MutDafny: A Mutation-Based Approach to Assess Dafny Specifications

Isabel Amaral
INESC TEC, Faculdade de Engenharia, Universidade do Porto
Porto, Portugal
isabel.andre.amaral@gmail.com

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

José Campos
Faculdade de Engenharia, Universidade do Porto & LASIGE, Universidade de Lisboa
Porto, Portugal
jcmc@fe.up.pt

Abstract

In verification-aware languages, such as Dafny, despite their critical role, specifications are as prone to error as implementations. Flaws in specifications can result in formally verified programs that deviate from the intended behavior. In this paper, we explore the use of mutation testing to reveal weaknesses in formal specifications written in Dafny.

We present MutDafny, a tool that increases the reliability of Dafny specifications by automatically signaling potential weaknesses. Using a mutation testing approach, we introduce faults (*mutations*) into the code and rely on formal specifications for detecting them. If a program with a mutant verifies, this may indicate a weakness in the specification. We extensively analyze mutation operators from popular tools, identifying the ones applicable to Dafny. In addition, we synthesize new operators tailored for the language from bugfix commits in publicly available Dafny projects on GitHub. Drawing from both, we equipped our tool with a total of 40 mutation operators. We evaluate MutDafny's effectiveness and efficiency on a dataset of 794 real-world Dafny programs, and manually analyze a subset of the resulting undetected mutants, identifying five weak real-world specifications (on average, one at every 241 lines of code) that would benefit from strengthening.

1 Introduction

Proving that a software program is correct is necessary in critical systems, as failures in these can have drastic consequences. It is thus important to guarantee that such systems have no faults. Although traditional software testing [46] is used in many contexts for software validation, it only checks a system against specific inputs, not guaranteeing full correctness. In contrast, *formal software verification* [11, 30, 36] can mathematically prove complete correctness concerning a precise specification of the intended behavior.

Verification-aware languages, such as Dafny [35], are designed to support the development of formally verified software by embedding formal specifications directly into the code. It allows developers to specify program behavior through pre-conditions (*requires*), post-conditions (*ensures*), and invariants (*invariant*), which are automatically verified by Dafny, helping to guarantee that the implementation conforms to its intended behavior and ensuring code correctness beyond conventional testing. Dafny has seen adoption in industry, including at AWS[1] [15] and Consensys [14].

[1] https://github.com/aws/aws-encryption-sdk-dafny, accessed Jan. 2026.

Since formal verification only guarantees that an implementation conforms to its specification, a flawed specification can lead to a verified program that does not exhibit the intended behavior. Although specifications are usually trusted to be correct, this should not be taken for granted[2]. A study conducted by Fonseca et al. [21] confirmed exactly that: it found 16 faults in three formally verified distributed systems (one written in Dafny). Some of these faults stemmed from incorrect and incomplete specifications, highlighting a critical limitation of verification when specifications themselves are flawed.

As a motivational example, consider Dafny's code in Listing 1 returning all elements belonging to both input arrays.

Listing 1: Example of a Dafny program with a weak specification.

```
1  predicate InArray(a: array<int>, x: int)
2  reads a
3  { exists i :: 0 <= i < a.Length && a[i] == x }
4
5  method SharedElements(a: array<int>, b: array<int>)
6      returns (result: seq<int>)
7  ensures forall x :: x in result ==> (InArray(a, x) && InArray(b, x))
8  ensures forall i, j :: 0 <= i < j < result ==> result[i] != result[j]
9  {
10     var res: seq<int> := [];
11     for i := 0 to a.Length
12     invariant 0 <= i <= a.Length
13     invariant forall x :: x in res ==> InArray(a, x) && InArray(b, x)
14     invariant forall i, j :: 0 <= i < j < res ==> res[i] != res[j]
15     {
16 -       if InArray(b, a[i]) && a[i] !in res {
17 -           res := res + [a[i]];
18 -       }
19     }
20     result := res;
21 }
```

Although this program verifies, removing the `if` block in Lines 16 to 18 (an error a human could make or a mutation tool could insert) causes it to always return an empty list — an incorrect behavior that still satisfies the specification. Despite the inserted error, Dafny's ability to verify this program hints at a possible problem with the spec. This is indeed an example of a *weak* specification, i.e., one that does not sufficiently constrain the program's behavior. Its strengthening would require replacing the `==>` in the first `ensures` (Line 7) with `<==>` and the addition of an invariant (between Lines 11 and 15) adding that, besides every element in `result` being in both arrays, every element that appears in both arrays must also be present in `result`, i.e.:

```
+ invariant forall j :: 0 <= j < i && InArray(b, a[j]) ==> a[j] in res
```

[2] In this paper, we use *specification* or *spec* to refer to Dafny's formal contracts (e.g., requires, ensures, and invariants); we use *implementation* or *code* to refer to Dafny's code within method or function bodies.

In this work, our main goal is to support Dafny developers in ensuring the strength of their specifications. To achieve this, we follow the *mutation testing* [23, 25, 49, 54] concept, where the quality of a test suite is evaluated by introducing faults, called *mutants*, into a program, and evaluating whether these are detected by the test suite.

Mutants are automatically created by applying *mutation operators*, i.e., syntactic changes that mimic common programming errors, each differing from the original program by a small localized change. Mutants that cause tests to fail are considered *killed*, while those that go undetected are *alive*, indicating untested behavior. While the mutants typically run against a test suite, in our approach, they run against the formal specification. Surviving mutants may reveal weaknesses in the spec, indicating underspecified behavior, which may help developers identify and strengthen those specs. Nevertheless, an alive mutant may not always indicate the presence of a weak specification; i.e., the mutant might just be equivalent to the original program [13, 43, 55], meaning that, despite the change, its behavior remains the same as that of the original program.

This raises the questions: **(RQ1)** *Which small syntactic changes could mimic programming errors in Dafny?* and **(RQ2)** *Are mutation operators for Dafny programs helpful in the identification of specification weaknesses?* To answer these, we first assembled a compilation of 147 mutation operators previously proposed for other six programming languages and studied their application in Dafny. Secondly, we complemented this set with new operators tailored for the language, derived from an analysis of 1,475 bugfix commits from Dafny programs available in 112 public GitHub repositories. Thirdly, since, to our knowledge, there is no existing tool that mutates Dafny implementations, we developed one, **MutDafny**, supporting these mutation operators. Finally, we conducted an empirical study on 118,458 pairs of original-mutant Dafny programs to assess the effectiveness of our approach and, additionally, answer the questions: **(RQ3)** *How effective are the different mutation operators?* and **(RQ4)** *How efficient is MutDafny at generating mutants?*

The main **contributions** of this paper are:

- ★ An extensive analysis of which mutation operators supported by the most popular mutation tools can be applied to Dafny programs, and why. (Section 3)
- ★ The proposal of a set of novel mutation operators tailored for Dafny programs. (Section 4)
- ★ MutDafny: a novel mutation testing tool for signaling possible specification weaknesses in Dafny programs. The tool is publicly available at https://github.com/MutDafny/mutdafny. (Section 5)
- ★ Evidence of MutDafny's usefulness in assisting Dafny developers increase the reliability of their specs, gathered through an empirical study on 794 Dafny programs and a manual evaluation of 284 alive mutants. All the scripts, data, and instructions required to reproduce the study are publicly available at https://github.com/MutDafny/mutdafny-study-data. (Section 6)

Even though this paper explores mutation testing in the context of Dafny, an added benefit is that this approach is applicable to any verification-aware language.

2 Related Work

Mutations in verification-aware languages. To our knowledge, IronSpec[3] [22] is the only previous work dealing with mutations in Dafny, being the work more closely related to ours. It aims to increase the reliability of formal specifications by operating on three different dimensions, one involving mutation testing, but unlike our approach, mutating specifications. Their technique considers mutants that are stronger than the original specification. If the method still verifies with the mutation, it means that the original specification is weaker than the implementation and can potentially be strengthened. In 14 Dafny methods, IronSpec detected two specification faults through this approach.

Much like in MutDafny, mutations to the implementation of Eiffel programs have been used to evaluate their specs [34]. However, no developer-oriented tool based on the mutation of implementations exists for providing specification quality feedback, nor has any in-depth study been conducted on suitable mutation operators.

Other related work in using mutation testing to assess specification quality includes Lahiri [33] and Endres et al. [20]. Lahiri [33] uses the mutation of test cases to validate specifications without the need for an implementation. Endres et al. [20] capture a specification's ability to reject LLM-generated implementation mutants.

Mutation in other specification-related contexts. MuAlloy [52] is a mutation testing and mutation-based test generation tool for Alloy. Although both Alloy and Dafny support formal methods, the former is a specification modeling language with a distinct purpose from verification-aware ones. Ma'ayan et al. [42] used mutations consisting of the deletion of specification guarantees (weakening) to validate a coverage-based scenario generation algorithm for systems generated by reactive synthesis. Finally, in a mutation-based study on the effectiveness of the JML-JUnit [16] tool for automatic unit test generation from Java specifications, Tan and Edwards [50] concluded that mutation testing can also help unveil specification bugs, not just implementation errors.

Synthesis of mutation operators. Unlike our manual approach, Tufano et al. [51] used deep learning trained on 787,178 bug-fixing commits from GitHub to automatically generate mutants closely resembling real faults. However, replicating their approach is difficult due to the need for large-scale bugfix data and the fact that we only identified 1,475 (*likely*) bug-fixing commits (0.19% of their dataset).

3 Mutation tools and operators proposed for traditional programming languages

Aiming to answer **RQ1**, in this section, we compile the mutation operators proposed for other, more traditional, programming languages and investigate their applicability to Dafny programs. We consider a mutation operator applicable to a Dafny program if (a) the Dafny programming language supports the operation performed by the operator, and (b) if it has the potential of leading to a valid program.

[3] https://github.com/GLaDOS-Michigan/IronSpec, accessed Jan. 2026.

3.1 Mutation tools proposed for other programming languages but Dafny

Sánchez et al. [48] conducted a study of the use of mutation testing in practice and identified 127 mutation testing tools. The most popular tools for the programming languages for which Dafny compiles to (C#, C++, Java, JavaScript, Go, and Python) are: Stryker.NET [9] for C#; Mull [5, 18] for C++; Major [3, 27, 28], MuJava [4, 38–41], and PIT [7, 17] for Java; StrykerJS [8] for JavaScript; Go-Mutesting [1, 2] for Go; Mutmut [6] and MutPy [19] for Python. These were selected based on the overall top 10 most popular tools and on the most popular for each programming language.

3.2 Mutation operators proposed for other programming languages but Dafny

We identified the set of mutation operators supported by the tools listed in Section 3.1. This involved reading the papers (when available), documentation, and, for Mutmut [6], studying its source code due to a lack of documentation.

We found many similar operators across the different tools and languages, but most use different names. We grouped the operators according to our perceived similarity between them: *Operator replacement*, which apply to binary operators: arithmetic (e.g., +, ∗), relational (e.g., ==, <=), conditional (e.g., &&, ||), logical (e.g., &, |), shift (e.g., <<, >>>), assignment (e.g., + =, >>>=); *Operator insertion/deletion*, which apply to unary operators: arithmetic (e.g., −, ++), conditional (e.g., !), logical (e.g., ~); *Literal value replacement*; *Expression replacement*; *Statement, variable, constant, or operator deletion* — where the latter differs from unary operator deletion in that it applies to all types of operators, unary and binary, and involves the deletion of all occurrences of a specific operator; *Object-Oriented (OO) specific*: Inheritance, Polymorphism, and Overloading. The operators listed for the arithmetic, relational, conditional, logical, shift, and assignment groups are those specific to Java.

3.3 Mutation operators proposed for other languages that are applicable in Dafny

Following the process detailed in Section 3.2, we compiled a total of 147 mutation operators[4]. Of these, we identified 97 that can be applied to Dafny programs and 50 that cannot due to the syntax structure of Dafny 4.10.0.[5] From the 97 operators that can be applied to Dafny, we concluded that seven always generate equivalent mutants when applied to Dafny. Finally, the 90 operators were narrowed down to 30 unique ones, as many reflected the same behavior. Next, we partially answer **RQ1** by listing and describing the applicability of each operator and any usage restrictions.

3.3.1 Arithmetic operators. Dafny provides five arithmetic binary (+, −, ∗, /, %) and a single unary operator (−). Shortcut arithmetic operators (++, −−), although common in other programming languages, are not available in Dafny.

Arithmetic Operator Replacement (AOR) [1, 6, 8, 9, 17–19, 27, 28, 39] replaces an arithmetic binary operator by another. However, Dafny does not allow the out-of-the-box replacement of an arithmetic operator with / or %, as the verifier requires a pre-condition stating that the divisor is not zero. One approach adopted by some tools consists of restricting which operators can be replaced by which. In Dafny, this reflects in allowing free replacement between the +, −, and ∗ group of operators, and only allowing / to be replaced with % and vice-versa. Replacement of unary arithmetic operators cannot be applied in Dafny, as it only supports one unary operator (−).

Arithmetic Operator Insertion (AOI) [17, 39] and *Arithmetic Operator Deletion (AOD)* [17, 19, 39] can only apply to unary operators, namely, to Dafny's unary minus (−).

3.3.2 Relational Operator Replacement (ROR) [1, 2, 6, 8, 9, 17–19, 27, 28, 39]. Dafny supports the six relational operators common to most languages (==, ! =, <, <=, >, >=) and their replacement can be applied without any restriction. Additionally, the replacement of expressions using relational operators with `true` or `false` is also commonly implemented and possible in Dafny. Henceforth, we denote this as *Boolean-Binary Expression Replacement (BBR)* [39].

3.3.3 Conditional operators. Dafny supports three conditional operators: two binary (&&, ||) and one unary (!).

Conditional Operator Replacement (COR) [1, 2, 6, 8, 9, 18, 19, 27, 39] replaces a binary conditional operator with an alternative. As in *ROR*, many tools replace conditionals with `true` or `false`, another use of *BBR* [1, 2, 6, 8, 9, 17, 27].

Conditional Operator Insertion (COI) [6, 8, 9, 19, 39] and *Conditional Operator Deletion (COD)* [6, 19, 39] can be both applied to Dafny's unary conditional operator (!).

3.3.4 Logical operators. Dafny's binary logical bitwise operators are: &, |, and ^, and its single unary operator is !. While ! can be both conditional and logical, it is used in different contexts, either for boolean expressions or for bit-vector ones, returning different types accordingly.

Logical Operator Replacement (LOR) [1, 6, 9, 17, 19, 27, 28, 39] can be applied to Dafny's binary logical operators without restrictions, replacing them interchangeably. It cannot be applied to the unary operator as there is only one.

Logical Operator Insertion (LOI) [39] and *Logical Operator Deletion (LOD)* [6, 19, 39] can be applied to Dafny without restrictions, i.e., any bit-vector expression in Dafny can be modified by inserting or removing the ! operator.

3.3.5 Shift Operator Replacement (SOR) [1, 6, 9, 27, 28, 39]. Like *logical operators*, shift ones operate over bit-vector types. Dafny supports the binary shift operators << and >>. Since there are no unary shift operators, insertion or deletion cannot be applied.

3.3.6 Literal Value Replacement (LVR) [1, 6, 8, 9, 17–19, 27, 28]. This operator is traditionally applied to boolean, string, and numerical literals, with modifications that vary significantly between tools. The simplest case is the boolean: the only possible replacement is the logical complement, i.e., `true` with `false` and vice-versa. For string literals, common

[4]The list of mutation operators, including their description, which tools implement them, and their applicability to Dafny, can be found in the *supplementary material* at https://doi.org/10.6084/m9.figshare.30640202.v2.
[5]At the time of writing, the latest version of Dafny available was 4.10.0.

transformations include replacing non-empty with empty ones, default values, or inserting random characters in any position. Numerical literals are often incremented, decremented, negated, or replaced by constants, e.g., 0, 1, and −1. *LVR* can be applied to Dafny without restriction.

3.3.7 Expression Value Replacement (EVR) [3, 6, 17]. Corresponds to the replacement of primitive-typed expressions (e.g., numerical, boolean) with literal values. Given the broad definition of what constitutes an expression in a program, mutation tools implement a wide range of expression value replacement operators. Some target any primitive-typed expression throughout the code, while others target specific contexts (e.g., return values) and perform more specific replacements (e.g., `true`, `false`, and `null` returns).

Method call replacement. Some tools replace method calls with a default value matching their return type, applying only to non-void methods. In Dafny, methods may return multiple values, meaning the replacement may include several defaults. This distinction from *EVR*, makes it a distinct operator, *Method Return Value Replacement (MRR)* [17, 18].

A more specific method call replacement operator is the *Method Argument Propagation (MAP)* [7], which replaces a method call with one of its arguments. This must match the method's return type to preserve program validity.

Object initialization replacement. An example is the *Collection Initialization Replacement (CIR)* [8, 9] operator, which handles collection initialization with type-specific default constructors. Stryker.NET implements this for C# arrays, lists, collections, and dictionaries. Although these collection types have different designations in Dafny, this concept extends to them, i.e., arrays, (multi)sets, sequences, and maps. Additionally, in class instances, initialization through a constructor call can be replaced with `null` [17].

3.3.8 Loop constructs replacement. This operator targets loop-specific constructs, replacing `break` with `continue` and vice-versa. Replacing both of these with a `return` is also valid when the enclosing method is `void`. We refer to this as the *Loop Statement Replacement (LSR)* [1, 6, 19]. *Loop Break Insertion (LBI)* [1], inserts a `break` at the loop's entry.

It is possible to apply these operators to Dafny programs, but their effectiveness often depends on the structure of the loop. For instance, certain mutations (e.g., inserting a `continue` before a counter update) may prevent Dafny from verifying loop termination. These can easily cause verification to fail simply due to the application context.

3.3.9 Case Block Replacement (CBR) [7]. This operator replaces the bodies of `case` blocks in switch statements: the default case is replaced with the body of the first non-default label, and the remaining cases are replaced with the default body. Dafny's analogous construct is the `match` statement, where the default case is denoted by an underscore (_).

3.3.10 Statement Deletion (SDL) [1, 2, 9, 27, 39]. Like *EVR*, statement deletion operators can be categorized by context. The most general form deletes any statement type (e.g., `return`, `break`, assignments, calls, etc.) without targeting a specific category. While applicable to Dafny, such deletions may lead to invalid programs (e.g., as in other languages, if a variable's initialization is removed but later used).

Block deletion. An extreme form of statement deletion is removing entire code blocks. In Dafny, this includes deleting method bodies [8, 9], entire `if`, `else if`, and `else` branches, or `case` blocks in match statements [1, 2]. Method bodies can only be deleted for `void` methods, and `if` branches only when no alternatives (i.e., `else`) are present.

Method call deletion [7, 18]. Calls to void methods can be deleted without invalidating the program. Deleting non-void calls is not supported (in existing work or in MutDafny), as their return values are typically used elsewhere in the program (e.g., in assignments or expressions), and removing them would result in a syntactically incorrect program. Such cases are better handled by the *MRR* operator.

The *Method Naked Receiver (MNR)* [7] operator is a specific type of deletion operator for methods that are members of a class. It deletes the method call while preserving the receiver object. To avoid type errors, the receiver must have the same type as the method's return value. Moreover, the operator can only be applied to non-void methods in Dafny.

Field initialization. Finally, there are two operators that initialize class fields with arbitrary values through deletion.

The first targets fields initialized at the declaration [7]. Dafny does not provide specific default values for each type but ensures that variables of certain types, called *auto-initializable*, always hold legal values. While it ensures these values are legal, it does not define fixed defaults for each type, assigning arbitrary values. This feature does not work for user-defined classes, as they are not auto-initializable, and only works for constant fields, as Dafny allows only constants to be initialized at declaration.

The second, implemented by MuJava [38, 40], deletes a class's constructor, since Java automatically generates a default constructor that initializes fields with default values. Dafny also supports automatic constructor creation, but these are invoked without parentheses, e.g., `new Car` instead of `new Car()`. Applying this operator to a Dafny program would invalidate it, as constructor calls with parentheses would not be valid. An alternative is to delete the constructor body, allowing fields to be initialized with arbitrary values.

3.3.11 Variable Deletion (VDL) [39]. This operator deletes all uses of a variable. If the variable is used in a binary expression, the corresponding operator must also be deleted.

Similarly, Python supports slicing as `s[start:end:step]`, and MutPy [19] includes an operator that deletes one of the slice elements. Dafny provides a similar slicing syntax for sequences, supporting `start` and `end` but not `step`. Since this operator removes only part of a slice, unlike *VDL*, we refer to it as *Subsequence Limit Deletion (SLD)*.

3.3.12 Operator Deletion (ODL). Much like the *VDL* operator, *ODL* deletes all occurrences of a specific operator (e.g., arithmetic, relational). For binary operators, program validity entails deleting one of the operands as well.

3.3.13 Object-Oriented operators. MuJava [38, 40] was the first tool to propose a comprehensive set of mutation operators targeting OO features like inheritance, polymorphism,

and encapsulation. Some of these can be adapted to Dafny, which also supports OO features.

Polymorphic Reference Replacement (PRV) replaces an assignment to a parent class object with a different child class instance. It applies when the parent has multiple children and at least two variables of different child types are in scope. This may cause side effects if subclasses implement the same methods or initialize fields differently. In the following example, each subclass sets a different value for `numSides`, so the mutation leads the program to output an incorrect value.

```
    var shape: Shape;
    var rectangle := new Rectangle(10.0, 20.0);
    var triangle := new Triangle();
-   shape := rectangle; // numSides = 4
+   shape := triangle; // numSides = 3
    print shape.numSides, "\n";
```

This Keyword Insertion (THI) and *This Keyword Deletion (THD).* These two operators manipulate the `this` keyword, one by inserting, and the other by deleting it. These apply only when a class method has a parameter with the same name as a class field. Neither can be used on the Left-Hand Side (LHS) of an assignment because Dafny does not allow assignments to parameters: inserting `this` would require the LHS to originally be a parameter, which is not possible, and deleting it would leave a parameter on the LHS, invalidating the program.

Accessor Method Replacement (AMR) and *Modifier Method Replacement (MMR)* replace one method with another having the same signature. The first targets accessor methods, typically starting with `get`, which access class fields. The second targets modifier methods, usually starting with `set`, and which update an object's state.

3.4 Mutation operators proposed for other languages excluded for Dafny

Most operators from other languages that cannot be applied to Dafny target syntax constructs that do not exist in the language, such as shortcut operators, compound assignments (e.g., +=), switch statements, exception-catching blocks, and method overloading. Others rely on language-specific elements like Python's `None`, Java's `BigInteger`, or libraries like C#'s Language Integrated Query (LINQ). Some constructs exist in Dafny, but the resulting mutants are invalid or equivalent due to language constraints. Below, we highlight operators excluded for more complex reasons.

3.4.1 Operator replacement operators. Major's *Operator Replacement Unary (ORU)* [27, 28] cannot be applied to Dafny, as it replaces unary operators that do not necessarily belong to the same group, e.g., Java's − and ~, which both take numerical arguments. The unary operators supported in Dafny are −, for numeric types, and !, for booleans or bit-vectors, and they cannot replace each other due to the different argument type requirements.

3.4.2 Expression replacement operators. Stryker.NET's *Initialization* [9] operator replaces a parameterized constructor with an empty one. Since Dafny only allows one constructor per class, this operator cannot be applied to it.

StrykerJS's *Method Expression* [8] replaces JavaScript strings, arrays, and math methods interchangeably, and Stryker.NET's *String Methods* [9] does this for .NET string methods (e.g., `toUpperCase` with `toLowerCase`). These do not apply to Dafny, since (i) it provides limited built-in support for its types, and (ii) its strings are not objects.

3.4.3 Statement deletion operators. Mutmut's *Argument List Mutation* [6] deletes an element from a method's argument list. This is valid in Python due to the support of optional arguments with default values. Dafny, however, always requires a fixed number of arguments.

3.4.4 Object-Oriented operators. It is important to note that Dafny classes are only allowed to extend traits, which act as abstract superclasses and cannot declare constructors or be instantiated. Methods and fields can either be declared in the trait and used in its subclasses, or at the subclass level.

Inheritance. MuJava's inheritance-related operators [38, 40] test three different aspects of this feature in Java: method overriding, hiding variables, and the use of the `super` keyword, none of which are supported in Dafny.

Polymorphism. MuJava's polymorphism operators [38, 40] replace child types with parent types (and vice-versa) in various contexts, such as variable and parameter declarations and type casting. While some are possible in Dafny, the resulting mutants are always equivalent to the original.

Since superclasses cannot be instantiated, object creation always uses subclass constructors. Thus, even when declared with a type of a superclass, Dafny knows the type of the object, using the correct values for its fields and the correct child-specific implementation for methods that are declared but not implemented in the parent.

In mutations that replace child with parent types, if child-specific methods or fields are used, it leads to an invalid program. Otherwise, the mutant is equivalent. The inverse, parent-to-child replacements via type casting, requires prior type checking, e.g., inside of an `if var is Type` block, where `Type` is the type `var` will be cast into. These type-checking statements are unlikely to be present in programs that do not already perform downcasting, thus these operators will almost always produce invalid programs.

In summary, child-parent type replacements in Dafny yield either invalid or equivalent mutants and, hence, are not useful. However, we can note that child-to-child replacement (Section 3.3.13) is a valid polymorphic mutation.

`Static` *keyword.* MuJava defines operators that insert or delete the `static` keyword from class fields [38, 40]. In Java, this operator is useful because a class's behavior will differ depending on whether the updated fields are static. If a field is made static by mistake, its update in one object will cause all of the other objects of the same class to wrongfully have the same field updated to the same value. The inverse can also generate unwanted side effects.

In Dafny, only constant fields can be static. While static fields are stored in memory shared among all class instances, their constant nature means their value, and consequently program behavior, remains unchanged. Thus, these operators always generate equivalent mutants in Dafny.

4 Novel mutation operators for Dafny

Mutation operators typically try to mimic errors commonly made by developers when writing a program, and it has been shown that tailored mutants are well-coupled to real faults [10]. These may differ based on the features of each programming language. Although most of the mutation operators presented in Section 3 can be applied to Dafny programs, mimicking human errors in Dafny may require operators tailored for Dafny's specificities. In this section, we analyze bugfixes in real Dafny programs and synthesize them into a novel set of mutation operators for Dafny, aiming to complement the answer to **RQ1**.

4.1 Methodology

To the best of our knowledge, no prior study has identified which errors commonly occur in Dafny programs. To study this, inspired by Brown et al. [12]'s approach, we analyzed evidence of bugfixes in publicly available Dafny projects on GitHub. This analysis enabled us to extract new mutation operators tailored to Dafny, some of which are also applicable to other languages. These operators aim to simulate realistic developer errors and strengthen the relevance of our mutation testing tool, MutDafny (described in Section 5).

4.1.1 Dafny programs' repositories. DafnyBench[6] [37] is currently the largest dataset of Dafny programs, with 785 programs from 120 software repositories. Out of these, we noted that (a) there is no url, in DafnyBench's paper, for four repositories[7], (b) two were unavailable[8], (c) three were repeated[9], and (d) one was a subset of another[10], resulting in the exclusion of 10 repositories. Additionally, we included the repositories for the AWS encryption SDK[11] and Dafny-EVM[12], which are recognized as key examples of Dafny programs in industry, leaving us with 112 repositories[13].

4.1.2 Bugfix commits. Evidence of programming errors in software repositories may manifest in bug reports (i.e., issues labeled as *bug* on GitHub) or in commit messages referencing fixes or corrections. Given that 108 of the 112 repositories had no bug reports, our analysis focused on commit messages, covering a total of 12,308 commits. To identify potential bugfixes without complete manual inspection, and aiming to achieve replicability in identifying which commit messages hint at bugfixes, we selected commit messages using the `git log` command together with `grep`, and a set of keyword patterns: *"fix"*; *"correct"*, including other variations such as *"correction"*, *"corrected"*; *"updat"*, with variations like *"update"*, *"updating"*; and *"chang"*, including words like *"change"* and *"changing"*. This resulted in a total of 1,475 commits with messages hinting at possible bugfixes, some of which contained more than one of the listed patterns.

4.1.3 Procedure. Since our goal is to mutate implementations, changes involving specifications were disregarded. Each author of this paper individually analyzed one-third (≈491) of the 1,475 commits[14] and registered any changes that had the potential to be synthetized as mutation operators — e.g., modifications to expressions, altered conditions, added or removed lines of code — that could reflect common coding mistakes. We identified 270 (18.3%) commits as potential candidates. The authors then met to review the registered potential operators and discuss each until a consensus was reached. Of the 270, 148 were accepted, and 122 were not (Section 4.3 describes some reasons for this). The list of 1,475 commits and the outcome for each can be found in `bugfixes.csv` of the *supplementary material*[15]. The list of operators synthesized from bugfixes together with some source commits is available in the *supplementary material*[15].

4.2 Synthesized operators

Our bugfix analysis led to (i) the synthesis of 10 new mutation operators (nine that, although found in bugfixes of Dafny programs, could also be applied to programs written in other programming languages; and one tailored for Dafny), and (ii) the extension of the previously proposed mutation operator *COR* to support Dafny specificities. We briefly describe the 11 operators in the following subsections. For each operator, we provide the URL to one example from the analyzed commits, and, for some, a code example of the original and mutated versions in diff format to complement their description[16]. Next, we complete our answer to **RQ1** by listing and describing these new mutation operators.

4.2.1 Conditional Operator Replacement (COR). Dafny supports the standard conditional operators (Section 3.3.3) && and ||, common to most traditional programming languages, as well as additional ones: ==> (implication), <== (reverse implication), and <==> (equivalence). We found evidence[17] of replacements occurring between these operators in Dafny code. We have thus extended the previously defined conditional replacement operator to include them.

4.2.2 Expression replacement. Evidence shows expressions, in particular identifiers, to be commonly swapped.

Variable Expression Replacement (VER)[18] targets variable name references. Potential replacements should be limited to in-scope variables of the same type as the original.

[6] https://github.com/sun-wendy/DafnyBench/tree/0cd28fe, accessed Jan. 2026.
[7] BinarySearchTree, CO3408-Advanced-Software-Modelling-Assignment-2022-23-Part-2-A-Specification-Spectacular, Programmverifikation-und-synthese, and Prog-Fun-Solutions.
[8] https://github.com/AoxueDing/Dafny-Projects and https://github.com/Aaryan-Patel-2001/703FinalProject, accessed Jan. 2026.
[9] https://github.com/Eggy115/Dafny, https://github.com/vladstejeroiu/Dafny-programs, and https://github.com/isobelm/formal-verification, accessed Jan. 2026.
[10] https://github.com/secure-foundations/iron-sync ⊂ https://github.com/secure-foundations/ironsync-osdi2023, accessed Jan. 2026.
[11] https://github.com/aws/aws-encryption-sdk/tree/1be4d62, accessed Jan. 2026.
[12] https://github.com/Consensys/evm-dafny/tree/e2e52e8, accessed Jan. 2026.
[13] The list of repositories can be found at https://github.com/MutDafny/mutdafny-study-data/blob/main/subjects/data/generated/repositories.csv.
[14] Given we aim to synthesize modifications, as mutation operators, that introduce mistakes to the code and not that fix the code, we analyzed the inverse of bugfix commits, i.e., bugfixes as bug-introducing.
[15] https://doi.org/10.6084/m9.figshare.30640202.v2.
[16] Each diff applies the corresponding mutation operator to an existing code, i.e., introduces a *mistake*.
[17] https://github.com/Consensys/evm-dafny/commit/4946bec#diff-5f537a3009d26ba61bcb2a3b39a3c4d0dbc32fff56fd177427b7e5a245dd6a3fL61-R61, accessed Jan. 2026.
[18] https://github.com/dafny-lang/Dafny-VMC/commit/27a6601#diff-9ead0207f306a1ba28e42076fd59fe8bb6802fa2faa5288b6e43bc42d95f3b9cL28-R28, accessed Jan. 2026.

Field Access Replacement (FAR)[19] replaces an access to a class field with another of the same class, preserving the object reference and typing, as in the following example:

```
class Item {
    var item: string
    var price: int
    var stock: int
}

method GetProfit(item: Item) returns (profit: int)
{
-   profit := item.price * item.stock;
+   profit := item.price * item.price;
}
```

Method Call Replacement (MCR)[20] *and Datatype Constructor Replacement (DCR)*[21] work similarly to *VER* and *FAR*. *MCR* replaces one method (or function) call with another:

```
-   var n := Sum(10, 20); // return type int
+   var n := Multiply(10, 20); // return type int
```

The *DCR* operator replaces one datatype constructor call with another available from the same datatype:

```
datatype MyQuantifier = None | Some(x: set<int>) | All(y: set<int>)
method Main()
{
-   var selection := Some({1, 2, 3, 4});
+   var selection := All({1, 2, 3, 4});
}
```

Both operators retain the original arguments. To preserve program validity, replacements are limited to callable elements with matching signatures, i.e., same return type (for *MCR*) and same number, types, and order of arguments.

Method-Variable Replacement (MVR)[22]. Besides replacing variables with variables and method calls with other method calls, we also found evidence of method-to-variable replacements. In such cases, the replacement variable must match the return type of the target method.

4.2.3 Argument Swapping (SAR). A common change involves swapping the arguments of a callable element[23], e.g., a method, a function, or a constructor. The corresponding mutation operator replicates this by swapping two arguments in the same call. The mutated program is only valid if the swapped arguments are of the same type.

4.2.4 Tuple Access Replacement (TAR). In Dafny, tuple elements can be accessed by their positional index (as in `tuple.0`)[24]. This operator replaces a tuple index with another that refers to an element of the same type.

4.2.5 Conditional Block Extraction (CBE). In addition to deleting a branch of an `if` statement (see Section 3.3.10), our study found a common modification[25] where one branch's contents are extracted to the outside of the `if` statement, and the remaining branches are deleted. Unlike the first operator, which preserves the `if` statement, this one replaces it entirely with one branch's code block, as in the following:

```
-   if a <= 0 then
-       []
-   else
    var b, c := a + a, a * a;
    [a, b, c]
```

4.2.6 Statement Swapping (SWS). A common program fix involves swapping two or more lines of code[26], affecting logical flow and program behavior. While the concept is too broad to be fully translated into a mutation operator, due to the exponential mutant possibilities that would result from swapping every group of lines in a program, simpler operations can approximate it. We propose swapping a statement with the one directly above or below it.

4.2.7 Variable Declaration Swapping (SWV). Lastly, the *SWV* operator reflects another case related to the swapping of program constructs, this time, applied to the declaration of variables in the same scope[27].

```
-   var perimeter := 2.0 * radius * 3.14;
-   var area := radius * radius * 3.14;
+   var perimeter := radius * radius * 3.14;
+   var area := 2.0 * radius * 3.14;
```

This operator is similar to *VER*, but it affects the entire scope of the variable, while *VER* only affects the single place where the variable is replaced.

4.3 Excluded modifications

Many commits contained modifications that, at first glance, seemed promising, especially to those unfamiliar with Dafny's syntax, but that were ultimately not included as mutation operators. We can highlight three groups of common modifications: (a) changes that alone do not alter program behavior, (b) changes inherent to migrations from Dafny 3 to Dafny 4, and (c) code improvements.

The first group includes, e.g., replacing types that have subset relationships (like `nat` with `int`) and type parameter property changes[28,29]. The second relates to syntax changes introduced by Dafny 4 in how functions and callables are declared[30]. For example, functions were `ghost` by default (i.e., not compiled, only available for verification purposes),

[19] https://github.com/secure-foundations/ironsync-osdi2023/commit/9420f92#diff-8b6fe65b7c3a6ba702032935fb409c82b5a20dac0efcd4517966f82c1b980ee2L196-R196, accessed Jan. 2026.

[20] https://github.com/aws/aws-encryption-sdk/commit/36d1fad#diff-d6402a5709ccca6c7b42036edacf38b136517d3b5d6206557decd0f8123522adL258-R258, accessed Jan. 2026.

[21] https://github.com/secure-foundations/ironsync-osdi2023/commit/1d7eabc#diff-dd4cf8cbf6aaf5bb44a285c7bec0890c1092315f2b6a3548b0bfa4ad4f28c2a0L666-R666, accessed Jan. 2026.

[22] https://github.com/dafny-lang/Dafny-VMC/commit/ef3152c#diff-66dcba1192e097932aec9462cdb8e65134fc25c93afcfa612894b2c94e64af5bL99-R99, accessed Jan. 2026.

[23] https://github.com/aws/aws-encryption-sdk/commit/b409ea5#diff-f945451ea4d5cc43ed01a97ffb67bfabf7fc17727818427f892b90a7ce2c04e7L55, accessed Jan. 2026.

[24] https://github.com/aws/aws-encryption-sdk/commit/4c9d5d6#diff-00af0f655962fd28a7232ec2053eadda1257bb4587d838f9957a75c069e80437L64-R64, accessed Jan. 2026.

[25] https://github.com/benreynwar/SiLemma/commit/4da22d6#diff-0eff5d0d272121340e6e9d6bad6e03612a3057426e284b56cd185a1524d7937aL572-R577, accessed Jan. 2026.

[26] https://github.com/dafny-lang/libraries/commit/5ecd461#diff-3a16ac84e5bda345c586253ae5dbaa34730085d17a587393650195f5248eea80L46-R48, accessed Jan. 2026.

[27] https://github.com/Consensys/evm-dafny/commit/58abe25#diff-eec5c501c95e783482e4e062ad3c9c30e947c394e0388bc197ad2306a5840ea5L315-R316, accessed Jan. 2026.

[28] https://github.com/secure-foundations/ironsync-osdi2023/commit/e27b26c#diff-7091ca089a282c06587ac4406db89d238ea977e5e47b93c290d769c1221f2be3L14-R14, accessed Jan. 2026.

[29] https://dafny.org/dafny/DafnyRef/DafnyRef.html#sec-type-characteristics, accessed Jan. 2026.

[30] https://github.com/namin/dafny-sandbox/commit/21e148e#diff-4698b1118ee26951776954c2d28642a7dc6ad3abb38743d0034655ffdf302364L112-R112, accessed Jan. 2026.

but now all functions compile unless declared as `ghost`. Finally, the last group includes common code improvement changes, e.g., the replacement of the declaration of a callable element as a method, function, predicate, or lemma, and the insertion or deletion of `ghost` keywords[31,32], which optimizes compilation by eliminating the need to compile certain elements, without affecting program behavior.

5 MutDafny

MutDafny supports 40 mutation operators[33]: the 30 described in Section 3 and the 10 described in Section 4. The tool is integrated into the Dafny verification framework, implemented as a plugin. Dafny's plugin architecture provides an extensible interface for Abstract Syntax Tree (AST) analysis and manipulation, allowing MutDafny to integrate directly with Dafny's execution flow without modifying its source code. MutDafny works at the level between the parser and verifier and consists of a two-phase pipeline: *mutant generation* and *mutation analysis*, illustrated in Figure 1.

Mutant generation. Operates by directly manipulating Dafny's internal representation. A *scanner* component traverses the AST to identify *mutation targets* — specific locations in the program where specific types of mutations can be applied — and collects the necessary context for performing those mutations. Namely, for each target, MutDafny records the node's position, the type of mutation operator to be used, and any additional arguments required to apply the mutation. The *mutator* component processes each of the identified mutation targets and changes the program's AST to create individual mutants prior to Dafny's resolution phase. This is done by traversing the AST until the target's location, i.e., the node to mutate, is reached, and then performing the necessary operations on it. The changes can involve altering a node's data, replacing an expression or statement with a different one by creating a new node (e.g., the *BBR* operator involves the replacement of a binary expression with a boolean literal one), or deleting nodes by removing their reference from the parent.

For certain mutation operators, identifying mutation targets involves analyzing the type information of the target node, which is only available post-resolution. An example is the *AOI* operator, which can only be applied to numerical expressions. Here, to know whether a tree node constitutes a valid mutation target, we must first know its type. As a result, and as depicted in Figure 1, a portion of the targets are identified during a pre-resolve tree visit, and the remaining are identified during a post-resolve one.

Mutation analysis. The generated mutants are evaluated using Dafny's verification capabilities and the specification under test. Mutants are classified according to the verification outcome — *Alive (survived):* the mutant verifies successfully against the original spec, hinting at a potential specification weakness; *Killed:* the mutant fails verification, meaning that the spec successfully detected the inserted fault; *Invalid:* the mutant fails during Dafny's resolution process (before verification), indicating a semantically incorrect program; *Timed Out:* the verifier could not determine an outcome within a set time limit[34].

6 Empirical Evaluation

In this section, we describe and discuss the results of our empirical evaluation, which aims to answer the following research questions:

RQ2: Are mutation operators for Dafny programs helpful in the identification of specification weaknesses?

RQ3: How effective are the different mutation operators?

RQ4: How efficient is MutDafny at generating mutants?

6.1 Experimental procedure

The empirical evaluation of MutDafny was conducted on a dataset with 43,178 lines of code consisting of 743 Dafny files[35] from DafnyBench [37], 17 from AWS encryption SDK, and 34 from Dafny-EVM. DafnyBench includes 785 files, but 42 of these were unverifiable with our environment. We excluded any files that (a) had validity errors with our Dafny/Z3 version, (b) timed out, (c) had verification errors, and (d) verified nothing at all (e.g., `Dafny program verifier finished with 0 verified, 0 errors`)[36].

The 794 program files that constitute our experimental subjects gather a total of 2,747 methods, functions, and similar constructs: 1,668 of these have no pre-conditions, 1,403 no post-conditions, and 1,044 have neither. This does not necessarily imply that the spec is *weak*. In fact, it is common for functions and predicates to omit such contracts. However, when these are invoked within methods that *do* have specs, mutations affecting them can still contribute to measuring the quality of the overall program's specification.

Our experimental procedure consists of running MutDafny against the 794 files. To accelerate the experiment runtime, we parallelized some workflow components. Each scanner job involves scanning a single program for mutation targets using a single mutation operator, while for the mutator, each job entails generating mutants for a single program. We executed each experiment (i.e., a mutation operator on a file) 10 times. We then removed 20% of the repetitions with the highest and the lowest runtimes.

The experiment was run on a cluster running Ubuntu 20.04, Linux kernel version 5.15, with 1GB of RAM, and an Intel(R) E5-2450 CPU at 2.1GHz. We use Dafny 4.10.0. (version corresponding to commit `7159879`[37]) and Z3 v4.12.6.

[31] https://github.com/secure-foundations/ironsync-osdi2023/commit/0da7158, accessed Jan. 2026.

[32] https://github.com/secure-foundations/ironsync-osdi2023/commit/de87232#diff-10aacdf18deb66491a2e4bb876eaeff2e7979b41d427563229817337b5302753L412-R420, accessed Jan. 2026.

[33] The full list of operators implemented in MutDafny can be consulted in the *supplementary material* at https://doi.org/10.6084/m9.figshare.30640202.v2.

[34] By default, Dafny's verifier runs for 20 seconds.

[35] In this paper, we consider a Dafny *file*, .dfy, as a Dafny *program* (as defined in DafnyBench [37]), and we may use the terms *file* or *program* interchangeably, in which case we imply the former.

[36] Examples of excluded files: (a) `groupTheory_tmp_tmppmmxvu8h_assignment1`; (b) `WrappedEther`; (c) `Clover_insert`; and (d) `DafnyExercises_tmp_tmpd6qyevja_QuickExercises_testing2`. The full list of files and the reason why each did not verify can be found in the *supplementary material* at https://doi.org/10.6084/m9.figshare.30640202.v2.

[37] https://github.com/dafny-lang/dafny/tree/7159879, accessed Jan. 2026.

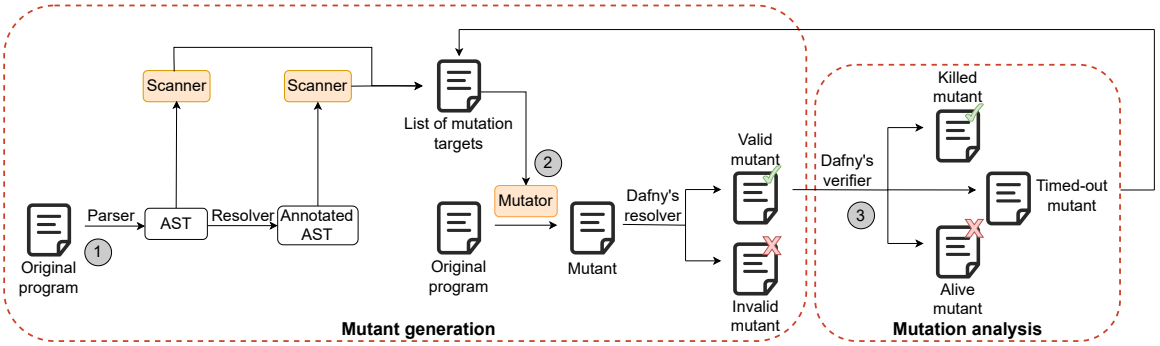

Figure 1: Mutation testing pipeline with MutDafny. MutDafny's components are depicted by the filled boxes, the rest corresponding to Dafny.

6.2 Experimental metrics

The dataset resulting from running this experiment includes all generated mutants, runtime measurements and mutation target data for all file scans, runtime measurements for every mutant generation, and data identifying the applied mutation and its analysis status. For each execution, we record the total runtime of each scan/mutation process and the time spent in each of Dafny's workflow stages and executing the plugin. This data allows us to compute the mutant generation, mutation analysis, and total runtime.

Additionally, we compute the *mutation score* [26], i.e., the ratio of killed mutants to the total number of mutants, excluding invalid mutants, e.g., that introduce non-compiling changes, and time-out ones, which reflect an inconclusive result. This metric reflects the fault-detection capability, typically of the test suite, and, in the context of this work, of the specifications. A higher mutation score increases confidence in the test suite/specification.

6.3 Experimental results

Overall, **MutDafny generated a total of 118,458 mutants** across our dataset of 794 programs, an average of 2.7 mutants per line of code.

6.3.1 ***Answer to RQ2.*** We compute the mutation score of each program in the dataset, illustrated in Figure 2. On average, the dataset's **specifications are able to kill 82% of the mutants**. The middle 50% scores range from 76% to 96% with a standard deviation of 23%, indicating a concentration of high mutation scores. These results show that Dafny specifications exhibit high mutant-detection capability. Still, a single alive mutant may be enough to reveal a weakness.

From the 30,459 surviving mutants, 11,035 affect methods with post-conditions. The remaining 19,424 can be weak due to a lack of specification in the mutated component, hint at a weakness in a component calling the mutated one, or be equivalent to the original program. Due to the infeasibility of manually analyzing 11,035 mutants to assess whether they survived due to a weakness or equivalence to the original program, we focused on 24 randomly selected files and their 284 surviving mutants in methods with post-conditions. This took eight hours of one person's effort (≈ 1.7 minutes per mutant), with the files averaging 50.2 lines of code (1,205 total) and 3.3 functions (79 total).

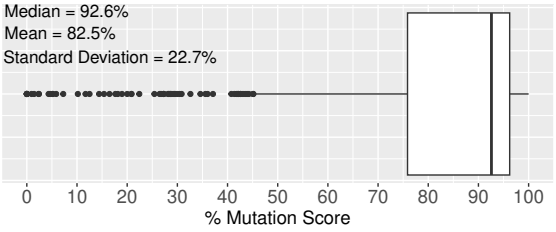

Figure 2: Mutation score distribution.

In these 24 files, we found five specification weaknesses[38,39,40,41,42] in five programs from two different projects, one[41] corresponding to the motivational example presented in Section 1. The repository of one of the weak specs[38] became unavailable, so we could not contact its developers. The remaining programs were generated by LLMs for a study [45], and a manual inspection led Misu et al. [45] to report one of the weaknesses[39] (confirming our finding). We contacted them for reporting the remaining three that went unnoticed[43] and, while Misu et al. acknowledged our discoveries, it was also pointed out that two of the weaknesses[40,41] had previously been detected by Endres et al. [20], the third one[42] being a new discovery.

In the case of `bst4copy`[38], the `insert` method ensures that the output tree is a valid binary search tree, but there is no condition asserting that the insertion of the input value indeed occurred. Similarly, the `insertRecursion` method checks the relation between the tree's lowest and highest value and the value to be inserted, but does not check if the latter is present in the output tree. If no modification is made to the input tree, neither method's specification would be able to detect the incorrect behavior. In `task_id_126`[39],

[38] https://github.com/sun-wendy/DafnyBench/blob/main/DafnyBench/dataset/ground_truth/BinarySearchTree_tmp_tmp_bn2twp5_bst4copy.dfy#L58, accessed Jan. 2026.
[39] https://github.com/sun-wendy/DafnyBench/blob/main/DafnyBench/dataset/ground_truth/dafny-synthesis_task_id_126.dfy#L1, accessed Jan. 2026.
[40] https://github.com/sun-wendy/DafnyBench/blob/main/DafnyBench/dataset/ground_truth/dafny-synthesis_task_id_161.dfy#L7, accessed Jan. 2026.
[41] https://github.com/sun-wendy/DafnyBench/blob/main/DafnyBench/dataset/ground_truth/dafny-synthesis_task_id_2.dfy#L7, accessed Jan. 2026.
[42] https://github.com/sun-wendy/DafnyBench/blob/main/DafnyBench/dataset/ground_truth/dafny-synthesis_task_id_249.dfy#L7, accessed Jan. 2026.
[43] https://github.com/Mondego/dafny-synthesis/issues/2, accessed Jan. 2026.

the post-condition states that every divisor common to two input numbers is lower than their sum. However, if an incorrect implementation caused only some of the common divisors to be added to the sum, this single post-condition would not be able to detect it. The purpose of `task_id_161`[40] is to return every unique element present in the first input array but not in the second. While the specification states that every element in the result must be in the first array but not in the second, it does not enforce that every element in the first array that is not present in the second must be a part of the result, making empty lists a valid output. `task_id_2`[41] and `task_id_249`[42] are very similar to `task_id_161`, the goal being to compute the unique elements common to two input arrays, and the weakness being the same. All of these weaknesses fail to address a crucial guarantee.

Of the 284 analyzed mutants, 77 hinted at these weaknesses and were successfully killed upon strengthening the specifications, 157 were equivalent to the original program, and we could not draw meaningful conclusions from the remaining 50. The number of equivalent mutants, along with the time to perform manual verification of the surviving mutants, poses an evident challenge. In practice, we expect MutDafny to be used as a validation tool alongside development to incrementally validate new components rather than entire projects at once. This will result in fewer mutants per iteration and shorten analysis time, making it feasible.

6.3.2 ***Answer to RQ3.*** Table 1 reports, for every mutation operator, how many of the mutants generated with it were killed, survived, invalid, or whose verification timed out. *AOI*, *ROR*, *EVR*, *LVR*, and *VER* are responsible for the generation of the higher number of mutants, which is expected since they target common syntactic constructs: respectively, binary operators, expressions, and variables. *AMR*, *PRV*, and *THD* do not generate any mutants, and *THI*, *TAR*, *SOR*, and *LOD* generate a low number (less than 20 each). This happens because each of these operators can only be applied in very specific scenarios. For example, *THD* entails a program location where a class field is accessed using the `this` keyword inside a method whose argument has the same name as that field. *THI* entails the opposite. Furthermore, logical unary and shift operators occur very rarely in our dataset.

As summarized in Table 2, the five specification weaknesses unveiled during our manual analysis resulted from mutants generated by 10 of the existing and five of the newly proposed operators. *EVR*, *VER* (new), and *ROR* were the ones that generated the largest number of weakness-revealing mutants. These do not correspond to the operators that, out of the 40, generate the highest percentages of survivors; only around 20-30%. Instead, they generate more mutants overall, which suggests a correlation between a high mutant count and a greater likelihood of discovering weaknesses.

Regarding the relative effectiveness between the previously proposed and the novel mutation operators, the Wilcoxon test (with a confidence level of 99%) reports that programs' specifications are statistically significantly (p-value = 1.749×10^{11}) more effective at killing mutants generated by the former than by the latter. These results further motivate one of our contributions, highlighting the need for mutation operators tailored for Dafny to more effectively evaluate Dafny specifications.

Table 1: Number of generated, killed, survived, invalid, and timeout mutants.

Op.	# Mut	# Killed	# Survived	# Invalid	# Timeout
AMR	0	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
AOD	250	199 (79.6%)	51 (20.4%)	0 (0.0%)	0 (0.0%)
AOI	15,787	13,650 (86.5%)	1,940 (12.3%)	122 (0.8%)	75 (0.5%)
AOR	6,202	4,963 (80.0%)	794 (12.8%)	371 (6.0%)	74 (1.2%)
BBR	5,722	3,695 (64.6%)	1,869 (32.7%)	144 (2.5%)	14 (0.2%)
CBR	369	3 (0.8%)	364 (98.6%)	2 (0.5%)	0 (0.0%)
CIR	926	657 (71.0%)	233 (25.2%)	36 (3.9%)	0 (0.0%)
COD	76	64 (84.2%)	12 (15.8%)	0 (0.0%)	0 (0.0%)
COI	3,851	2,883 (74.9%)	814 (21.1%)	142 (3.7%)	12 (0.3%)
COR	1,792	839 (46.8%)	731 (40.8%)	216 (12.1%)	6 (0.3%)
EVR	13,976	10,080 (72.1%)	3,276 (23.4%)	557 (4.0%)	63 (0.5%)
LBI	738	621 (84.2%)	115 (15.6%)	1 (0.1%)	1 (0.1%)
LOD	1	0 (0.0%)	1 (100.0%)	0 (0.0%)	0 (0.0%)
LOI	33	19 (57.6%)	11 (33.3%)	2 (6.1%)	1 (3.0%)
LOR	20	9 (45.0%)	10 (50.0%)	0 (0.0%)	1 (5.0%)
LSR	54	30 (55.6%)	24 (44.4%)	0 (0.0%)	0 (0.0%)
LVR	13,927	9,860 (70.8%)	4,005 (28.8%)	31 (0.2%)	31 (0.2%)
MAP	5,399	345 (6.4%)	2,512 (46.5%)	2,539 (47.0%)	3 (0.1%)
MMR	20	10 (50.0%)	10 (50.0%)	0 (0.0%)	0 (0.0%)
MNR	421	124 (29.4%)	237 (56.3%)	60 (14.2%)	0 (0.0%)
MRR	2,286	976 (42.7%)	433 (18.9%)	874 (38.2%)	3 (0.1%)
ODL	4,418	2,115 (47.9%)	265 (6.0%)	2,014 (45.6%)	24 (0.5%)
PRV	0	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
ROR	13,180	8,187 (62.1%)	3,996 (30.3%)	978 (7.4%)	19 (0.1%)
SDL	5,952	3,269 (54.9%)	1,301 (21.9%)	1,369 (23.0%)	13 (0.2%)
SLD	127	85 (66.9%)	41 (32.3%)	0 (0.0%)	1 (0.8%)
SOR	2	2 (100.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
THI	8	5 (62.5%)	1 (12.5%)	2 (25.0%)	0 (0.0%)
THD	0	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
VDL	1,248	196 (15.7%)	589 (47.2%)	460 (36.9%)	3 (0.2%)
Total	96,785	62,886 (65.0%)	23,635 (24.4%)	9,920 (10.2%)	344 (0.4%)
CBE	1,801	1,217 (67.6%)	564 (31.3%)	15 (0.8%)	5 (0.3%)
DCR	415	243 (58.5%)	113 (27.2%)	57 (13.7%)	2 (0.5%)
FAR	93	60 (64.5%)	30 (32.3%)	3 (3.2%)	0 (0.0%)
MCR	478	112 (23.4%)	349 (73.0%)	17 (3.6%)	0 (0.0%)
MVR	1,972	1,258 (63.8%)	481 (24.4%)	226 (11.5%)	7 (0.3%)
SAR	923	458 (49.6%)	401 (43.5%)	60 (6.5%)	4 (0.4%)
SWS	3,671	1,258 (34.3%)	2,194 (59.8%)	206 (5.6%)	13 (0. %)
SWV	400	188 (47.0%)	153 (38.2%)	59 (14.8%)	0 (0.0%)
TAR	16	8 (50.0%)	3 (18.8%)	5 (31.2%)	0 (0.0%)
VER	11,904	8,829 (74.2%)	2,536 (21.3%)	428 (3.6%)	111 (0.9%)
Total	21,673	13,631 (62.9%)	6,824 (31.5%)	1,076 (5.0%)	142 (0.7%)
Overall Total	118,458	76,517 (64.6%)	30,459 (25.7%)	10,996 (9.3%)	486 (0.4%)

Table 2: The number of mutants generated by each mutation operator that led to the identification of weak specifications.

Op.	`bst4copy`[38]	`task_id_126`[39]	`task_id_161`[40]	`task_id_2`[41]	`task_id_249`[42]	Total
EVR	1	5	4	1	4	15
VER	2	11	1	0	0	14
ROR	5	2	0	0	0	7
BBR	2	3	1	0	1	7
COR	0	4	1	0	1	6
MAP	6	0	0	0	0	6
MVR	4	0	0	0	0	4
SDL	0	0	2	0	2	4
DCR	3	0	0	0	0	3
LBI	0	0	1	1	1	3
CBE	1	1	0	0	0	2
CIR	0	0	1	0	1	2
SWS	0	0	1	0	1	2
LVR	0	1	0	0	0	1
UOI	1	0	0	0	0	1
Total	25	27	12	2	11	77

6.3.3 ***Answer to RQ4.*** Figure 3 shows the distribution of the runtime of the mutation testing process of each program. **The average total runtime is 22 minutes, with half of the experiments falling between two and 13 minutes.** These runtimes may be acceptable for the analysis of individual files, but can raise scalability concerns for multi-file projects. However, these issues can be mitigated through parallelization, as done in our experimental procedure.

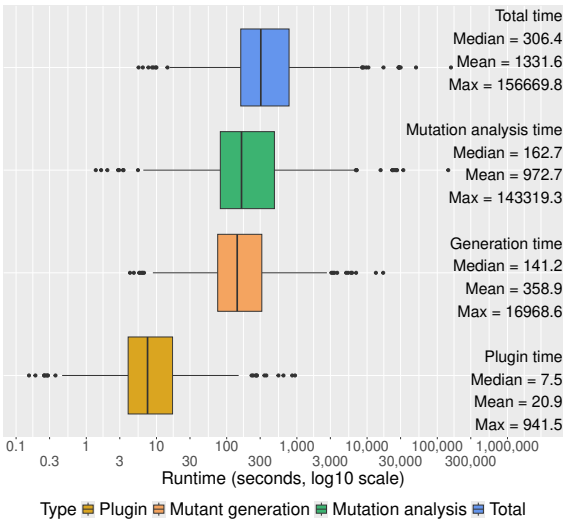

Figure 3: Plugin, mutant generation, mutation analysis, and total runtime to perform mutation testing of a Dafny file.

The two phases of the mutation testing pipeline — generation and analysis — present similar average runtimes. Therefore, we conclude that the total runtime is approximately divided between mutant generation and verification.

While MutDafny usually takes one to five minutes to generate all the mutants for one file, this does not accurately reflect our implementation's runtime. In half the files, the plugin scans and mutates the program in four to 17 seconds, accounting for about 5% of the mutant generation runtime. The remaining time is spent on mutant resolution, a regular part of Dafny's execution but crucial for identifying invalid mutants. MutDafny's efficiency is, thus, mainly affected by the resolution and verification of the mutated file.

6.4 Threats to validity

In this subsection, we discuss threats to validity according to the guidelines defined by Wohlin et al. [53] and Yin [56].

Threats to construct validity. We identify two main constructs in our study: MutDafny's integration with Dafny and specification effectiveness. Regarding the former, we define how the tool integrates into Dafny's workflow, identifying the mutation and verification points. For the latter, we adopt the standard mutation score to evaluate specification effectiveness, clearly defining mutant statuses and detection criteria to align with established mutation measurements.

Threats to internal validity. The main threats we identify are CPU warm-up distortion and software bugs in MutDafny. The first was mitigated by running the experiment 10 times and removing outliers. Regarding software bugs, we employed testing and manually validated a subset of results. Our source code and experimental results are publicly available to support verification and reproducibility.

Threats to external validity. To support external validity, we rely on the representativeness of DafnyBench, which includes synthesized and real-world program files, the latter constituting 75% of the dataset and having been collected from GitHub with minimal filtering. Programs vary in complexity, some comprising introductory examples written by new learners of Dafny, thus with less relevance for rigorous assessment. Yet, all include specifications, allowing the evaluation of their response to the tool. In addition, the files from AWS and Dafny-EVM are recognized as key industry examples. Nevertheless, the limited availability of high-quality open-source Dafny repositories is still a challenge.

7 Future Work

It is crucial to ease the manual examination of surviving mutants by minimizing the number of weakness hints that are either equivalent or false positives. As future work, we aim to investigate the application of code coverage of Dafny specifications to minimize the number of generated mutants [28], i.e., if a particular code portion is not covered by any existing specification in the program, then no mutant in it can ever be killed. In addition, we aim to explore automatic techniques to find and discard equivalent [24, 29, 31, 32, 47] or ineffective [44] mutants in the context of Dafny.

8 Conclusion

We presented a methodology for verification-aware languages and its implementation in a novel tool for detecting specification weaknesses through mutation of Dafny programs, benefiting from direct integration with the language's workflow. Another key contribution of this paper is the systematic study of mutation operators suitable for Dafny, resulting in the synthesis of 30 unique operators previously proposed for other programming languages, as well as 10 newly synthesized from evidence collected in real-world Dafny bugfixes.

Our empirical evaluation across 794 Dafny programs and 118,458 mutants demonstrated the value of MutDafny in detecting weaknesses. The results show that Dafny specifications have a high mutant-detection capability. However, the analysis of a sample of alive mutants revealed five weaknesses in the benchmark programs. Despite the runtimes being highly conditioned by Dafny's verification pipeline and the lengthy manual inspection of alive mutants, the tool can be feasibly applied in real-world settings. Ultimately, MutDafny has the potential to advance formal verification by aiding Dafny developers in improving system confidence.

Acknowledgments

Isabel Amaral 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). Alexandra Mendes is funded 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). José Campos was supported by the LASIGE Research Unit, ref. UID/00408/2025, DOI 10.54499/UID/00408/2025.

References

[1] 2016. *Go-mutesting (fork).* https://github.com/avito-tech/go-mutesting Last accessed Nov. 2025.

[2] 2016. *Go-mutesting (original).* https://github.com/zimmski/go-mutesting Last accessed Nov. 2025.

[3] 2011. *Major.* https://mutation-testing.org Last accessed Nov. 2025.

[4] 2005. *MuJava.* https://github.com/jeffoffutt/muJava Last accessed Nov. 2025.

[5] 2018. *Mull.* https://github.com/mull-project/mull Last accessed Nov. 2025.

[6] 2018. *Mutmut.* https://github.com/boxed/mutmut Last accessed Nov. 2025.

[7] 2016. *PIT.* http://pitest.org Last accessed Nov. 2025.

[8] 2017. *StrykerJS.* https://stryker-mutator.io/docs/stryker-js/introduction/ Last accessed Nov. 2025.

[9] 2021. *Stryker.NET.* https://stryker-mutator.io/docs/stryker-net/introduction/ Last accessed Nov. 2025..

[10] Miltiadis Allamanis, Earl T. Barr, René Just, and Charles Sutton. 2016. Tailored Mutants Fit Bugs Better. arXiv:1611.02516 [cs.SE] https://arxiv.org/abs/1611.02516

[11] Lennart Beringer, Adam Petcher, Q Ye Katherine, and Andrew W Appel. 2015. Verified Correctness and Security of OpenSSL HMAC.. In *USENIX Security Symposium*. 207–221.

[12] David Bingham Brown, Michael Vaughn, Ben Liblit, and Thomas Reps. 2017. The care and feeding of wild-caught mutants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 511–522. doi:10.1145/3106237.3106280

[13] Timothy A. Budd and Dana Angluin. 1982. Two notions of correctness and their relation to testing. *Acta Inf.* 18, 1 (March 1982), 31–45. doi:10.1007/BF00625279

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

[15] Aleks Chakarov, Jaco Geldenhuys, Matthew Heck, Michael Hicks, Sam Huang, Georges-Axel Jaloyan, Anjali Joshi, K Rustan M Leino, Mikael Mayer, Sean McLaughlin, et al. 2025. Formally Verified Cloud-Scale Authorization. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 703–703.

[16] Yoonsik Cheon and Gary T. Leavens. 2002. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. In *ECOOP 2002 — Object-Oriented Programming*, Boris Magnusson (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 231–255.

[17] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. PIT: a practical mutation testing tool for Java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 449–452. doi:10.1145/2931037.2948707

[18] Alex Denisov and Stanislav Pankevich. 2018. Mull It Over: Mutation Testing Based on LLVM. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 25–31. doi:10.1109/ICSTW.2018.00024

[19] Anna Derezińska and Konrad Hałas. 2014. Analysis of Mutation Operators for the Python Language. In *Proceedings of the Ninth International Conference on Dependability and Complex Systems DepCoS-RELCOMEX. June 30 – July 4, 2014, Brunów, Poland*, Wojciech Zamojski, Jacek Mazurkiewicz, Jarosław Sugier, Tomasz Walkowiak, and Janusz Kacprzyk (Eds.). Springer International Publishing, Cham, 155–164.

[20] Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K. Lahiri. 2024. Can Large Language Models Transform Natural Language Intent into Formal Method Postconditions? *Proc. ACM Softw. Eng.* 1, FSE, Article 84 (July 2024), 24 pages. doi:10.1145/3660791

[21] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. 2017. An Empirical Study on the Correctness of Formally Verified Distributed Systems. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 328–343. doi:10.1145/3064176.3064183

[22] Eli Goldweber, Weixin Yu, Seyed Armin Vakil Ghahani, and Manos Kapritsos. 2024. IronSpec: Increasing the Reliability of Formal Specifications. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 875–891. https://www.usenix.org/conference/osdi24/presentation/goldweber

[23] Rahul Gopinath, Jie M. Zhang, Marinos Kintis, and Mike Papadakis. 2022. Mutation analysis and its industrial applications. *Software Testing, Verification and Reliability* 32, 8 (2022), e1831. doi:10.1002/stvr.1831 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1831

[24] Mahdi Houshmand and Samad Paydar. 2017. TCE+: An Extension of the TCE Method for Detecting Equivalent Mutants in Java Programs. In *Fundamentals of Software Engineering*, Mehdi Dastani and Marjan Sirjani (Eds.). Springer International Publishing, Cham, 164–179.

[25] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2011), 649–678. doi:10.1109/TSE.2010.62

[26] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2011), 649–678. doi:10.1109/TSE.2010.62

[27] René Just. 2014. The major mutation framework: efficient and scalable mutation analysis for Java. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 433–436. doi:10.1145/2610384.2628053

[28] René Just, Franz Schweiggert, and Gregory M. Kapfhammer. 2011. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. 612–615. doi:10.1109/ASE.2011.6100138

[29] Marinos Kintis, Mike Papadakis, Yue Jia, Nicos Malevris, Yves Le Traon, and Mark Harman. 2018. Detecting Trivial Mutant Equivalences via Compiler Optimisations. *IEEE Transactions on Software Engineering* 44, 4 (2018), 308–333. doi:10.1109/TSE.2017.2684805

[30] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 207–220.

[31] Benjamin Kushigian, Samuel J. Kaufman, Ryan Featherman, Hannah Potter, Ardi Madadi, and René Just. 2024. Equivalent Mutants in the Wild: Identifying and Efficiently Suppressing Equivalent Mutants for Java Programs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 654–665. doi:10.1145/3650212.3680310

[32] Benjamin Kushigian, Amit Rawat, and René Just. 2019. Medusa: Mutant Equivalence Detection Using Satisfiability Analysis. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 77–82. doi:10.1109/ICSTW.2019.00035

[33] Shuvendu K. Lahiri. 2024. Evaluating LLM-driven User-Intent Formalization for Verification-Aware Languages. doi:10.34727/2024/ISBN.978-3-85448-065-5_19

[34] Y. Le Traon, B. Baudry, and J.-M. Jezequel. 2006. Design by Contract to Improve Software Vigilance. *IEEE Transactions on Software Engineering* 32, 8 (2006), 571–586. doi:10.1109/TSE.2006.79

[35] 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.

[36] Xavier Leroy. 2009. A formally verified compiler back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363.

[37] 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. (2024). arXiv:2406.08467 [cs.SE] https://arxiv.org/abs/2406.08467

[38] Yu-seung Ma and Jeff Offutt. 2014. Description of Class Mutation Mutation Operators for Java. (2014). https://cs.gmu.edu/~offutt/mujava/mutopsClass.pdf

[39] Yu-seung Ma and Jeff Offutt. Update 2016. Description of muJava's Method-level Mutation Operators. (Update 2016). https://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf

[40] Yu-Seung Ma, Jeff Offutt, and Yong Kwon. 2005. MuJava: An automated class mutation system. *Softw. Test., Verif. Reliab.* 15 (06 2005), 97–133. doi:10.1002/stvr.308

[41] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. 2006. MuJava: a mutation system for java. In *Proceedings of the 28th International Conference on Software Engineering* (Shanghai, China) *(ICSE '06)*. Association for Computing Machinery, New York, NY, USA, 827–830. doi:10.1145/1134285.1134425

[42] Dor Ma'ayan, Shahar Maoz, and Roey Rozi. 2022. Validating the correctness of reactive systems specifications through systematic exploration. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems* (Montreal, Quebec, Canada) *(MODELS '22)*. Association for Computing Machinery, New York, NY, USA, 132–142. doi:10.1145/3550355.3552425

[43] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Jozala. 2014. Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation. *IEEE Trans. Softw. Eng.* 40, 1 (Jan. 2014), 23–42. doi:10.1109/TSE.2013.44

[44] Phil McMinn, Chris J. Wright, Colton J. McCurdy, and Gregory M. Kapfhammer. 2019. Automatic Detection and Removal of Ineffective Mutants for the Mutation Analysis of Relational Database Schemas. *IEEE Transactions on Software Engineering* 45, 5 (2019), 427–463. doi:10

.1109/TSE.2017.2786286

[45] Md Rakib Hossain Misu, Cristina V. Lopes, Iris Ma, and James Noble. 2024. Towards AI-Assisted Synthesis of Verified Dafny Methods. arXiv:2402.00247 [cs.AI] https://arxiv.org/abs/2402.00247

[46] Glenford J. Myers, Corey Sandler, and Tom Badgett. 2011. *The Art of Software Testing* (3rd ed.). Wiley Publishing.

[47] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. 2015. Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 936–946. doi:10.1109/ICSE.2015.103

[48] Ana B. Sánchez, Pedro Delgado-Pérez, Inmaculada Medina-Bulo, and Sergio Segura. 2022. Mutation testing in the wild: findings from GitHub. *Empirical Softw. Engg.* 27, 6 (Nov. 2022), 35 pages. doi:10.1007/s10664-022-10177-8

[49] Ana B. Sánchez, José A. Parejo, Sergio Segura, Amador Durán, and Mike Papadakis. 2024. Mutation Testing in Practice: Insights From Open-Source Software Developers. *IEEE Transactions on Software Engineering* 50, 5 (2024), 1130–1143. doi:10.1109/TSE.2024.3377378

[50] Roy Patrick Tan and Stephen H. Edwards. 2004. Experiences evaluating the effectiveness of JML-JUnit testing. *SIGSOFT Softw. Eng. Notes* 29, 5 (Sept. 2004), 1–4. doi:10.1145/1022494.1022545

[51] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. Learning How to Mutate Source Code from Bug-Fixes . In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE Computer Society, Los Alamitos, CA, USA, 301–312. doi:10.1109/ICSME.2019.00046

[52] Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. 2018. MuAlloy: a mutation testing framework for alloy. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 29–32. doi:10.1145/3183440.3183488

[53] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, Anders Wesslén, et al. 2012. *Experimentation in software engineering*. Vol. 236. Springer.

[54] M.R. Woodward. 1993. Mutation testing—its origin and evolution. *Information and Software Technology* 35, 3 (1993), 163–169. doi:10.1016/0950-5849(93)90053-6

[55] Xiangjuan Yao, Mark Harman, and Yue Jia. 2014. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 919–930. doi:10.1145/2568225.2568265

[56] R.K. Yin. 2009. *Case Study Research: Design and Methods*. SAGE Publications.