-Based Repair, Software verification

1 Introduction

As humans increasingly rely on software in their daily lives, minimizing software faults becomes crucial. Verification-aware programming languages, such as Dafny [18] and Verus [13], are powerful tools for developing software proven correct against a formal specification, integrated directly into the code through the use of annotations (e.g., pre-conditions, post-conditions, and invariants). Assuming the provided specification precisely reflects the developers' intent, a verified program is guaranteed to satisfy the intended behavior. Unlike software testing, which only validates correctness for the program inputs that are exercised, formal verification establishes correctness for all executions that satisfy the specification.

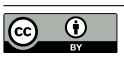
Despite the usefulness of verification-aware languages, when verification fails, developers struggle to identify which part of the program does not conform to the specification and how to fix the issue. Poor error messages and inadequate feedback are some key challenges when using these languages [24]. Although verification tools provide diagnostic information, such as failing conditions and counterexamples, this often does not translate into actionable feedback on how to modify the program so that verification succeeds. Automated Program Repair (APR), which has been widely explored in traditional programming languages (e.g., Java, C) to automatically fix faults and reduce the burden of debugging, has the potential to also assist developers of verification-aware languages.

APR has been widely studied over the past two decades [6, 7, 17, 22]. Still, despite formal specifications offering a more comprehensive correctness oracle than the traditional test suites, there is a lack of solutions for verification-aware languages. In Dafny, particularly, only specification repair [1] and arithmetic implementation-error repair with Large Language Models (LLMs) [28] have been explored.

While prior repair techniques for Dafny focus on narrow subsets of faults, we investigate repair across a broader range of implementation fault types. In addition, deterministic repair remains underexplored in the context of Dafny (and in verification-aware languages in general), but we argue that several common verification failures can be repaired systematically, without probabilistic generation. Deterministic repair is reproducible, predictable, and may incur lower monetary cost compared to LLM-based approaches, producing patches that can be directly traced to specific verification failures. It therefore provides a practical alternative for recurring repair patterns and a baseline for understanding which repairs require LLM-based reasoning. Furthermore, as LLMs are increasingly used for code generation [19, 21, 29], deterministic repair techniques can prove useful in systematically correcting common errors introduced by LLMs, providing predictable and explainable repair results, while preserving the bulk of the generated implementation.

In this paper, we present **DafnyFix**, a repair framework that searches over a space of single-transformation candidate patches to address verification failures in Dafny caused by faults in a program's implementation. While verification failures may result from a variety of reasons (e.g., implementation errors, specification errors, lack of verification annotations), this work targets implementation faults, assuming the correctness and completeness of the specification. The patches that DafnyFix applies in this preliminary work are

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



minimal, in the sense that each considers a single fault location and results from a single transformation. This allows us to isolate and evaluate the effectiveness of different patch generation strategies.

DafnyFix starts by attempting repair using a collection of mutation operators from the MutDafny [2] mutation testing tool for Dafny. While some of these already provide repair capabilities, because they were designed to mimic faults introduced by Dafny programmers (bug-introducing-oriented), we also extend a subset of these operators with their inverse (fixing-oriented) behavior. To target more complex faults, DafnyFix applies state-guided templates inspired by the AutoFix [25] tool for Eiffel, designed to avoid likely faulty program states. The patches generated by these three heuristics form our search space. Rather than exploring it exhaustively, we constrain the search using prior work on fault localization (FL) in verification-aware languages [26], which identifies both program lines and states suspected to be the source of the fault.

We validate our approach on 16 programs with implementation errors curated from a dataset of LLM-generated Dafny programs, *dafny-synthesis* [21], achieving a promising repair rate of 62.5%, despite our minimal patching setup. Beyond repair effectiveness, we also study the kinds of LLM faults present in our dataset and the repair heuristics most effective at correcting them.

In summary, the main **contributions** of this paper are:

- **DafnyFix: a single-transformation search-based repair framework** combining mutation-based and state-guided template-based patch generation for repairing Dafny programs with implementation errors.
- **Evidence of DafnyFix's usefulness in Dafny program repair**, demonstrated through an evaluation on a curated dataset of real LLM-generated programs with implementation faults, reaching a promising repair success of 62.5%[1].
- An **empirical analysis of the faults** present in our dataset and their repairability.

2 Motivational Example

To illustrate the kinds of minimal implementation faults that prevent successful verification and are targeted in this work, let us consider the LLM-generated program shown in Listing 1, which checks whether an input value `n` can be expressed as the difference of the squares of two numbers, `i` and `j`, both less than or equal to `n`. For computing the difference of squares, the original implementation only explores values of i up to $\sqrt{n}$, as visible in Line 8. While this loop guard seems reasonable, since we are dealing with squares, it prematurely terminates the search, discarding valid solutions that require a large value of `i` and a low value of `j`. For instance, for input 3, whose square root is ≈1.73, the loop only explores `i` = 0 and `i` = 1, missing the solution `i` = 2 ∧ `j` = 1 ($3 = 2^2 - 1^2$). One possible correct implementation explores every integer from 0 to `n`, which can be achieved by replacing one of the usages of the `i` variable in the loop guard with 1, as done in Line 9. This change corresponds to a mutation implemented by MutDafny's expression value replacement (EVR) operator, which replaces an expression with a default literal value of its type, and is supported by DafnyFix.

[1]The framework and all the scripts, data, and instructions to reproduce the experiment are publicly available at https://github.com/VeriFixer/DafnyFix.

Listing 1: Example of an incorrect Dafny program fixed by applying a expression value replacement mutation operator.

```
1  method IsDifferenceOfSquares(n: int) returns (result: bool)
2  requires n >= 0
3  ensures result <==> exists i: int, j: int ::
4      0 <= j <= i <= n && n == i * i - j * j
5  {
6      result := false;
7      var i := 0;
8  -   while i * i <= n
9  +   while 1 * i <= n
10       invariant 0 <= i <= n + 1
11       invariant !result ==> forall k, y ::
12           0 <= y <= k < i ==> n != k * k - y * y
13       invariant result ==> exists k, y ::
14           0 <= y <= k < i && n == k * k - y * y
15       {
16           var j := 0;
17           while j <= i
18           invariant 0 <= j <= i + 1
19           invariant forall y :: 0 <= y < j ==> n != i * i - y * y
20           {
21               if n == i * i - j * j {
22                   result := true;
23                   break;
24               }
25               j := j + 1;
26           }
27           if result {
28               break;
29           }
30           i := i + 1;
31       }
32 }
```

While generic mutation operators can be useful for some types of implementation faults, others require more structured modifications informed by the conditions in which the fault manifests. The LLM-generated program in Listing 2, for example, which checks if an input array `a` contains duplicates, fails verification because Dafny cannot prove that the for loop's lower bound does not exceed its upper bound (a language requirement). In fact, since the program iterates from 0 to the second-to-last element of the array, empty arrays violate this constraint (0 > `a.Length` − 1, with `a.Length` being 0). Fixing the program involves ensuring the loop is not executed for empty arrays, which can be achieved with an early return. Our state-guided template-based heuristic generates this patch by identifying, through state-based FL, a suspicious program state at the method's entry in which `a.Length == 0` holds, and using this predicate as the branch condition to instantiate one of our adapted repair templates, namely an early default-value return template.

3 Repair Search Framework

DafnyFix combines a mutation-based heuristic, complemented by an inverse-mutation one, and a state-guided template-based heuristic to target different classes of small-scale implementation errors. This section describes these three components that define the candidate patch space (Sections 3.1 to 3.3), how fault localization guides their application, and our overall repair process (Section 3.4).

3.1 MutDafny's Mutation Operators

Our first patch generation heuristic involves applying small code transformations drawn from the mutation operators implemented in MutDafny [2]. Originally designed for assessing Dafny specifications' strength, in this work, we repurpose them as repair transformations to generate candidate patches for implementation faults.

Listing 2: Example of an incorrect Dafny program fixed by an early return for empty array edge cases.

```
method ContainsDuplicate(a: array<int>) returns (result: bool)
requires a != null
ensures result ==> exists i, j ::
    0 <= i < a.Length && 0 <= j < a.Length && i != j && a[i] == a[j]
ensures !result ==> forall i, j ::
    0 <= i < a.Length && 0 <= j < a.Length && i != j ==> a[i] != a[j]
{
+   if (a.Length == 0) {
+       return false;
+   }

    result := false;
    for i := 0 to a.Length - 1
    invariant 0 <= i <= a.Length - 1
    invariant !result ==> forall x, y ::
        0 <= x < i && 0 <= y < a.Length && x != y ==> a[x] != a[y]
    {
        for j := i + 1 to a.Length
        invariant i < j <= a.Length
        invariant !result ==> forall k :: i < k < j ==> a[i] != a[k]
        {
            if a[i] == a[j] {
                result := true;
                return;
            }
        }
    }
}
```

Employing mutation operators for repair may seem counterintuitive, as they are traditionally used to introduce faults. Nevertheless, most operators are either inherently reversible, i.e., they can undo the exact same faults they model, or reverse the behavior introduced by another operator. For example, the arithmetic operator replacement (AOR) replaces an arithmetic operator (+, -, *, /, %) with another, and, if replacing, for instance, a + with a - introduces a fault, then replacing a - with a + will restore the correct behavior. Both transformations correspond to valid applications of the AOR operator, showing that it is reversible. Another example is the arithmetic operator insertion (AOI), which inserts a unary minus in front of an expression, and the arithmetic operator deletion (AOD), which does the opposite. Though neither of these operators is reversible, they reverse each other, thus both possess repair capabilities. While not all mutation operators exhibit these properties, they may still prove useful for repairing other types of implementation faults. Rather than making assumptions about their effectiveness, we include all of them in the repair search space.

In this light, DafnyFix supports the complete set of 40 mutation operators implemented in MutDafny, which includes 30 previously proposed in mutation tools for other programming languages but that are also applicable to Dafny, and 10 additional ones resulting from an analysis of bugfix commits in Dafny GitHub repositories. The methodology for deriving the additional mutation operators provides evidence that they are representative of real errors made by Dafny developers, while the adapted operators model common faults in other programming languages and are widely adopted.

3.2 Reversed Mutation Operators

Mutation operators are traditionally designed to model the introduction of faults, and their reversing transformations naturally model the corresponding repairs. As discussed in Section 3.1, while most MutDafny operators already support the inverse (bug-fixing) behavior — either because they are capable of reversing themselves or reversing another mutation operator — a subset does not, motivating the introduction of explicitly defined reversed variants.

Table 1: Mutation operators admitting meaningful inverse transformations not included in MutDafny. Operators in the top section are implemented; the remaining are future work.

Op.	Description	Reversed behavior
BBR	Replacement of a relational or conditional expression with `true` or `false`	Replacement of a `true` or `false` expression with a relational or conditional one
EVR	Replacement of an expression with a default literal	Replacement of a literal with a non-literal expression
SDL	Deletion of a statement or of an entire code block	Insertion of a statement or of an entire code block
MRR	Replacement of a call with a default return-type literal	Replacement of a literal with a same-type return value call
MAP	Replacement of a call with one of its arguments with the same type as the return value	Replacement of an expression with a same-type return value call taking it as one of its arguments
MVR	Replacement of a call with a same-type variable	Replacement of a variable with a same-type return value call
MNR	Deletion of a class method call, its receiver being maintained	Replacement of a class object variable with a same-type return value call on it
CBE	Extraction of one of the blocks of an if statement and deletion of the remaining	Insertion of an if statement surrounding an existing statement

Among MutDafny's 40 mutation operators, Table 1 summarizes the ones admitting meaningful inverse transformations that are not already represented by existing MutDafny operators. Even though we identify eight operators for which reversed mutation operators can be implemented, in this preliminary work, we only implement reversed variants for MutDafny's boolean-binary expression replacement (BBR), expression value replacement (EVR), and statement deletion (SDL) operators, which we respectively refer to as *RevBBR*, *RevEVR*, and *RevSDL*. We select these in particular because, by targeting some of the most common programming constructs, they are the most broadly applicable operators [2], making them more likely to contribute substantially to the repair search space. Since our goal is to evaluate the effectiveness of the reversed mutation operator repair heuristic, we leave the remaining, more specialized inverse transformations for future work.

While the intuition behind the three reversed operators is straightforward and akin to other mutation operators, their implementation shares a common challenge: determining the program construct to introduce during the inverse transformation. Traditional mutation operators define very specific rules regarding what transformations are allowed. For example, AOR replaces any arithmetic operator in the program with either +, -, *, /, or %, and EVR replaces any numeric expression with either 0, 1, or −1. In contrast, reversing BBR and EVR, which generally target expressions, involves choosing an

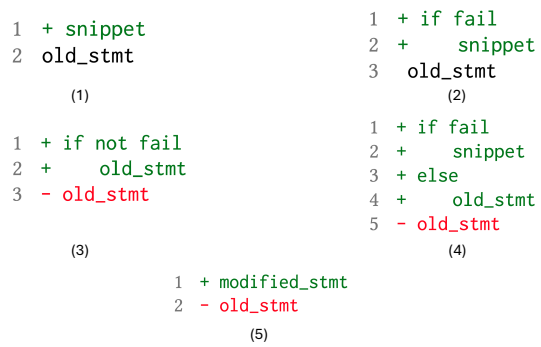


Figure 1: Implemented state-guided repair templates.

arbitrary expression of the appropriate type to perform the replacement, while reversing SDL, which generally targets statements, involves inserting an arbitrary statement, both with no further guidance regarding the construct to introduce. To address this, we follow GenProg's [16] strategy of borrowing program constructs already present elsewhere in the code, under the hypothesis that the code required to fix a program is likely already present in it.

3.3 State-guided Templates

Our third patch generation heuristic uses a template-based repair strategy guided by suspicious program states inspired by AutoFix [25]. In addition to AutoFix being the first APR technique tailored for verification-aware languages — directly employing contracts to identify suspicious program states — we decide to include template-based repair due to the need to target more complex errors requiring semantic, rather than purely syntactic, modifications.

This heuristic relies on a collection of repair templates parameterized using information from suspicious program states. Each suspicious state characterizes the program's faulty behavior as a boolean predicate, i.e., a property that either does or does not hold at a given program location. The program location of the target suspicious state marks where the repair template will be applied. The predicate and its value are directly used in its instantiation.

Figure 1 shows the five templates we implement. Each exposes a set of parameters that are instantiated during patch generation: `old_stmt`, common to all templates and denoting the original statement at the suspicious location, as well as `snippet`, `fail`, and `modified_stmt`, which are described below. Templates 1 to 4 are borrowed from AutoFix, with a small enhancement that we describe next, while template 5 is newly introduced in this work.

Templates 1 to 4 attempt to fix the program by applying transformations that avoid the suspicious state, changing the values of variables that appear in the suspicious predicate, which we term *state-changing target variables*. In AutoFix, `snippet` is instantiated as what we call a *state-changing assignment*, assigning a default value to a target variable. For example, for a state-changing integer target variable `var`, AutoFix considers the following assignments for `snippet`: {`var = 0`, `var = 1`, `var = -1`, `var = var + 1`, `var = var - 1`}, with similar defaults for other variable types. To simplify this process and further expand the set of candidate assignments, we start with a single default (`var = var + 1`, for integer target variables) and apply our mutation operators over it. Additionally, we extend the notion of `snippet` to also include early return statements, with the set of candidate return values being generated following the same process used for assignments. Given a suspicious program state, for each of its candidate snippets, we instantiate templates 1 to 4.

The `fail` parameter is a placeholder for the branch condition `p == v`, where `p` is the boolean predicate associated with the suspicious state and `v` its value at the suspicious location. These branching structures account for different repair applications, allowing the repair actions (`snippet`) to either be applied unconditionally (template 1), or only when the suspicious state manifests (templates 2 and 4), and, conversely, for the original behavior (`old_stmt`) to either be preserved unconditionally (templates 1 and 2), or only when the suspicious state does not manifest (templates 3 and 4).

Template 5 views suspicious states not as properties to avoid, but as evidence that a program expression may have been used in place of another. Given a suspicious state predicate `lhs` *op* `rhs`, where *op* is a relational operator, we treat both sides as alternative expressions that may have been confused by the programmer. Consequently, if either `lhs` or `rhs` appears in the suspicious statement (`old_stmt`), we produce `modified_stmt` by replacing that occurrence with its counterpart, i.e., an occurrence of `lhs` is replaced with `rhs`, and vice-versa. Exemplifying in the context of Listing 1, a suspicious program state `i * i != i` at Line 8 indicates that executions where `i * i` does not equal `i` produce incorrect behavior. Since `i * i` appears in the suspicious program line, the template generates a candidate repair by replacing it with `i`. Similarly, if `old_stmt` is an assignment and its left-hand side is either `lhs` or `rhs`, `modified_stmt` is produced by replacing the assignment's right-hand side with the opposite side of the suspicious predicate.

3.4 Search Space Exploration

To constrain our search space, we employ fault localization techniques proposed by Silva et al. [26] for Dafny, following the insights of their empirical study comparing several techniques. FL allows us to focus program transformations on locations more likely to contain faults, reducing the patch space compared to an exhaustive search of the program. Concretely, we adopt two techniques they present: a counterexample-based approach (CNTM) that evaluates suspicious program lines by the frequency in which they occur in counterexamples issued for failed verification attempts, and a state-based approach (SNAP) that evaluates suspicious program states according to boolean predicates that hold in failing vs. passing test executions. The state-based approach was adapted from AutoFix [25], which also inspired our state-guided repair heuristic.

We use the counterexample-based technique as our primary localization strategy because it achieved the strongest results in prior work [26]. As done by Debroy and Wong [4], lines are explored from most to least suspicious, independently applying each applicable mutation or reversed-mutation operator to each line, so that every candidate patch contains a single mutation.

While this FL technique prioritizes suspicious program locations, it lacks the suspicious program state information required by state-guided template-based patch generation. For this use case, Silva et al. [26]'s state-based FL technique is more appropriate. However, it was

reported to be significantly less effective than its counterexample-based counterpart. To preserve counterexample-based FL's high effectiveness while still making use of the state-ranking capability of state-based FL, we combine the two approaches to guide the instantiation of our state-guided templates. For each suspicious program line identified by counterexample-based FL, we pair it with the most suspicious states identified at that location by state-based FL, and instantiate a repair template for each resulting pair. Since the state-based technique optimizes a different notion of suspiciousness (program states rather than lines), the overall highest-ranked states, even if not corresponding to the highest-ranked program lines in counterexample-based FL, may still identify program states indicative of the fault in locations overlooked by line-based ranking. To avoid discarding these promising repair opportunities, we additionally instantiate repair templates using the overall highest-ranking suspicious states reported by the state-based technique.

It should be noted that, besides being less effective, the state-based technique is also reported to produce a high empty-ranking rate, largely due to its reliance on an early prototype test generation tool. Hence, for programs unsupported by the state-based technique, only mutation-based candidate patches will be generated.

To further control the search space size, since FL may identify many suspicious program lines and states, we limit the number of each considered for patch generation. More specifically, we initially consider only the top L suspicious lines, the top S suspicious states for each line, and the overall top S suspicious states. L and S are configurable. In our experiments, we used $L = 5$ and $S = 10$.

Building on this description of the process, we can formally define the candidate patch space. Let l be a program line considered for patch generation (either among the top L counterexample-ranked lines or associated with one of the overall top S states from state-based FL), S_l the set of program states considered for template instantiation at line l, M_l the set of mutation operators (regular and reversed) applicable at l, and $T_{l,s}$ the set of repair templates applicable at l using state s. For lines introduced solely by the overall state ranking, $M_l = \emptyset$, as only state-guided repair is applied there. The set of candidate patches CP generated for l is defined as follows:

$$CP(l) = \{P_m(l) \mid m \in M_l\} \ \cup \ \left\{P_t(l,s) \;\middle|\; s \in S_l,\ t \in T_{l,s}\right\}$$

where $P_m(l)$ denotes a candidate patch resulting from applying mutation operator m to l, and $P_t(l,s)$ denotes a candidate patch generated by instantiating repair template t with state s at line l.

The above definition describes the candidate patch space explored in a single search iteration for a single program line. The values of L and S define the *initial search space* of program lines and states to explore. Since we aim to characterize our framework's repair range, we exhaustively evaluate all candidate patches in this initial space, regardless of whether a successful patch has already been found. This may yield multiple successful patches for the same program. If no successful repair is found within this initial space, we *progressively expand* it based on fault localization rankings. We first consider additional program states beyond the initial S for the lines already explored, instantiating further repair templates at those locations. We then consider additional program lines beyond the initial L, following the counterexample-based ranking, and repeat the candidate generation process. During these expansions, the search terminates as soon as a successful repair is found.

4 Evaluation

We evaluate DafnyFix on a dataset of verification failures caused by implementation errors in LLM-generated Dafny programs.

4.1 Dataset Curation

Mainstream programming languages have long benefited from curated benchmarks for APR research, such as Defects4J [10] for Java and ManyBugs and IntroClass [15] for C/C++, providing collections of real bugs and their developer-written fixes. However, to our knowledge, no equivalent dataset of real-world buggy programs exists for Dafny or other verification-aware languages, making the evaluation of repair techniques particularly challenging.

On the other hand, several Dafny benchmarks, such as *dafny-synthesis* [21] and *VeriEquivBench* [29], have been proposed to evaluate LLMs' code generation capabilities. While these were not designed for APR, they contain large collections of LLM-generated programs together with their verification outcomes. We use the verification failures in these datasets as a proxy for an APR dataset, repurposing unverified LLM-generated programs as candidate buggy programs from which to construct our evaluation dataset.

Dataset source. We build our dataset on *dafny-synthesis* [21], which evaluates GPT-4 and PaLM-2 at synthesizing Dafny methods for 178 MBPP-derived programming tasks. We chose this over *VeriEquivBench* [29], as the latter's greater scale and complexity would make the manual inspection curation step (see below) impractical.

Candidate selection. From the complete set of LLM-generated programs, we retain only those that fail verification, and filter out the ones that do not parse or type-check by running Dafny's resolution (`dafny resolve`). This yields 60 programs that parse and type-check successfully but fail verification. A manual inspection of these 60 candidates discarded programs whose verification failures resulted solely from missing annotations, while retaining those containing genuine implementation faults. This initial screening resulted in 18 candidate programs to be repaired.

Specification strengthening. Since successful verification requires both a correct implementation and a properly annotated program, a patch can only fix a verification failure if the specification contains the necessary verification annotations. Furthermore, employing formal specifications as the repair oracle relies on their completeness to rule out vacuous or incorrect repairs. To isolate implementation faults from specification-related issues, each candidate was subject to a second, more in-depth inspection to manually strengthen weak pre- and post-conditions and add any missing loop invariants needed to enable successful verification of a correct implementation. This process resulted in the exclusion of two additional programs: one with a vacuous specification, which was logically always true, and another for which we could not establish sufficient verification annotations for the existing implementation.

The need for strengthening is illustrated by the different effects of incomplete specifications on DafnyFix's repair effectiveness. Weak post-conditions do not prevent it from generating the successful patches it would produce under a strong post-condition. However, they can also cause many patches to be incorrectly labeled as successful, due to a lack of constraints to distinguish correct behavior. For instance, omitting the second post-condition in Listing 2 would

Table 2: DafnyFix's repair effectiveness across our dataset.

Program	Model	# Successful Patches	# Generated Patches
task_id_19[4]	PaLM-2	8	1,274
task_id_68[3]	GPT-4	29	1,483
task_id_68[4]	PaLM-2	22	1,079
task_id_72[3]	GPT-4	2	141
task_id_72[4]	PaLM-2	0	14,245
task_id_249[2]	GPT-4	1	3,842
task_id_281[3]	PaLM-2	5	1,572
task_id_285[3]	GPT-4	0	65
task_id_304[4]	PaLM-2	1	697
task_id_397[4]	PaLM-2	0	2,118
task_id_443[3]	GPT-4	0	518
task_id_448[2]	GPT-4	0	20,268
task_id_622[4]	PaLM	5	247
task_id_748[3]	GPT-4	1	162
task_id_751[4]	PaLM-2	0	59,850
task_id_776[4]	PaLM-2	9	819

cause a patch that deletes its outer for loop (SDL operator) to be misclassified as successful. Missing loop invariants may make a fault impossible to repair through implementation changes alone, as they are necessary proof obligations to establish the post-conditions.

Our evaluation dataset resulting from this curation process comprises **16 programs**, summarized in the first two columns of Table 2.

4.2 Results

Our evaluation is guided by the following research questions:

- **RQ1:** How effective is DafnyFix at automatically repairing implementation faults in LLM-generated Dafny programs?
- **RQ2:** How much do the different patch generation heuristics contribute to the repair effectiveness of our framework?
- **RQ3:** What type of implementation faults in LLM-generated Dafny programs can be fixed by DafnyFix?

4.2.1 ***RQ1: Overall Repair Effectiveness.*** To assess DafnyFix's overall effectiveness in searching for successful repairs, we applied it to each program in our dataset. Table 2 presents the results, reporting, for each program, the number of successful patches found and the total number of candidate patches generated during the search. Successful repairs were generated for 10 of the 16 programs (62.5%), demonstrating our search strategy's ability to automatically repair a substantial number of the verification failures in the dataset.

The six remaining programs were left unrepaired for distinct reasons, highlighting some of the current limitations of our framework.

[2]https://github.com/Mondego/dafny-synthesis/blob/master/RQs/RQ2-%5BSignature-Prompting%5D/rq2-GPT-4-temp_0.75-verified-unverified-up.json, accessed Aug. 2026.
[3]https://github.com/Mondego/dafny-synthesis/blob/master/RQs/RQ3-%5BDynamic-Few-Shot-Prompting%5D/rq3-GPT-4-temp_0.5-verified-unverified-up.json, accessed Aug. 2026.
[4]https://github.com/Mondego/dafny-synthesis/blob/master/RQs/RQ3-%5BDynamic-Few-Shot-Prompting%5D/rq3-PaLM-2-temp_0.5-verified-unverified.json-up.json, accessed Aug. 2026.

Table 3: Breakdown of successful patches by the patch generation heuristic that generated them.

Program	Mut	Rev Mut	T1	T2	T3	T4	T5
task_id_19[4]				3	1	4	
task_id_68[3]				18	3	8	
task_id_68[4]				14	4	4	
task_id_72[3]	2x EVR						
task_id_249[2]		1x RevSDL					
task_id_281[4]				2	1	2	
task_id_304[4]							1
task_id_622[4]		5x RevSDL					
task_id_748[3]				1			
task_id_776[4]					1	8	

Both `task_id_397` and `task_id_448` require several program modifications to be fully repaired, and, as preliminary work, our framework only searches over the space of single-transformation patches. A more extreme case is observed in PaLM-2's `task_id_72` and `task_id_751`, whose implementations are fundamentally incorrect and would need to be fully rewritten, making them altogether unsuitable targets for APR. The fault present in `task_id_285` is similar to the one presented in Listing 2, showing potential for being successfully fixed via the state-guided templates. However, this program is among those mentioned in Section 3.4 for which the state-based FL technique does not produce a state ranking because the test generation tool does not support it. Finally, although `task_id_443` can be repaired through a single transformation, this transformation is not among those implemented in our framework.

The large number of candidate patches comes from a deliberate design choice motivated by the exploratory nature of this work. We aim to comprehensively evaluate the potential of the proposed heuristics to better understand which transformations are effective, redundant, and which faults they address (RQ2). This comes at the cost of high computational overhead. The entire pipeline takes a median of 1.3 and an average of 6.5 hours per program, with the latter heavily skewed by a single program (`task_id_751`) with substantially longer runtime than the others. Nevertheless, successful repairs are found much earlier, with a median time to the first repair of 18 minutes. Most of the runtime is spent verifying candidate patches, with state-based FL incurring additional overhead for some programs. The insights of this work can guide future efforts to design more targeted search strategies that substantially reduce the number of explored patches and execution times.

4.2.2 ***RQ2: Individual Heuristic Effectiveness.*** To understand the contribution of the different patch generation heuristics, we analyzed each successful patch generated for our evaluation dataset. Table 3 breaks down, for each program, the heuristics that generated the successful repairs. MutDafny's mutation operators are capable of repairing one program (the motivational example in Listing 1), the newly implemented reversed mutation operators repair another two programs, and the state-guided templates the remaining seven.

Notably, each program can only be repaired by one single heuristic, i.e., our dataset contains no faults fixable via both mutation and template instantiation, highlighting the complementary nature of

the proposed heuristics. Furthermore, although template instantiation repairs the most programs, five of the seven programs it repairs exhibit the same kind of fault (detailed in Section 4.2.3). This suggests that the heuristic's high repair rate is largely explained by the prevalence of this fault category in our dataset, rather than by it necessarily being more effective than the others.

A closer inspection of the successful patches provides further insights. All successful repairs resulting from the newly introduced reversed operators use RevSDL, either inserting a break statement or an assignment that repairs an uninitialized variable fault. Regarding state-guided templates, template 1 is never successful, while templates 2-4 generate almost all repairs, often producing equivalent patches for the same program. Template 5 was successful in only one program, where the specification directly encodes its solution. This relationship is reflected in a suspicious state predicate, allowing template 5 to generate a correct repair by an assignment to the expected return value. While this single success does not provide sufficient evidence of template 5's effectiveness, our early return extension to the adapted templates shows impact, accounting for most successful patches generated by templates 2 and 4.

4.2.3 ***RQ3: Fault Characterization.*** Beyond evaluating repair effectiveness, we analyze the implementation errors present in our dataset and their repairability by DafnyFix. This provides insight into the kinds of faults commonly introduced by LLMs and helps explain the repair behavior observed in RQ1 and RQ2. We identify the six distinct categories of implementation faults discussed below.

For-loop boundaries. This is the most common type of fault found in our dataset, appearing in seven programs. In all of them, the generated implementation omits an edge case, causing the lower bound of a for-loop to exceed its upper bound for certain inputs and preventing verification. This pattern is illustrated in the motivational example in Listing 2. As discussed in Section 4.2.2, the state-guided templates effectively address this fault pattern, successfully repairing five of the affected programs. The remaining two fall outside the current capabilities of our framework, as discussed in Section 4.2.1. Of the seven instances, four involve array-length edge cases (`task_id_19`, GPT-4's `task_id_68`, PaLM-2's `task_id_68`, and `task_id_281`), two involve string-length edge cases (`task_id_285` and `task_id_776`), and one involves several integer edge cases (`task_id_448`).

Uninitialized variable. This fault appears in two programs, being caused by a method output variable used in loop invariants before initialization. This is addressed by inserting an initialization statement before any usages, which, in one program, is achieved by the RevSDL operator through an assignment to the uninitialized variable appearing later in the program (`task_id_622`). This repair is not possible in the other program, as it lacks a default initialization statement that can be borrowed (`task_id_443`).

Wrong conditional expression. In another two programs, the fault consists of one or more incorrect conditional expressions. The first consists of the example in Listing 1 (GPT-4's `task_id_72`), fixed by mutation of the while loop guard, whereas the second (`task_id_397`) contains several incomplete if conditions that omit valid execution cases, which are not repaired by our framework since the correction requires changing multiple conditions.

Out of bounds index access. This fault occurs in one program (`task_id_748`) where the first character of a potentially empty string is accessed, preventing the verifier from proving the index access's safety. As with the for-loop boundary faults, this is repaired via template instantiation by checking for the empty string case.

Missing break statement. One program (`task_id_249`) fails due to the omission of a break statement after successfully handling the desired case in a loop. The RevSDL operator successfully repairs the fault by inserting the missing break statement.

Fundamentally incorrect implementation. This category comprises three programs with fundamentally incorrect implementations, consisting of arbitrary computations that do not contribute to satisfying the specification (PaLM-2's `task_id_72`, `task_id_304`, and `task_id_751`). While these are not localized implementation faults, they are important for characterizing the LLM code generation failures in our dataset. Interestingly, one of these programs is successfully repaired by template 5 (`task_id_304`). Because its specification explicitly encodes the program's intended result, as discussed in Section 4.2.2, DafnyFix can synthesize a correct assignment for the return value despite the incorrect implementation.

5 Related Work

We focus on deterministic APR techniques, which are the subject of the work presented in this paper.

Search-based repair explores a space of code modifications, iteratively selecting, generating, and validating candidates against an oracle. GenProg [16] proposes a genetic algorithm to evolve a population of candidate patches generated through *insert*, *delete*, and *replace* code edits, with *insert* and *replace* reusing existing code elements of the program. Rather than exploring the repair space at random, later techniques introduce mechanisms to guide the search toward more promising candidates [9, 14, 27]. Another family of techniques only applies predefined sets of repair patterns (templates) curated from observed human-written patches [11, 12].

Debroy and Wong [4] employ mutation operators for repair, rather than solely for testing, an idea we extend to Dafny. MUTAPR [3] deals with mutations of binary operators, applying GenProg's algorithm to study if this type of mutation can broaden the range of fault types beyond those requiring the presence of the correct code. Akin to our work, HDRepair [14] applies transformations from multiple sources, including mutation operators.

AutoFix [25], for Eiffel, is the first APR tool employing contracts, along with tests generated from them. It follows a state-based approach, where program states likely to be associated with a contract violation are derived from both static program analysis and passing and failing test executions. Candidate repairs are generated by modifying the program to prevent it reaching the identified faulty states. Specifically for Dafny, to our knowledge, repair has only been attempted using LLMs [28] and focusing on arithmetic errors.

Constraint-based APR models repair as a constraint-solving problem. SemFix [23] and Angelix [20] use symbolic execution with test cases to derive constraints that the suspicious statement must satisfy for all tests to pass. Component-based synthesis [8] then generates an alternative statement by encoding expected program semantics into an SMT-solvable constraint. ExtractFix [5] follows

this approach, but extracts repair constraints from program assertions, documentation, or properties enforced by dynamic analysis.

6 Conclusion

We presented DafnyFix, a search-based repair framework that combines three patch generation heuristics: MutDafny mutation operators, some of their reversed variants, and state-guided repair templates inspired by AutoFix. These provide complementary candidate transformations, while existing fault localization techniques for Dafny constrain the search space. DafnyFix repairs 10 out of 16 (62.5%) implementation faults in a curated dataset of LLM-generated Dafny programs. Our evaluation shows that the three patch generation heuristics target distinct implementation faults without overlap, collectively addressing a substantial portion of the faults in our dataset through small, deterministic program transformations, including those from weaker models such as PaLM-2.

Despite promising results, our framework has limitations. Its restriction to single transformations proved to be insufficient for some programs in our dataset. Also, because of its dependence on pre-existing verification annotations, particularly invariants, it is not applicable to improperly annotated programs, and the reliance on a test generation tool excludes some programs that could benefit from template-based repair. Lastly, the large search space leads to substantial runtimes, limiting DafnyFix's scalability. Future work will address these limitations by supporting multiple transformations, improving fault localization robustness, and prioritizing more promising patches to reduce computational costs. A valuable alternative for state-based FL would be to derive template instantiations directly from verifier feedback, and an operator ranking based on a more detailed analysis of their repair capabilities could improve search efficiency. Future work will also compare DafnyFix with an LLM baseline to assess relative strengths and weaknesses.

Overall, our results demonstrate the potential of search-based repair for Dafny and provide a foundation for the development of more scalable and broadly applicable techniques.

Acknowledgments

This work was 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 (DOI: 10.54499/2023.15557.PEX).

References

[1] Alexandre Abreu, Nuno Macedo, and Alexandra Mendes. 2023. Exploring Automatic Specification Repair in Dafny Programs. In *2023 38th IEEE/ACM Int. Conference on Automated Software Engineering Workshops (ASEW)*. 105–112.

[2] 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)*.

[3] Fatmah Yousef Assiri and James M. Bieman. 2014. An Assessment of the Quality of Automated Program Operator Repair. In *Proceedings of the 2014 IEEE Int. Conf. on Software Testing, Verification, and Validation (ICST '14)*. IEEE Computer Society.

[4] Vidroha Debroy and W. Eric Wong. 2010. Using Mutation to Automatically Suggest Fixes for Faulty Programs. In *2010 Third International Conference on Software Testing, Verification and Validation (ICST '10)*. IEEE Computer Society.

[5] Xiang Gao, Bo Wang, Gregory J. Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. 2021. Beyond Tests: Program Vulnerability Repair via Crash Constraint Extraction. *ACM Trans. Softw. Eng. Methodol.* 30, 2 (2021).

[6] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* 45, 1 (2019), 34–67.

[7] Kai Huang, Zhengzi Xu, Su Yang, Hongyu Sun, Xuejun Li, Zheng Yan, and Yuqing Zhang. 2024. Evolving Paradigms in Automated Program Repair: Taxonomy, Challenges, and Opportunities. *ACM Comput. Surv.* 57, 2 (2024).

[8] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 1. 215–224.

[9] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *27th ACM SIGSOFT Int. Symposium on Software Testing and Analysis (ISSTA 2018)*.

[10] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *2014 Int. Symposium on Software Testing and Analysis (ISSTA 2014)*. ACM, 437–440.

[11] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*. 802–811.

[12] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. FixMiner: Mining relevant fix patterns for automated program repair. 25, 3 (2020), 1980–2024.

[13] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. 2024. Verus: A Practical Foundation for Systems Verification. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24)*. ACM.

[14] Xuan Bach D. Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *Int. Conf. on Software Analysis, Evolution, and Reengineering (SANER)*.

[15] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering* 41, 12 (2015), 1236–1256.

[16] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72.

[17] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65.

[18] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer Berlin Heidelberg, 348–370.

[19] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In *Advances in Neural Information Processing Systems*, Vol. 36. Curran Associates, Inc., 21558–21572.

[20] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*.

[21] Md Rakib Hossain Misu, Cristina V. Lopes, Iris Ma, and James Noble. 2024. Towards AI-Assisted Synthesis of Verified Dafny Methods. *Proc. ACM Softw. Eng.* 1, FSE (2024).

[22] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *ACM Comput. Surv.* 51, 1 (2018).

[23] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. 772–781.

[24] 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)*. IEEE, 203–214.

[25] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. 2014. Automated Fixing of Programs with Contracts. *IEEE Transactions on Software Engineering* 40 (2014), 427–449.

[26] Álvaro Silva, Isabel Amaral, João Pascoal Faria, and Alexandra Mendes. 2026. Improving Debugging in Verification-Aware Languages Through Automated Fault Localization: A Case Study in Dafny. In *2026 IEEE 37th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. doi:10.48550/arXiv.2608.05399

[27] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*.

[28] Valentina Wu, Alexandra Mendes, and Alexandre Abreu. 2025. Specification-Guided Repair of Arithmetic Errors in Dafny Programs Using LLMs. In *Software Engineering and Formal Methods: 23rd International Conference, SEFM 2025, Toledo, Spain, November 10–14, 2025, Proceedings*. Springer-Verlag, 261–278. doi:10.1007/978-3-032-10444-1_16

[29] Lingfei Zeng, Fengdi Che, Xuhan Huang, Fei Ye, Xu Xu, Binhang Yuan, and Jie Fu. 2026. VeriEquivBench: An Equivalence Score for Ground-Truth-Free Evaluation of Formally Verifiable Code. In *Int. Conf. on Learning Representations*, Vol. 2026.