

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

MutDafny: A Mutation Testing Tool for Dafny Programs

Isabel André Amaral



Master in Informatics and Computing Engineering

Supervisor: Alexandra Mendes

Second Supervisor: José Campos

July 24, 2025

MutDafny: A Mutation Testing Tool for Dafny Programs

Isabel André Amaral

Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

President: João Carlos Pascoal Faria

Examiner: Cristina Videira Lopes

Supervisor: Alexandra Mendes

July 24, 2025

Resumo

Verificação formal é essencial em sistemas críticos, nos quais falhas de software podem ter consequências catastróficas. As linguagens de programação com suporte para verificação formal (*verification-aware*), tal como Dafny, garantem que a implementação de um programa se alinha com a sua especificação. Ainda assim, enquanto que especificações são habitualmente consideradas corretas, programadores podem tão facilmente cometer um erro ao escrever uma como ao escrever uma implementação, resultando em programas que verificam com sucesso sem capturar a totalidade da funcionalidade pretendida. A investigação de técnicas para ajudar a identificar pontos fracos em especificações relativamente a uma implementação que se assume capturar a intenção do programador é crucial para aumentar a confiança nestes sistemas.

Esta tese explora uma metodologia de deteção de pontos fracos em especificações assente no conceito de teste de mutação (*mutation testing*). Considerando a especificação como um critério de correção equiparável a um conjunto de testes, o processo de análise de mutação (*mutation analysis*), particularmente versões mutadas do programa original que passam o processo de verificação sem serem detetadas, podem ser usadas para sugerir a presença de uma especificação fraca. Neste trabalho, levamos a cabo um estudo aprofundado explorando e avaliando operadores de mutação adequados para a linguagem Dafny, inicialmente através de um levantamento de operadores previamente propostos para outras linguagens de programação e, de seguida, através de uma análise sistemática de *commits* de correção de erros em programas em Dafny. Este conhecimento foi reunido no desenvolvimento de uma ferramenta de *mutation testing*, o MutDafny, diretamente integrado com o fluxo de execução do Dafny.

Efetuamos a avaliação da capacidade de geração de mutantes e eficiência do MutDafny, bem como da capacidade de deteção de mutantes das especificações através de um dataset de 745 programas com cerca de 53,000 linhas de código. Para este dataset, a ferramenta gera 87,941 mutantes, 71.24% destes sendo detetados pelas suas especificações. Efetuamos uma avaliação manual de um subgrupo dos mutantes que passam a verificação sem serem detetados e encontramos vários casos de especificações fracas com potencial para serem fortalecidas, permitindo validar a viabilidade da nossa abordagem. Procedemos também à análise do número de mutantes gerados e detetados por operador de mutação, reunindo alguma compreensão acerca de quais são os operadores mais frequentemente aplicáveis em Dafny e mais facilmente detetados por especificações, assim como quais operadores acabam por ser raramente aplicáveis, não constituindo grande contribuição para o nosso objetivo.

Palavras-chave: Verificação de software, Teste de mutação, Dafny

Abstract

Formal verification is essential in critical systems, where failures may have catastrophic consequences. Verification-aware programming languages, such as Dafny, ensure that a program’s implementation aligns with its specification. Nevertheless, while specifications are usually trusted to be correct, programmers can as easily make an error when writing one as when writing an implementation, resulting in programs that verify successfully without capturing the entirety of their intended functionality. Researching techniques to help identify weaknesses in specifications in regards to an implementation that is assumed to capture the programmer’s intent is key to increasing confidence in these systems.

This thesis explores a specification weakness detection methodology grounded in the idea of mutation testing. Treating the specification as a correction criterion comparable to a test suite, mutation analysis, particularly mutated versions of the original program that undergo verification without being detected, can be used to hint at weak specifications. We conduct an in-depth study exploring and evaluating mutation operators suited for the Dafny language, first through a comprehensive survey of operators already proposed for other programming languages, and afterwards through a systematic analysis of bugfix commits in Dafny programs. We combine this knowledge in the development of a mutation testing tool, MutDafny, seamlessly integrated with Dafny’s workflow.

We evaluate MutDafny’s mutant generation capacity and efficiency, as well as specifications’ mutant detection capability in a dataset of 745 programs with about 53,000 lines of code. For this dataset, the tool generates 87,941 mutants, with 71.24% of them being caught by their specifications. We manually analyzed a subset of the undetected mutants and found several weak specifications with potential for strengthening, validating the practical feasibility of our approach. We also analyze the number of generated and detected mutants per mutation operator, gathering some insight on what operators are most commonly applicable in Dafny and are more easily detected by specifications, and which operators end up being rarely applicable, not constituting a great contribution to our goal.

Keywords: Software verification, Mutation testing, Dafny

UN Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) provide a global framework to achieve a better and more sustainable future for all. It includes 17 goals to address the world's most pressing challenges, including poverty, inequality, climate change, environmental degradation, peace, and justice. We reflect on the developed work in the light of these goals.

We believe the higher impact of our research is reflected in the SDGs where the application of critical systems can be involved. This thesis aims to increase confidence in formal methods, particularly formal specifications, which, in turn, improve the reliability of critical systems. Reliable critical systems are essential in the prevention of catastrophic failures in healthcare, infrastructure, and industry, directly supporting the SDGs related to these topics:

SDG 3 Ensure healthy lives and promote well-being for all at all ages.

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

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

SGD	Target	Contribution	Performance Indicators and Metrics
3	3.8	Critical systems applied to health-care services and the development of medicines (eg., MRI and CT scan machines, pacemakers)	Reduction in software-related medical device failures and patient safety incidents
9	9.1	Critical systems applied to resilient infrastructure	Reduction in critical infrastructure downtime due to software failures
	9.b	Critical systems supporting industrial development (eg., robotics control software)	Reduction in production defects
11	11.2	Critical systems associated with the correct functioning of vehicles and traffic control systems	Reduction in accidents caused by vehicle software, in particular regarding autonomous vehicles, and by failures in traffic control systems
	11.5	Critical systems applied to disaster management (eg., flood monitoring, emergency response systems)	Faster and more accurate disaster detection and activation of emergency response measures

Acknowledgements

As I complete this academic journey, I would like to express my deepest appreciation to all those who have supported me during the last five years and have contributed to the completion of this thesis.

First and foremost, my gratitude goes to my supervisors for their guidance and patience, for always being available for discussion and always pushing me to the fullest. To Professor Alexandra Mendes for helping me find a topic that perfectly bridges my interests in formal methods, software testing, and compilers, for her ability to keep me motivated, and for her deep expertise in Dafny. To Professor José Campos for the valuable technical insight in empirical validation, and his attention to detail.

I am incredibly grateful to my parents for shaping me into the person I am today, for teaching me the value of hard work, and encouraging me to be the best version of myself. Thank you for allowing me freedom in choosing my academic path and helping me overcome every setback and stressful period.

To my boyfriend, Miguel, thank you for making my everyday brighter and being my constant support system through every obstacle. Thank you for hiding my phone when I desperately needed to work and for being my personal tech support guy. Above all, thank you for being my favourite person in the world.

Finally, a special thanks to my friends Anete, Milena, and Hugo for being by my side in every project, exam, and coffee break. As we end this journey together, I wish you nothing but the best. Thank you also to my friends Mafalda and Margarida for being my favourite company during every study break.

This work is financed by National Funds through the FCT - Fundação para a Ciência e a Tecnologia, I.P. (Portuguese Foundation for Science and Technology) within the project VeriFixer, with reference 2023.15557.PEX (<https://doi.org/10.54499/2023.15557.PEX>).

Isabel André Amaral

*“Beware of bugs in the above code;
I have only proved it correct, not tried it.”*

Donald E. Knuth

Contents

Resumo	i
Abstract	ii
UN Sustainable Development Goals	iii
Acknowledgements	iv
List of Figures	ix
List of Tables	x
List of Listings	xii
List of Acronyms	xiii
1 Introduction	1
1.1 Context	1
1.2 Motivation	2
1.3 Problem statement	2
1.4 Research questions	3
1.5 Contributions	4
1.6 Document structure	4
2 Literature Review	6
2.1 Relevant literature selection methodology	6
2.2 Relevant literature content analysis	8
2.2.1 Mutation of Dafny programs	8
2.2.2 Other related work	10
2.2.3 Mutation operators for Dafny Programs	11
2.3 Summary	11
3 Existing Mutation Operators	13
3.1 Mutation tools	13
3.2 Mutation operators proposed for other programming languages	14
3.3 Mutation operators proposed for other languages and applicable in Dafny	16
3.3.1 Arithmetic operators	16
3.3.2 Relational Operator Replacement	17

3.3.3	Conditional operators	17
3.3.4	Logical operators	19
3.3.5	Shift Operator Replacement	20
3.3.6	Literal Value Replacement	20
3.3.7	Expression Value Replacement	20
3.3.8	Expression and statement replacement	25
3.3.9	Statement Deletion	27
3.3.10	Variable Deletion	30
3.3.11	Operator Deletion	31
3.3.12	Object-Oriented operators	32
3.4	Mutation operators proposed for other languages but excluded for Dafny	35
3.4.1	Operator replacement operators	35
3.4.2	Expression replacement operators	35
3.4.3	Statement deletion operators	36
3.4.4	Object-Oriented operators	36
3.5	Answer to RQ1, Given the set of mutation operators that have been proposed in the literature, which ones could be applied to Dafny programs?	38
3.6	Summary	38
4	New Mutation Operators for Dafny	39
4.1	Methodology	39
4.2	Synthesized operators	41
4.2.1	Extended Conditional Operator Replacement	41
4.2.2	Expression replacement	42
4.2.3	Argument Swapping	43
4.2.4	Tuple Access Replacement	44
4.2.5	Conditional Block Extraction	45
4.2.6	Statement Swapping	45
4.2.7	Variable Declaration Swapping	46
4.3	Common operations that do not correspond to mutation operators	46
4.4	Answer to RQ2, What mutation operators could be synthesized from bugfixes or previous changes to Dafny programs?	47
4.5	Summary	48
5	Mutation Testing of Dafny Programs	49
5.1	Solution overview	49
5.1.1	MutDafny's mutation operators	51
5.2	Mutation target identification	51
5.3	Mutant generation	53
5.4	Implementation details	55
5.5	Summary	58
6	Empirical evaluation	59
6.1	Experimental subjects	59
6.2	Mutation operators	61
6.3	Experimental procedure	61
6.4	Experimental metrics	64

6.5	Threats to validity	65
6.5.1	Threats to construct validity	65
6.5.2	Threats to internal validity	66
6.5.3	Threats to external validity	67
6.6	Experimental results	67
6.6.1	Answer to RQ3, How efficient is MutDafny at creating mutants, and how many are generated overall and by mutation operator?	67
6.6.2	Answer to RQ4, How do Dafny specifications perform at killing mutants?	72
6.7	Summary	77
7	Conclusion and Future work	78
7.1	Conclusion	78
7.2	Future work	78
	References	80
A	Complete list of records retrieved from literature database search	87
B	Mutation operators proposed in the literature for non-Dafny programs	89
C	Mutation operators synthesized from bugfixes to Dafny programs	99

List of Figures

2.1	Record screening and selection flowchart.	8
5.1	Overview of MutDafny’s pipeline.	50
5.2	Dafny’s workflow simplified.	50
5.3	Example of the mutation target identification flow for the program in Listing 5.1.	52
5.4	Integration of MutDafny’s scanner component in Dafny’s pipeline.	54
5.5	Integration of MutDafny’s mutator component in Dafny’s pipeline.	55
5.6	Example of the flow for the application of a binary operator replacement mutation.	56
5.7	Example of the flow for the replacement of a binary expression with a boolean literal.	56
5.8	Example of the flow for the application of a statement deletion mutation.	57
5.9	Overview of the plugin architecture.	57
6.1	Distribution of number of mutants per program and per mutation operator.	68
6.2	Distribution of total mutant generation runtime per mutation operator.	69
6.3	Distribution of average mutant generation runtime per mutation operator.	70
6.4	Plugin, mutant generation, mutation analysis, and total runtime to perform mutation testing of a Dafny program.	71
6.5	Comparison between the average scan runtime and the overall mutant generation runtime. . .	72
6.6	Ratio of killed mutants per mutation operator.	74
6.7	Mutation score distribution.	75

List of Tables

2.1	Literature related to mutation testing in Dafny. “S” stands for Scopus, “GS” for Google Scholar, “IEEEEX” for IEEE Xplore.	7
2.2	Mutation operators adapted from other programming languages for Dafny mutation tools. . .	11
2.3	Mutation operators specific for Dafny.	11
3.1	Full list of existing mutation testing tools for C++.	14
4.1	DafnyBench repositories not included in study.	40
6.1	Non-verifiable programs in the DafnyBench [40] benchmark.	60
6.2	Programs with mutation analysis and total runtime in the 10^4 order of magnitude.	71
6.3	Number of generated, killed, survived, invalid, and timeout mutants.	73
6.4	Programs with a mutation score of 0%.	75
A.1	Records retrieved from literature database search (see Section 2.1). “S” stands for Scopus, “SD” for ScienceDirect, “GS” for Google Scholar, “IEEEEX” for IEEE Xplore, and “ACM” for ACM Digital Library.	87
B.1	Compilation of mutation operators that have been proposed in a group of selected tools for non-Dafny programs (see Section 3.2). “Yes” means that the operator can be applied to Dafny, “No” means it cannot, “Partially” means that only a subset of the behaviour described by the operator can be applied to Dafny. * Even though the operator can be applied to Dafny, it makes no sense to implement as it always generates programs that are either invalid or equivalent to the original one.	89
C.1	Synthesized mutation operators and commit evidence of their relevance (see Section 4.2). . .	99
C.2	Commit evidence of common fixes that do not translate into mutation operators (see Section 4.3).	102

List of Listings

1.1	Example of a Dafny program with a weak specification.	3
3.1	Example of an arithmetic replacement mutation applied to a Dafny program	16
3.2	Example of an arithmetic division replacement mutation applied to a Dafny program	17
3.3	Example of an arithmetic deletion mutation applied to a Dafny program	17
3.4	Example of an arithmetic insertion mutation applied to a Dafny program	17
3.5	Example of a relational replacement mutation applied to a Dafny program	18
3.6	Example of a replacement mutation of a relational statement with a boolean expression applied to a Dafny program	18
3.7	Example of a conditional replacement mutation applied to a Dafny program	18
3.8	Example of a conditional insertion mutation applied to a Dafny program	19
3.9	Example of a logical replacement mutation applied to a Dafny program	19
3.10	Example of a logical insertion mutation applied to a Dafny program	19
3.11	Example of a shift replacement mutation applied to a Dafny program	20
3.12	Example of a numerical literal replacement mutation (increment) applied to a Dafny program	20
3.13	Example of a string literal replacement mutation applied to a Dafny program	21
3.14	Example of a numerical expression replacement mutation applied to a Dafny program	21
3.15	Example of an empty return expression replacement mutation applied to a Dafny program	22
3.16	Example of a boolean return expression replacement mutation applied to a Dafny program	22
3.17	Example of a return expression null replacement mutation applied to a Dafny program	22
3.18	Example of a lambda expression replacement mutation applied to a Dafny program	22
3.19	Example of a method call replacement mutation applied to a Dafny program	23
3.20	Example of a method call replacement mutation applied to a Dafny program	23
3.21	Example of an argument propagation mutation applied to a Dafny program	24
3.22	Example of an empty array initialization replacement mutation applied to a Dafny program	24
3.23	Example of a non-empty set initialization replacement mutation applied to a Dafny program	25
3.24	Example of a constructor call/null replacement mutation applied to a Dafny program	25
3.25	Example of a break/continue replacement mutation applied to a Dafny program	25
3.26	Example of an invalid break/continue replacement mutation applied to a Dafny program	26
3.27	Example of a loop condition replacement mutation applied to a Dafny program	26
3.28	Example of a match case replacement mutation applied to a Dafny program	27
3.29	Example of an assignment deletion mutation applied to a Dafny program	27
3.30	Example of a method block deletion mutation applied to a Dafny program	28
3.31	Example of an else branch deletion mutation applied to a Dafny program	28
3.32	Example of a case block deletion mutation applied to a Dafny program	28
3.33	Example of a method call deletion mutation applied to a Dafny program	29
3.34	Example of a naked receiver mutation applied to a Dafny program	29
3.35	Example of a field initialization deletion mutation applied to a Dafny program	30

3.36	Example of a constructor deletion mutation applied to a Dafny program	31
3.37	Example of a variable deletion mutation applied to a Dafny program	31
3.38	Example of a subsequence boundary deletion mutation applied to a Dafny program	31
3.39	Example of an operator deletion mutation applied to a Dafny program	31
3.40	Base example of a Dafny OO program	32
3.41	Example of a subclass reference assignment replacement mutation applied to a Dafny program	33
3.42	Example of a this keyword insertion mutation applied to a Dafny program	34
3.43	Example of a this keyword deletion mutation applied to a Dafny program	34
3.44	Example of an accessor method replacement mutation applied to a Dafny program	34
3.45	Example of a modifier method replacement mutation applied to a Dafny program	34
3.46	Example of child/parent declaration type replacement mutation applied to a Dafny program .	37
4.1	Example of a conditional replacement mutation extended to include Dafny-specific operators	41
4.2	Example of a variable expression replacement mutation applied to a Dafny program	42
4.3	Example of a field access replacement mutation applied to a Dafny program	42
4.4	Example of a method call replacement mutation applied to a Dafny program	43
4.5	Example of a datatype constructor replacement mutation applied to a Dafny program	43
4.6	Example of a replacement mutation between a method and a variable applied to a Dafny program	44
4.7	Example of an argument swapping mutation applied to a Dafny program	44
4.8	Example of a tuple access replacement mutation applied to a Dafny program	44
4.9	Example of a conditional block extraction mutation applied to an if Dafny statement	45
4.10	Example of a conditional block extraction mutation applied to an if-then-else Dafny statement	45
4.11	Example of a statement swapping mutation applied to a Dafny program	46
4.12	Example of a variable declaration swapping mutation applied to a Dafny program	46
5.1	Example of a simple Dafny program	52
5.2	List of mutation targets generated for the program in Listing 5.1	53
5.3	Usage of MutDafny's plugin scanning component.	58
5.4	Usage of MutDafny's plugin mutator component for the mutation target listed in Line 1 of Listing 5.2.	58
5.5	Running MutDafny.	58
6.1	Alterations to Dafny's source code to measure plugin's runtime	62
6.2	Alterations to Dafny's source code to measure each execution pipeline's phase runtime . . .	63

List of Acronyms

AMR Accessor Method Replacement	34
AOD Arithmetic Operator Deletion	17
AOI Arithmetic Operator Insertion	17
AOR Arithmetic Operator Replacement	16
AST Abstract Syntax Tree	9
BBR Boolean-Binary Expression Replacement	17
BOR Binary Operator Replacement	51
CBE Conditional Block Extraction	45
CBR Case Block Replacement	27
CIR Collection Initialization Replacement	24
COD Conditional Operator Deletion	18
COI Conditional Operator Insertion	18
COR Conditional Operator Replacement	18
DCR Datatype Constructor Replacement	42
EVR Expression Value Replacement	20
FAR Field Access Replacement	42
LBI Loop Break Insertion	25
LHS Left-Hand Side	34
LINQ Language Integrated Query	35
LLM Large Language Model	10
LOD Logical Operator Deletion	19
LOI Logical Operator Insertion	19
LOR Logical Operator Replacement	19
LSR Loop Statement Replacement	25
LVR Literal Value Replacement	20
MAP Method Argument Propagation	23
MCR Method Call Replacement	42
MMR Modifier Method Replacement	34

MNR Method Naked Receiver	29
MRR Method Return Value Replacement	23
MVR Method-Variable Replacement	43
ODL Operator Deletion	31
OO Object-Oriented	1
PRV Polymorphic Reference Replacement	33
RHS Right-Hand Side	52
ROR Relational Operator Replacement	17
SAR Argument Swapping	43
SDL Statement Deletion	27
SLD Subsequence Limit Deletion	31
SOR Shift Operator Replacement	20
STP Spec-Testing Proof	8
SWS Statement Swapping	45
SWV Variable Declaration Swapping	46
TAR Tuple Access Replacement	44
THI This Keyword Insertion	33
THD This Keyword Deletion	33
UOD Unary Operator Deletion	51
UOI Unary Operator Insertion	51
VER Variable Expression Replacement	42
VDL Variable Deletion	30

Chapter 1

Introduction

1.1 Context

Proving that a program is correct is a particular necessity in critical systems, as failures in these scenarios may have drastic consequences such as loss of vital data, expensive equipment, or even human injury. While traditional software testing ensures that the system meets its requirements for a given group of inputs, it can never fully prove its correctness. Formal verification, on the other hand, is the process of proving an implementation is correct given a mathematical specification that captures the desired behaviour of a system. This is often done through an automated theorem prover that checks whether the specification holds for every possible execution. Design by contract [47] is a program design methodology that consists of providing specifications, expressed in the target programming language, to specific software components. This results in the interaction between the different components functioning as a contract, i.e., a component can only be exerted given proof that the set of preconditions it requires are met, and its execution guarantees that its postconditions are always satisfied.

Big companies such as Amazon are heavily investing in formally verifying their most critical software, e.g., AWS’s Encryption Software Development Kit (SDK)¹ — as the application of encryption is not always straightforward, the use of verification tools is capable of providing a higher degree of confidence that the library holds certain properties. AWS currently leads the development of Dafny [36], an example of a verification-aware programming language.

Dafny is an imperative programming language with support for traditional functional and Object-Oriented (OO) programming concepts, as well as design by contract. Contracts in Dafny are written through the embedding of annotation keywords into the program’s structure. Pre and postconditions are expressed using `requires` and `ensures` clauses, respectively, and `invariant` expressions state properties that remain true throughout iterations of a loop, helping to prove its termination. Additionally, Dafny supports quantifiers to assist in the expression of these properties. Dafny is equipped with a static verifier that attempts to prove the correctness of a program’s implementation based on the specified

¹<https://github.com/aws/aws-encryption-sdk-dafny/>, accessed 24 Jul 2025.

contract, and a compiler to other programming languages, such as Java and C#, facilitating integration with more common development workflows.

Mutation testing [29, 69] is a software testing technique that aims to measure the quality of a test suite in terms of fault detection capability, using the results as feedback for improving it. This is done by creating faulty versions of the original program — called mutants — running them against the test suite, and observing whether it is capable of detecting the inserted fault. If the mutant causes at least one test to fail, this means that the test suite is capable of detecting the fault, and the mutant is considered to be killed. Alive mutants are those that undergo the test suite without being detected, signaling a portion of program behaviour that is not being tested. The mutation score corresponds to the percentage of generated mutants that the test suite is able to kill and is a metric used to quantify its quality. Generally speaking, a higher mutation score increases confidence in the test suite.

Mutants are systematically and automatically generated by applying mutation operators to the original code. A mutation operator is a syntactic rule that performs a specific and small modification to a specific syntactic construct, so that the mutant differs from the original program in a single localized change that mimics a common programming error. Mutants can, however, be equivalent to the original program at times, i.e., though it is written differently, its behaviour is the same.

1.2 Motivation

When employing verification-aware languages, such as Dafny, we have the assurance that the program’s implementation matches the behaviour defined by the specification provided by the programmer. By their nature, specifications are trusted to be correct. However, this is merely an assumption and should not be taken as a given. In fact, it is just as easy for a programmer to commit an error when writing a specification as it is when writing an implementation, resulting in a seemingly correct program that does not match the requirements.

Historically, faults have indeed been found in formally verified systems. A study conducted by Fonseca et al. [17] in 2017 found 16 faults in three different formally verified distributed systems. While some of them were observed to be caused by wrong assumptions between verified and unverified software components, external libraries, for instance, and problems with the verification tools themselves or the auxiliary layers that interact with them, such as scripts, the study also highlighted a specific category of faults caused by incorrect specifications.

1.3 Problem statement

In formal verification we frequently encounter specifications that partially reflect the desired behaviour, but are either too weak — meaning they do not constrain the program’s input and output enough and

allow undesired results — or too strong — if they overly restrict them and therefore exclude valid execution cases. Our work aims to target weak specifications, so in this light, we consider a specification to be weak if it allows the verification of one or more implementations that exhibit undesired behavior.

The Dafny code example shown in Listing 1.1 searches for a given key in an array and returns its index or -1, if the key is not present. The example has a weak specification, as it does not define any behaviour for outputs lower than -1. This particular implementation would be considered correct in any traditional programming language, since it achieves the intended outcome, yet the specification’s range of undefined behaviour would allow one to introduce a fault regarding the program’s return value.

Listing 1.1: Example of a Dafny program with a weak specification.

```

1 method Find(a: array<int>, key: int) returns (index: int)
2 ensures index >= 0 ==> (index < a.Length && a[index] == key)
3 ensures index == -1 ==> (forall k :: 0 <= k < a.Length ==> a[k] != key)
4 {
5     index := 0;
6
7     while index < a.Length
8     invariant 0 <= index <= a.Length
9     invariant forall k :: 0 <= k < index ==> a[k] != key
10    {
11        if (a[index] == key) {
12            return index;
13        }
14        index := index + 1;
15    }
16    index := -1;
17 }

```

We developed a tool, MutDafny, capable of finding potential weaknesses in specifications by inserting mutations in Dafny programs and generating incorrect implementations that are still verified. For example, if MutDafny mutates Line 16 of Listing 1.1 from `index := -1;` to `index := -2;` the program will still be verified. This mutation causes the program to return the wrong value when the key is not found in the array, which is an incorrect implementation as it does not meet the requirements. Dafny’s ability to verify this program despite the inserted mutation hints at a possible problem with the specification. It is important to note that the discovery of an alive mutant will not always indicate the presence of a weakness in the specification. It will rather signal a potential one, prompting the programmer to inspect the mutant further and, only if necessary, strengthen the specification.

1.4 Research questions

In this thesis, we aim to answer the following four research questions:

RQ1: Given the set of mutation operators that have been proposed in the literature, which ones could be applied to Dafny programs?

In this RQ, we aim to investigate (a) whether the most common mutation operators that are

applied to programs in common programming languages, such as Java and C#, can be applied out-of-the-box to Dafny programs and (b) further explore mutation operators that may have already been proposed for Dafny programs.

RQ2: What mutation operators could be synthesized from bugfixes or previous changes to Dafny programs?

This RQ is designed to study, by exploring open source repositories, namely the ones featured in the DafnyBench² [40] benchmark, what errors previously made by Dafny programmers constitute mistakes that would be commonly made in similar contexts and can be synthesised into Dafny-specific mutation operators.

RQ3: How efficient is MutDafny at creating mutants, and how many are generated overall and by mutation operator?

In this RQ we aim to assess the efficiency of MutDafny at generating mutants.

RQ4: How do Dafny specifications perform at killing mutants?

In this RQ we aim to evaluate the effectiveness of Dafny specifications at detecting mutants.

1.5 Contributions

Addressing these research questions has led to the following contributions:

- A compilation of mutation operators proposed for other programming languages and an analysis of their applicability to Dafny.
- A collection of 1,475 bugfix commits in GitHub Dafny programs and a manual analysis of their ability to be synthesized into mutation operators for Dafny.
- The conceptualization of 32 mutation operators for the generation of mutants of Dafny programs, 11 of them novel, not having previously been proposed in mutation testing tools for other programming languages.
- The design and implementation of a mutation testing tool for Dafny programs.
- The detection of a limitation in Dafny’s plugin architecture and an accepted pull request proposing a solution³.
- A mutation analysis empirical study on a dataset of 745 Dafny programs, revealing MutDafny’s ability to generate 87,941 mutants across this corpus.

1.6 Document structure

This report is organized into seven chapters. Chapter 1 introduces the problem we propose to solve. Chapter 2 documents the literature review process and provides the state of the art on mutation of Dafny programs. Chapter 3 explores which mutation operators proposed for other programming languages

²<https://github.com/sun-wendy/DafnyBench>, accessed 24 Jul 2025.

³<https://github.com/dafny-lang/dafny/pull/6144>, accessed 24 Jul 2025.

can be applied to Dafny, and Chapter 4 presents our study of which new mutation operators can be synthesized from faults observed in Dafny programs. Chapter 5 introduces MutDafny’s design and implementation. Chapter 6 details the experimental procedure conducted to evaluate MutDafny’s efficiency and the impact of mutation in Dafny specifications. Finally, Chapter 7 presents this thesis’s conclusions and future work.

Chapter 2

Literature Review

We focus on the analysis of previously existing work related to the application of mutation testing in Dafny. In this chapter, we first outline the methodology adopted for literature collection and selection and, later, describe identified relevant work.

2.1 Relevant literature selection methodology

The literature review was narrowed down to focus on the very core of our research topic, which greatly simplified the search for related work. In fact, we used a single search string — “dafny” “mutation testing” — to find relevant papers in the first phase of the process. This search string retrieves all records containing both the word `dafny` and the expression `mutation testing` at any position within the record. The search was carried out across several data sources — IEEE Xplore¹, ACM Digital Library², Scopus³, ScienceDirect⁴, arXiv⁵, and Google Scholar⁶ — each with a relatively low number of records and Google Scholar with the highest and encompassing the records obtained with the previous data sources. We applied a single search filter, ensuring the exclusion of conference proceedings.

The screening between relevant and non-relevant records was conducted mainly through the title and abstract. In the few cases of records that neither mentioned Dafny nor mutation testing in these fields, we performed a full-text search of the character sequence `muta`, which englobes both the terms `mutation` and `mutant`, aiming to better understand the context in which the study in question used mutation.

Our inclusion criterion required that studies presented solutions that, in some way, dealt with mutation of the Dafny language, either as the main subject of research or simply as a part of the development of the solution to a different problem. As a result, studies focused on other testing forms, for instance compiler testing like XDsmith [26] or automatic test case generation like the work of Gil et al. [21], were

¹IEEE Xplore <https://ieeexplore.ieee.org>, accessed 24 Jul 2025.

²ACM Digital Library webpage <https://dl.acm.org>, accessed 24 Jul 2025.

³Scopus webpage <https://www.scopus.com>, accessed 24 Jul 2025.

⁴ScienceDirect webpage <https://www.sciencedirect.com/>, accessed 24 Jul 2025.

⁵arXiv webpage <https://arxiv.org>, accessed 24 Jul 2025.

⁶Google Scholar webpage <https://scholar.google.com>, accessed 24 Jul 2025.

Table 2.1: Literature related to mutation testing in Dafny. “S” stands for Scopus, “GS” for Google Scholar, “IEEEEX” for IEEE Xplore.

Ref.	Title	Source(s)
[22]	IronSpec: Increasing the Reliability of Formal Specifications	S and GS
[66]	fuzz-d: Random Program Generation for Testing Dafny	GS
[14]	Randomised Testing of the Compiler for a Verification-Aware Programming Language	IEEEEX and GS
[6]	Robustness Testing of Intermediate Verifiers	S and GS
[35]	Evaluating LLM Driven User Intent Formalization for Verification-Aware Languages	GS
[16]	Can Large Language Models Transform Natural Language Intent into Formal Method Postconditions?	GS (and <i>cited by</i> [35])
[19]	Identifying method-level mutation subsumption relations using Z3	<i>Cited by</i> [22]

excluded from our analysis. We were also able to observe that many records consisted of references to mutation testing merely in related or future work, which once again is the case of XDsmith [26], and not always in the context of Dafny, namely EvalPlus [38], Fast [28], and the work by Gil et al. [21]. Finally, some records were excluded because their type was not conducive to review, such as books (e.g., [60]) or individual abstracts (e.g., [34]).

We carried out a second phase of record collection by performing backward and forward snowballing [68], in this order, on all previously included studies. To do this, we inspected the citations of each study and the publications that subsequently cited it, using the databases where each was found. Yet again, we employed a complete analysis, having evaluated all records from both processes without any filtering criteria. By conducting this process in addition to repository keyword searching, we reduced the risk that relevant literature would be omitted from our study. The single study identified in this phase — the one conducted by Gheyi et al. [19] — consists of an exception to our inclusion criterion since it does not deal with Dafny, however, it was deemed relevant due to having been used as a reference to the curation of mutation operators for Dafny in the study that cites it.

As we can see in Figure 2.1, the initial database record collection phase was responsible for yielding a total of 56 records, 38 of which were unique. From these, only five were deemed relevant in the context of our work, and one additional study was identified through the snowballing search process. Although six records were effectively selected, two of these correspond to different publications regarding the same tool, i.e., fuzz-d [14, 66], thus, we are dealing with five different studies. Table 2.1 lists the identified relevant records for both collection phases and the source for each. Table A.1 provides the full list of the 38 unique records retrieved from literature database searching.

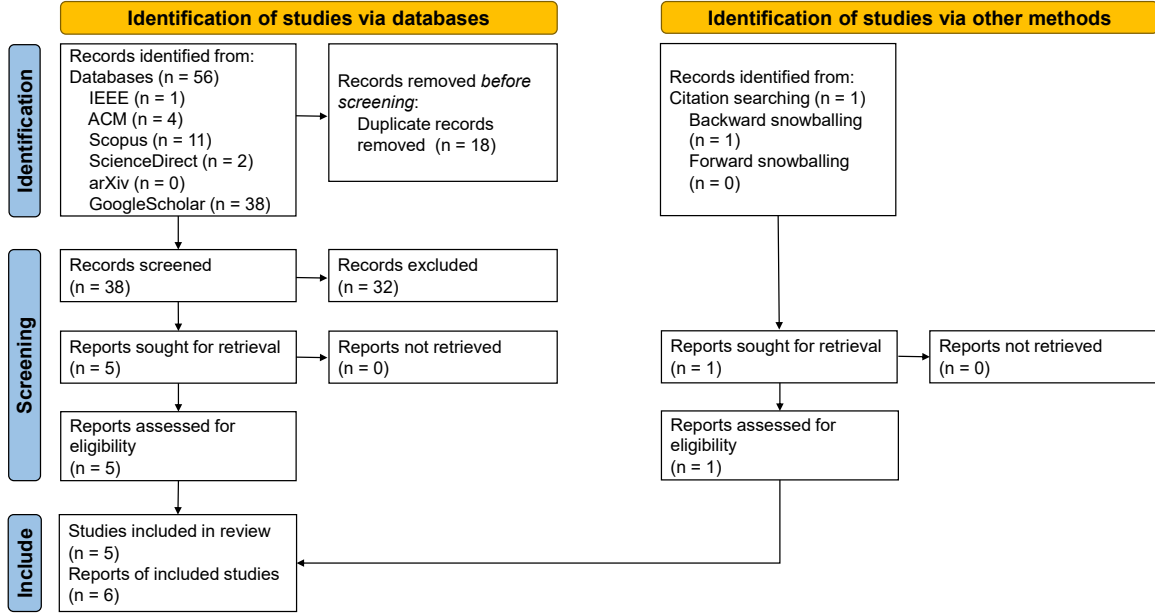


Figure 2.1: Record screening and selection flowchart.

2.2 Relevant literature content analysis

2.2.1 Mutation of Dafny programs

IronSpec⁷ [22] is an approach that, much like MutDafny, aims to increase the reliability of formal specifications. Its tool operates on three different dimensions, described below, each offering different guarantees and complementing each other to identify specification-related issues.

The tool’s *Automatic Sanity Checker* scrutinizes the specifications to assess whether they follow a particular set of properties, whose violation hints at cases where the spec may be weaker than intended. Depending on the severity assigned to each property violation, there is a higher or lower probability that the weakness may lead to the verification of a faulty implementation. High severity flags occur when none of the postconditions depend on the input or when they do not constrain the output. This particular component works in the absence of an implementation and, therefore, allows for the quick identification of possible specification problems.

IronSpec also includes a methodology for writing Spec-Testing Proofs (STPs). The concept is inspired by the one of unit testing, considering that STPs allow to prove that manually chosen inputs and outputs are valid or invalid in the current spec, enabling developers to compare the latter to their own understanding of the program’s functionality.

Particularly relevant to our work is IronSpec’s *automated specification mutation testing* feature. IronSpec performs mutation of methods’ specifications and evaluates whether the implementation is

⁷<https://github.com/GLaDOS-Michigan/IronSpec>, accessed 24 Jul 2025.

still verified. To be more precise, IronSpec only considers mutated specs that are stronger than the original ones. Even if the existence of implementation errors is a very plausible scenario, they consider implementations as an expression of the programmer’s intent, i.e., the intended functionality of the program, using them to discover cases in which the specification is weaker than the implementation and can potentially be strengthened. The identification of these cases corresponds to live mutations, which occur when the same implementation that was proven by the original spec is also proven by a stronger one.

Specification mutants are generated by applying one single mutation operator in the Abstract Syntax Tree (AST) of a Dafny program. After generation, mutants are killed over three different phases which gradually discard mutants according to different criteria. Since IronSpec is only interested in analyzing mutated specifications that are stronger than the original, the first stage involves discarding logically redundant generated specs, which are the ones logically equivalent or weaker than the original. This is done by automatically generating a lemma that evaluates the logical implication between the original and mutated spec—given postconditions S and S' , $S \Rightarrow S'$ means S' is at least as weak as S and, therefore, can be discarded. The second stage entails identifying mutants that cause vacuity, in other words, that cause the proof to pass trivially due to predicates that always evaluate to `true`. Finally, the system is re-verified with the mutated specification and live mutants are identified by full passing proofs.

Alive mutations only hint at a possible disconnect between the spec and the implementation. The final decision as to whether it constitutes a case that requires strengthening of the spec is made by the programmer. STPs can be used to arrive at a counterexample proof of undesired behaviour based on an alive mutation.

IronSpec was evaluated on the specifications of 14 Dafny methods, eight of them with artificially introduced faults and the remaining six from real-world code bases: QBTF⁸, a verified distributed validator protocol⁹, TrueSAT¹⁰ [1], Eth2.0¹¹, DaisyNFS¹² [5], and AWS’s Encryption SDK library¹³. They were able to uncover 10 specification faults in the open-source projects, all confirmed by their owners, across the use of the three different components of the tool. The *Automatic Sanity Checker* led to the discovery of five faults, the writing of STPs led to the discovery of another three, and the final two were hinted at by the mutation testing component.

Another tool, fuzz-d¹⁴ [14, 66], also deals with simple mutations of Dafny specification constructs. The purpose of this tool is to test and validate both the Dafny compiler and verifier by generating valid programs and passing them to Dafny’s components as test cases. Subsequent to this process, the fuzz-d pipeline describes an additional mutator module, exclusive to the testing of the verifier. The empirical

⁸<https://github.com/Consensys/qbft-formal-spec-and-verification>, accessed 24 Jul 2025.

⁹<https://github.com/Consensys/distributed-validator-formal-specs-and-verification>, accessed 24 Jul 2025.

¹⁰<https://github.com/andricicezar/sat-solver-dafny>, accessed 24 Jul 2025.

¹¹<https://github.com/Consensys/eth2.0-dafny>, accessed 24 Jul 2025.

¹²<https://github.com/mit-pdos/daisy-nfsd>, accessed 24 Jul 2025.

¹³<https://github.com/aws/aws-encryption-sdk-dafny/>, accessed 24 Jul 2025.

¹⁴<https://github.com/fuzz-d>, accessed 24 Jul 2025.

evaluation of the tool succeeded in identifying 18 distinct faults and submitting 14 new bug reports to Dafny developers.

After Dafny correctly verifies one of fuzz-d’s test cases, the tool’s mutator component, in turn, generates another set of programs, in this case invalid, by mutating assertions in the current program. This is carried out with the intention of ensuring the verifier correctly identifies cases in which the program should remain unverified, adding yet another layer of testing to the tool.

2.2.2 Other related work

Like fuzz-d [14, 66], μ gie^{15 16} [6] is also a tool developed to test verifiers, however, in this case, intermediate verifiers such as Boogie [2], used by Dafny as well as other verification tools. Verifiers tend to exhibit brittle behaviour, i.e., drastic changes in performance with small changes of the input program that ought to be inconsequential, for example changing the order of program statements. μ gie uses mutation of Boogie programs to identify instances of lack of verifier robustness caused by this volatile behaviour.

μ gie starts from a seed program known to be correct and verified, and the test cases are generated by mutating it. A key difference between μ gie and fuzz-d is that the former generates equivalent mutants, which differ from the seed program in how they are written yet the semantics remain the same. Therefore, the mutation operators used by the tool are curated with this constraint in mind. Some examples include swapping procedure or local variable declarations, splitting the declaration of multiple variables into multiple declarations, and joining two or more pre or postconditions into a conjunctive one.

Given that the mutants are semantically equivalent to each other and to the verified seed program, all of them should be verified as well. The failure to verify a mutant identifies a fault in the verifier in the form of brittle behaviour.

Mutation of Dafny programs is also explored in the context of the problem of user-intent formalization through the use of Large Language Models (LLMs). Lahiri [35]¹⁷ conducted a study with the aim of defining automated metrics to measure the quality of LLM-generated specifications and developed correctness and completeness metrics. Correctness is measured in reference to a set of validation tests that are hidden from the LLM. Completeness refers to the ability of a specification to distinguish between a possible correct implementation and other incorrect ones. The latter is crucial to evaluate a specification’s quality since correction alone can not detect weak or even vacuous specifications.

The generation of mutations plays a central role in measuring completeness. Lahiri [35] mutates the outputs of validation tests, which are implemented by embedding input and output values in Dafny programs, hence mutating Dafny programs. However, mutations consist solely of simple replacement of literal values.

¹⁵<https://emptylambda.github.io/mu-gie/>, accessed 24 Jul 2025.

¹⁶<https://github.com/emptylambda/mu-gie>, accessed 24 Jul 2025.

¹⁷<https://github.com/microsoft/nl-2-postcond>, accessed 24 Jul 2025.

Table 2.2: Mutation operators adapted from other programming languages for Dafny mutation tools.

Operator	Description	Tool(s)
Arithmetic Operator Replacement (AOR)	Replacement of an arithmetic operator with another	IronSpec [22]
Relational Operator Replacement (ROR)	Replacement of a relational operator with another	IronSpec [22], fuzz-d [14, 66]
Logical Operator Replacement (LOR)	Replacement of a logical operator with another	IronSpec [22]
Constant Operator Insertion (COI)	Replacement of numerical literals with their increment/decrement	IronSpec [22]
Expression Negation Operator (ENO)	Insertion/deletion of the <code>!</code> operator in front of boolean expressions	IronSpec [22]
Unary Operator Replacement (UOR)	Insertion/deletion of unary operators such as collection's cardinality operators	IronSpec [22]
Variable Name Operator Replacement (VNOR)	Replacement of an identifier with one of the arguments of its scope of the same type	IronSpec [22]

Table 2.3: Mutation operators specific for Dafny.

Operator	Description	Tool
Heap Operator Replacement (HOR)	Replacement of expressions containing the keyword <code>old</code> with its argument	IronSpec [22]
Set Operator Replacement (SOR)	Replacement between <code>in</code> and <code>!in</code>	IronSpec [22]

2.2.3 Mutation operators for Dafny Programs

IronSpec supports a set of operators (see Table 2.2), commonly used in popular programming languages, inspired by MuJava [42] and a study conducted by Gheyi et al. [19]. Gheyi et al. [19] investigated subsumption relations among MuJava's method-level operators, identifying redundant ones. A mutation M1 is said to subsume another M2 when the set of tests that kill the mutants generated from M1 is a subset of the set of tests that kill the mutants generated by M2. This means that M2 will be killed by any test that kills M1 and there is no need to generate it. Additionally, IronSpec proposes two new operators specific for the Dafny language, listed in Table 2.3.

fuzz-d's mutations correspond to elementary operations that aim to transform valid assertions into invalid ones. One way the tool does this is by performing replacement between the `==` and `!=` operators, and alternatively by replacing entire expressions with `false`. Both of these operators fit into the category of Relational Operator Replacement (ROR).

2.3 Summary

From the identified relevant literature, only IronSpec [22] implements mutations in the context of finding specification faults in Dafny programs. fuzz-d [14, 66] and μ gie [6] implement mutations in the context

of testing Dafny’s internal flow itself, and Lahiri [35], while indeed looking to assess the quality of LLM-generated specifications, does not implement a traditional deterministic mutation generation algorithm, which we argue is not enough in a mutation testing context due to the likelihood of results varying between executions.

IronSpec is thus highlighted as the work that aligns more closely with the tool we propose, both regarding the problem that is aimed to address and the adopted approach. Even so, its functionality is limited to only mutating program’s specifications and the use of a reduced set of mutation operators. MutDafny proposes to fill this gap by conducting a deeper investigation into mutation operators tailored for the Dafny language and implementing program mutation, rather than specification mutation.

Chapter 3

Existing Mutation Operators

Mutation testing has become a popular research topic, with extensive literature and numerous tools for a wide range of programming languages. In this chapter, we are interested in assembling a compilation of the mutation operators proposed in other programming languages, intending to gather knowledge that will allow us to answer **RQ1**: *Given the set of mutation operators that have been proposed in the literature, which ones could be applied to Dafny programs?*

3.1 Mutation tools

Given the extensive number of programming languages with support for mutation testing and the overall number of available tools, it was necessary to perform a selection of languages and respective tools to consider in our analysis, in order to make it feasible. With this in mind, we deemed it relevant to focus our study on the target languages of the Dafny compiler — C#, C++, Java, JavaScript, Go, and Python — and, for each of them, on a subsection of tools.

Sánchez et al. [62] conducted a recent study of the use of mutation testing in practice, having carried out a compilation of existing mutation testing tools, 127 in total, and a GitHub search for evidence of their use. Their findings identify the most popular tools and their main areas of usage, highlighting that some are more adopted in teaching and research while others are more widely applied in development. They derive a list of the 10 most used mutation testing tools by pinpointing the ones imported by a higher number of GitHub repositories:

- Humbug¹ and Infection² for PHP.
- Major³ [30, 31], MuJava⁴ [41, 42, 43, 44], and PIT⁵ [10] for Java.
- Mutant⁶ for Ruby.

¹<https://github.com/humbug/humbug>, accessed 24 Jul 2025.

²<https://infection.github.io/>, accessed 24 Jul 2025.

³<https://mutation-testing.org>, accessed 24 Jul 2025.

⁴<https://github.com/jeffoffutt/muJava>, accessed 24 Jul 2025.

⁵<http://pitest.org>, accessed 24 Jul 2025.

⁶<https://github.com/mbj/mutant>, accessed 24 Jul 2025.

Table 3.1: Full list of existing mutation testing tools for C++.

Tool	Number of GitHub repositories
mutate	1
Proteum	3
mutgen	N/A
ESTP	N/A
Milu	7
SMT-C	N/A
CCMUTATOR	0
MuVM	N/A
MuCPP	6
MuTomVo	N/A
MUSIC	1
Mull	11

- Mutmut⁷ and MutPy⁸ [13] for Python.
- StrykerJS⁹ for JavaScript.
- Stryker.NET¹⁰ for C#.

This list allowed us to narrow down the set of tools for further examination. Additionally, we also wished to include some representatives for the C++ and Go languages, aiming to cover all of the languages for which Dafny compiles to.

To do so, we went back to the full list of the 127 tools identified and filtered the ones for these two languages. According to this list, there is only one mutation testing tool for Go, Go-Mutesting^{11 12}, therefore that is where our analysis for this language resides. However, when it comes to C++, 12 tools were identified (see Table 3.1). Consequently, we inspected the supplementary information provided by Sánchez et al. [62] to see how many GitHub repositories had been found for each of these tools and, by employing the same metric used to find the top 10 tools, we were able to single out Mull¹³ [12] as the most popular option for C++.

3.2 Mutation operators proposed for other programming languages

Given the tools listed in Section 3.1, we analyzed and collected the set of mutation operators supported by each. This task involved reading the papers that proposed each tool, when these existed, examining

⁷<https://github.com/boxed/mutmut>, accessed 24 Jul 2025.

⁸<https://github.com/mutpy/mutpy>, accessed 24 Jul 2025.

⁹<https://stryker-mutator.io/docs/stryker-js/introduction/>, accessed 24 Jul 2025.

¹⁰<https://stryker-mutator.io/docs/stryker-net/introduction/>, accessed 24 Jul 2025.

¹¹<https://github.com/zimmski/go-mutesting>, accessed 24 Jul 2025.

¹²<https://github.com/avito-tech/go-mutesting>, accessed 24 Jul 2025.

¹³<https://github.com/mull-project/mull>, accessed 24 Jul 2025.

their documentation, and, specifically for Mutmut⁷, since no documentation was found, it was necessary to study the tool's source code in order to discover what their implemented set of operators was. Table B.1 lists all 147 mutation operators identified across the nine selected mutation tools.

Throughout this process, we recognized the existence of many similar operators among the different tools and languages. Even so, almost every tool uses different names for common operators. We attempt to group the operators in Table B.1 according to our perceived similarity between them. We can highlight the following groups of operators:

- Operator replacement, which applies to the binary operators in each subgroup:
 - Arithmetic, e.g., +, -, *, /, %
 - Relational, e.g., ==, !=, <, <=, >, >=
 - Conditional, e.g., &&, ||, &, |, ^
 - Logical, e.g., &, |, ^
 - Shift, e.g., <<, >>, >>>
 - Assignment, e.g., +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=
- Operator insertion and deletion, which apply to the unary operators in each subgroup:
 - 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, both unary and binary, and because it involves the deletion of all occurrences of a specific operator.
- OO specific:
 - Inheritance
 - Polymorphism
 - Overloading

The set of operators listed as an example for the arithmetic, relational, conditional, logical, shift, and assignment subgroups are the ones that exist specifically in the Java language and their usage in the mutation context is detailed in MuJava [42]. Most of them are common to the majority of programming languages, even if represented with a different syntax. Nonetheless, it is important to notice that other languages may not support some of these operators, or they may support a few additional ones in each subgroup. For instance, Python also supports the `**` and `//` arithmetic operators. Furthermore, some mutation tools classify the `&&` and `||` operators in the logical subgroup instead of the conditional. Considering these cases correspond to the minority, throughout this work we will always refer to them as conditional operators.

3.3 Mutation operators proposed for other languages and applicable in Dafny

From the compiled total of 147 mutation operators listed in Section 3.2, we identified 82 of them as being applicable to the mutation of Dafny programs, with another eight being partially useful. Seven of them can be applied to Dafny programs but make no sense as mutation operators, due to their application in Dafny always generating equivalent mutants, and 50 cannot be applied to the syntax structure of Dafny 4.10.0. Below, we present how each unique operator can be applied to Dafny and any restrictions that may exist in its application. Additionally, we provide concrete examples of mutations of Dafny programs, representing insertions in green and deletions in red.

3.3.1 Arithmetic operators

Dafny provides five arithmetic binary operators (+, −, *, /, %) and a single unary operator (−). Shortcut arithmetic operators (++ , --), though a common mutation target in other programming languages, are not available in Dafny.

3.3.1.1 Arithmetic Operator Replacement (AOR)

Arithmetic replacement operators proposed for other programming languages are typically applied between all of the arithmetic binary operators existing in a language, i.e., each arithmetic operator can be replaced by any of the remaining existing ones. However, Dafny’s / and % operators have the particularity of the verifier requiring a precondition stating that the operation’s divisor is not zero, to ensure a division by zero error does not occur. Considering it is unlikely that this precondition will be present in a method that originally does not deal with divisions, replacement of any of the other operators, for instance +, with one of these, such as /, should, most of the times, throw a verification error.

Another approach adopted by some tools that can help us overcome this problem consists of restricting which operators can be replaced by which. In Dafny, this could reflect in allowing free replacement between the +, −, * group of operators, as depicted in Listing 3.1 and only allowing / to be replaced with % and vice versa, as reproduced in Listing 3.2.

Listing 3.1: Example of an arithmetic replacement mutation applied to a Dafny program

```

1 method Sum(a: int, b: int) returns (c: int)
2 {
3 -   c := a + b;
4 +   c := a - b;
5 }
```

Replacement among the unary arithmetic operators available in a programming language is commonly implemented in mutation testing tools as well. In Dafny, though, only one unary operator exists (−), making it impossible to implement unary replacement.

Listing 3.2: Example of an arithmetic division replacement mutation applied to a Dafny program

```

1 method Divide(a: int, b: int) returns (c: int)
2 {
3   - c := a / b;
4   + c := a % b;
5 }

```

3.3.1.2 Arithmetic Operator Insertion (AOI) and Arithmetic Operator Deletion (AOD)

Operator insertion and deletion only apply to unary operators so, for Dafny, it could be possible to implement insertion and deletion of the `-` operator. Listing 3.3 shows that it is possible to delete the operator and still get a valid program and Listing 3.4 exemplifies a valid operator insertion.

Listing 3.3: Example of an arithmetic deletion mutation applied to a Dafny program

```

1 method Negation(a: int) returns (b: int)
2 {
3   - b := -a;
4   + b := a;
5 }

```

Listing 3.4: Example of an arithmetic insertion mutation applied to a Dafny program

```

1 method Abs(x: int) returns (abs: int)
2 {
3   if x >= 0 {
4     - abs := x;
5     + abs := -x;
6   } else {
7     abs := -x;
8   }
9 }

```

3.3.2 Relational Operator Replacement (ROR)

Dafny supports the six relational operators that are common to most languages (`==`, `!=`, `<`, `<=`, `>`, `>=`) all of which are binary, hence only replacement mutation operations can be performed. This group of operators does not impose any replacement restrictions, so for each of them, any of the remaining constitutes a valid replacement. Listing 3.5 shows an example. Additionally, many tools implement replacement of statements that use these expressions with `true` or `false`, which is also possible in Dafny, as illustrated in Listing 3.6. Henceforth, we denote this second operator Boolean-Binary Expression Replacement (BBR). It is also possible to employ relational replacement in the context of ternary statements, such as `m := if x != 0 then x / 10 else 0;`.

3.3.3 Conditional operators

Dafny includes three conditional operators, two binary (`&&`, `||`) and one unary (`!`).

Listing 3.5: Example of a relational replacement mutation applied to a Dafny program

```

1 method Abs(x: int) returns (abs: int)
2 {
3   - if x >= 0 {
4   + if x == 0 {
5       abs := x;
6   } else {
7       abs := -x;
8   }
9 }

```

Listing 3.6: Example of a replacement mutation of a relational statement with a boolean expression applied to a Dafny program

```

1 method Abs(x: int) returns (abs: int)
2 {
3   - if x >= 0 {
4   + if true {
5       abs := x;
6   } else {
7       abs := -x;
8   }
9 }

```

3.3.3.1 Conditional Operator Replacement (COR)

Each conditional binary operator can be replaced by its conditional alternative, as exemplified in Listing 3.7. Much like what is done for relational operators, many mutation tools replace conditional statements with `true` or `false`, constituting another application case of the **BBR** operator. Dafny allows implementing this at both the relational and conditional levels.

Listing 3.7: Example of a conditional replacement mutation applied to a Dafny program

```

1 method IsBetween(a: int, b: int, c: int) returns (res: bool)
2 {
3   - if (c > a && c < b) {
4   + if (c > a || c < b) {
5       res := true;
6   } else {
7       res := false;
8   }
9 }

```

3.3.3.2 Conditional Operator Insertion (COI) and Conditional Operator Deletion (COD)

Conditional operator insertion and deletion can only be applied to `!`, Dafny's single unary conditional operator. Listing 3.8 illustrates an insertion mutation. A deletion would consist of the inverse action.

Listing 3.8: Example of a conditional insertion mutation applied to a Dafny program

```

1 method IsBetween(a: int, b: int, c: int) returns (res: bool)
2 {
3 -   if (c > a && c < b) {
4 +   if (!(c > a && c < b)) {
5       res := true;
6   } else {
7       res := false;
8   }
9 }

```

3.3.4 Logical operators

Logical operators are the ones that perform bitwise operations. Dafny's binary logical operators are `&`, `|` and `^`. Additionally, there is one unary operator, `!`, which, being equivalent to the `~` operator in Java, returns the logical complement of a bit-vector type. It is important to note that, while the `!` operator is both considered a conditional and a logical operator, it is used in different contexts, one being its application to a boolean expression and the other to a bit-vector type, returning different types accordingly.

3.3.4.1 Logical Operator Replacement (LOR)

Given the existence of one single unary logical operator, it is not possible to implement logical unary replacement. Nevertheless, each of the binary operators can be replaced interchangeably to generate mutations. An example is shown in Listing 3.9.

Listing 3.9: Example of a logical replacement mutation applied to a Dafny program

```

1 method BitwiseAnd(a: bv4, b: bv4) returns (result: bv4)
2 {
3 -   c := a & b;
4 +   c := a ^ b;
5 }

```

3.3.4.2 Logical Operator Insertion (LOI) and Logical Operator Deletion (LOD)

Any bit-vector expression in Dafny can be mutated by inserting the `!` operator in front of it, as done in Listing 3.10, or by removing it, in cases where it is already being used.

Listing 3.10: Example of a logical insertion mutation applied to a Dafny program

```

1 method BitwiseAnd(a: bv4, b: bv4) returns (result: bv4)
2 {
3 -   c := a & b;
4 +   c := !a & b;
5 }

```

3.3.5 Shift Operator Replacement (SOR)

Like logical operators, shift ones operate over bit-vector types. When it comes to binary shift operators available in Dafny, we have two ($\ll$, $\gg$), their replacement mutation exemplified in Listing 3.11. These differ from the previously presented binary operators since one of the operands acts as an argument. Since there are no unary operators, insertion and deletion is not possible.

Listing 3.11: Example of a shift replacement mutation applied to a Dafny program

```

1 method ShiftNBits(a: bv4, n: nat) returns (b: bv4)
2   requires n <= 4
3   {
4     - b := a << n;
5     + b := a >> n;
6   }

```

3.3.6 Literal Value Replacement (LVR)

Literal value replacement operators are traditionally applied to numerical, boolean, and string literals, and the transformations applied to each vary greatly among the different tools. The simplest case is the boolean literal replacement, for which the only possible replacement is the logical complement of the literal being used, i.e., replacement of `true` with `false` and vice versa. For string literals, it is common to replace non-empty strings with empty ones or replace any string with default values depending on the tool (e.g., “mutpy”, “python”, etc.). Furthermore, some tools even alter the original string by inserting random characters at specific positions (e.g., insertion of “XX” after the first character and before the last character). When it comes to numerical literals, these are often incremented or decremented, replaced by their negative/positive version or simply replaced by pre-defined static values, such as 0, 1, -1, etc.

We found no restrictions to literal value replacement in Dafny, as long as the original types are still respected. Even so, we consider the replacement between boolean literals to be equivalent to the application of a **LOI** operator, i.e., the insertion of a `!` operator, in front of the original literal. Listing 3.12 showcases an example of numerical literal replacement and Listing 3.13 of string literal replacement. These are just two examples out of the many mutations that would be possible for each of the programs.

Listing 3.12: Example of a numerical literal replacement mutation (increment) applied to a Dafny program

```

1 method Double(a: int) returns (b: int)
2   {
3     - b := a * 2;
4     + b := a * 3;
5   }

```

3.3.7 Expression Value Replacement (EVR)

Given the broad nature of what constitutes an expression in a program, there is a wide range of expression value replacement operators across the analyzed tools. Some operators consist of simpler replacements,

Listing 3.13: Example of a string literal replacement mutation applied to a Dafny program

```

1 method AppendSuffix(s: string) returns (res: string)
2 {
3   - var suffix := "example_suffix";
4   + var suffix := "";
5   res := s + suffix;
6 }

```

to numerical, boolean, or other primitive-typed expressions, anywhere within the program, while others are more specific in their application context. Namely, some operators are applied specifically to return values and collections, among other contexts. The simplest form of replacement involves substituting an expression with a literal value, as done in Listing 3.14.

Listing 3.14: Example of a numerical expression replacement mutation applied to a Dafny program

```

1 method Sum(a: int, b: int) returns (c: int)
2 {
3   - c := a + b;
4   + c := a + 0;
5 }

```

3.3.7.1 Return value replacement

As subsets of the **EVR** operator, we have others that only target specific types of syntactic constructs. Here, we present a group of operators that mutate return statements. The most general case of this operator is the replacement of a return value, of any type, with a literal of the same type. In addition to this elementary operation, PIT [10] also implements a group of operators whose functionality is restricted to return statements of specific types. One operator replaces return values with the default for the type being used (e.g., 0 when dealing with a numerical return type, an empty string, etc.), another replaces return values solely with `null`, and two others only perform mutation of boolean return statements, replacing them with `true` or `false`.

All of these replacements can be applied to Dafny programs, with the particularity that Dafny's `null` keyword can only be used in certain contexts. One limitation is the fact that the value `null` can only be used in the context of Dafny's reference types, such as arrays and class objects. Therefore, this mutation can only be applied to methods that have reference types as their original return value. Another constraint is that a return statement can only validly take this value if the method's return type is annotated with a `?` (e.g., `array?<int>`), which in Dafny means that a value can either be nullable or adhere to the specified type.

Listing 3.15 and Listing 3.16 constitute, respectively, examples of *Empty Returns* and *False Returns* mutations, both of which are included in a general return value operator. Listing 3.17 depicts a mutation generated by the *Null Returns* operator. It is important to notice the `?` in the mutated method's return type, which is a reference type. The same mutation would not be valid in either Listing 3.15's or Listing 3.16's original programs, since integers and booleans are neither reference types nor nullable.

Listing 3.15: Example of an empty return expression replacement mutation applied to a Dafny program

```

1 method Sum(a: int, b: int) returns (c: int)
2 {
3   c := a + b;
4 - return c;
5 + return 0;
6 }

```

Listing 3.16: Example of a boolean return expression replacement mutation applied to a Dafny program

```

1 method IsBetween(a: int, b: int, c: int) returns (res: bool)
2 {
3 - return c > a && c < b;
4 + return false;
5 }

```

Listing 3.17: Example of a return expression null replacement mutation applied to a Dafny program

```

1 method CreateNArray(n: nat) returns (a: array<int>)
2 {
3   a := new int[n];
4   var i := 0;
5
6   while i < n
7   {
8     a[i] := i + 1;
9     i := i + 1;
10  }
11 - return a;
12 + return null;
13 }

```

In Python, return value mutations are also applied to the return of lambda expressions. This concept also exists in Dafny, though, as with the remaining operators, only replacements with the same type as the original can be made. Listing 3.18 shows a possible mutation to a lambda function with an integer return type.

Listing 3.18: Example of a lambda expression replacement mutation applied to a Dafny program

```

1 method IncrementNumbers(numbers: array<int>) returns (incrementedNumbers: array<int>)
2 {
3 - var increment := x => x + 1;
4 + var increment := x => 0;
5
6   incrementedNumbers := new int[numbers.Length];
7   var i := 0;
8   while i < numbers.Length {
9     incrementedNumbers[i] := increment(numbers[i]);
10    i := i + 1;
11  }
12 }

```

3.3.7.2 Method call replacement

Another type of expression replacement can be that of method calls. Other tools replace the method call with a default value with the same type as the method's return type, as illustrated in Listing 3.19. Since this operates over methods' return values, this operator is only applicable to non-void methods. In Dafny, methods may return a set of values, instead of just one, as exemplified in Listing 3.20, making the operation slightly more complex. Due to this fundamental difference to the EVR operator we consider this a separate operator, the Method Return Value Replacement (MRR).

Listing 3.19: Example of a method call replacement mutation applied to a Dafny program

```

1 method Abs(x: int) returns (abs: int)
2 {
3   abs := 0;
4   if x > 0 {
5     abs := x;
6   } else if x < 0 {
7     abs := -x;
8   }
9 }
10
11 method Main()
12 {
13 - var n := Abs(-3);
14 + var n := 0;
15 }

```

Listing 3.20: Example of a method call replacement mutation applied to a Dafny program

```

1 method IntegerDivision(a: int, b: int) returns (q: int, r: int)
2 requires a > 0
3 requires b > 0
4 {
5   q := 0;
6   r := a;
7
8   while r >= b {
9     q := q + 1;
10    r := r - b;
11  }
12 }
13
14 method Main()
15 {
16 - var q, r := IntegerDivision(9, 3);
17 + var q, r := 0, 0;
18 }

```

A more specific type of the method call replacement operator is the Method Argument Propagation (MAP), first implemented by PIT [10], which consists of the replacement of a full method call with one of its arguments. Independently of the language, the type of the argument selected to replace the

expression must match the method’s return type so that program validity is kept. Listing 3.21 exemplifies this replacement in an **OO** context, but in Dafny this can also be accomplished with regular methods that are not class members, such as the one being called in Listing 3.19.

Listing 3.21: Example of an argument propagation mutation applied to a Dafny program

```

1 class Item {
2   var value: int
3
4   constructor(value: int)
5   {
6     this.value := value;
7   }
8
9   method IncreaseValue(n: int) returns (v: int)
10  modifies this
11  {
12    this.value := this.value + n;
13    return this.value;
14  }
15 }
16
17 method CreateAndUpdateItem(value: int)
18 {
19   var newItem := new Item(value);
20 - var itemValue := newItem.IncreaseValue(3);
21 + var itemValue := 3;
22 }

```

3.3.7.3 Object initialization replacement

This category groups operators that replace several object initialization mechanisms. On the one hand, we have the Collection Initialization Replacement (**CIR**) operator handling the initialization of different types of collections, each with its own construction syntax. Stryker.NET implements this for C#’s arrays, lists, collections, and dictionaries. Some of these, such as lists and dictionaries, do not exist in Dafny or have different designations. However, the concept can be employed in any of Dafny’s collection types, even the ones that do not exist in C# and are considered in Stryker.NET. Namely, it is possible to implement this mutation in Dafny’s arrays and sets, as done in Listing 3.22 and Listing 3.23 respectively, multisets, sequences, and maps. We also consider this operator separately from the **EVR** because, besides replacing collection initializations with empty (default) ones, it replaces empty collection initializations with standard non-empty ones.

Listing 3.22: Example of an empty array initialization replacement mutation applied to a Dafny program

```

1 method CreateArray() returns (a: array<int>)
2 {
3 - a := new int[3][1, 2, 3];
4 + a := new int[3];
5 }

```

Listing 3.23: Example of a non-empty set initialization replacement mutation applied to a Dafny program

```

1 method CreateSet() returns (s: set<int>)
2 {
3 - s := {};
4 + s := {1, 2, 3};
5 }

```

On the other hand, we have class instances, whose initialization through a constructor call can be replaced with `null`, as presented in Listing 3.23. This operation, being a simple replacement, is viewed as a part of the **EVR** operator.

Listing 3.24: Example of a constructor call/null replacement mutation applied to a Dafny program

```

1 method CreateItem(value: int)
2 {
3 - var newItem := new Item(value);
4 + var newItem := null;
5 }

```

3.3.8 Expression and statement replacement

In Section 3.3.7, we introduced the replacement of expressions with literal values. Some operators go further by replacing expressions or statements with different expressions or statements.

3.3.8.1 Loop constructs replacement

This category of operators deals with mutation of constructs that exist specifically in the context of loops, such as the `break` and `continue` keywords and loop conditions. Listing 3.25 exemplifies a valid replacement of a `break` keyword with a `continue` one in Dafny. The inverse is also possible, as well as the replacement with a `return` statement, in the case of the enclosing method being `void`. We will refer to this mutation operator as Loop Statement Replacement (**LSR**). A different type of mutation, the Loop Break Insertion (**LBI**), involves the insertion of a `break` keyword at the beginning of the loop.

Listing 3.25: Example of a break/continue replacement mutation applied to a Dafny program

```

1 method Find(a: array<int>, key: int) returns (index: int)
2 {
3   index := -1;
4
5   for i := 0 to a.Length
6     invariant 0 <= i <= a.Length
7     {
8       if a[i] == key {
9         index := i;
10 -      break;
11 +      continue;
12     }
13   }
14 }

```

Listing 3.26: Example of an invalid break/continue replacement mutation applied to a Dafny program

```

1 method Find(a: array<int>, key: int) returns (index: int)
2 {
3     index := 0;
4
5     while index < a.Length
6     invariant 0 <= index <= a.Length
7     {
8         if (a[index] == key) {
9 -         break;
10 +         continue;
11         }
12         index := index + 1;
13     }
14 }

```

Even though it is possible to create valid examples for all of these mutations, Dafny’s verifying nature makes it easy to create an invalid loop when using invariants. Listing 3.26 presents a program with the same functionality as the one in Listing 3.25, using a `while` instead of a `for` loop. Dafny can verify the version with the `for` loop but not the other one. This happens because, in the latter, loop termination is dependent on a counter variable that is manually incremented at each iteration and, by executing a `continue` statement before the increment happens, Dafny can no longer prove termination. In fact, this mutation would generate an infinite loop. Based on this example, we can conclude that the validity of mutations involving loop-specific keywords greatly depends on the way the program is structured.

Listing 3.27 presents a different loop mutation, consisting of the replacement of the loop’s condition with `1 < 1`, as done by Go-Mutesting¹⁴. The provided example works in Dafny simply because the expression equates to `false`, making the loop not execute. We consider that this specific replacement would be equivalent to the application of either a **BBR** or an **EVR** operator to the original guard of the loop. Any other expression would most likely cause the verifier to be unable to prove loop termination.

Listing 3.27: Example of a loop condition replacement mutation applied to a Dafny program

```

1 method Find(a: array<int>, key: int) returns (index: int)
2 {
3     index := 0;
4
5 - while index < a.Length
6 + while 1 < 1
7     invariant 0 <= index <= a.Length
8     {
9         if (a[index] == key) {
10             break;
11         }
12         index := index + 1;
13     }
14 }

```

¹⁴<https://github.com/avito-tech/go-mutesting>, accessed 24 Jul 2025.

3.3.8.2 Case Block Replacement (CBR)

PIT⁷ implements an expression replacement operator that specifically targets the body of `case` blocks in switch statements. It replaces the body of the switch's default label with the body of the first non-default label and the remaining labels' bodies with the default one. In Dafny a similar syntax construct exists, the match statement, in which the default case is marked with the wild variable `_`. Therefore, the same transformation can be applied, as shown in Listing 3.28.

Listing 3.28: Example of a match case replacement mutation applied to a Dafny program

```

1 method GetValue(n: nat) returns (r: nat)
2 {
3     match n
4         case 0
5 -         => r := 1;
6 +         => r := n + 1;
7         case 1
8 -         => r := 10;
9 +         => r := n + 1;
10        case _
11 -         => r := n + 1;
12 +         => r := 1;
13 }

```

3.3.9 Statement Deletion (SDL)

Similar to expression replacement, statement deletion operators can also be categorized according to the context in which they are applied. The most general operator, however, consists of the deletion of a statement of any type (`return`, `break`, assignment, method call, etc.) instead of focusing on the statements belonging to a specific category. Listing 3.29 shows an example of the application of this operator to an assignment statement. Regarding this type of statement, we take the precaution of only applying this operator when the variable being assigned to has already been initialized. For instance, its application to Line 3 would generate an invalid mutant due to the `average` variable (accessed in Line 4) being uninitialized.

Listing 3.29: Example of an assignment deletion mutation applied to a Dafny program

```

1 method AverageTwoNumbers(a: int, b: int) returns (average: int)
2 {
3     average := a + b;
4 -     average := average / 2;
5 }

```

In the following sub-sections, we present other application cases of the **SDL** operator.

3.3.9.1 Block deletion

An extreme case of statement deletion can be the deletion of an entire code block. In Dafny, it is possible to delete the entire body of a method, as done in Listing 3.30, entire `if`, `else if`, and `else` branches, as done in Listing 3.31, or `case` blocks used in the context of match statements, like the example in Listing 3.32. However, deleting a method's body only generates valid programs for void methods and `if` branches can only be deleted when they do not have alternatives associated.

Listing 3.30: Example of a method block deletion mutation applied to a Dafny program

```

1 method DoubleArrayElements(a: array<int>)
2 modifies a
3 {
4 -   for i := 0 to a.Length
5 -   {
6 -       a[i] := a[i] * 2;
7 -   }
8 }
9
10 method Main()
11 {
12     var a := new int[3][1, 2, 3];
13     DoubleArrayElements(a);
14 }

```

Listing 3.31: Example of an else branch deletion mutation applied to a Dafny program

```

1 method Abs(x: int) returns (abs: int)
2 {
3     abs := 0;
4     if x > 0 {
5         abs := x;
6 -   } else if x < 0 {
7 -       abs := -x;
8 -   }
9 }

```

Listing 3.32: Example of a case block deletion mutation applied to a Dafny program

```

1 method GetValue(n: nat) returns (r: nat)
2 {
3     match n
4         case 0 => r := 1;
5 -     case 1 => r := 10;
6         case _ => r := n + 1;
7 }

```

3.3.9.2 Method call deletion

The deletion of void method calls can be done without generating invalid programs, as depicted in Listing 3.33. For non-void method calls, this is not implemented since they usually stand as a placeholder

Listing 3.33: Example of a method call deletion mutation applied to a Dafny program

```

1 method DoubleArrayElements(a: array<int>)
2   modifies a
3   {
4     for i := 0 to a.Length
5     {
6       a[i] := a[i] * 2;
7     }
8   }
9
10 method Main()
11 {
12   var a := new int[3][1, 2, 3];
13 -   DoubleArrayElements(a);
14 }

```

for a value that is used in the program—their return value may be assigned to a variable, used as an operand to a binary expression, etc.—and deletion of the call altogether would result in a syntactically incorrect program. As presented in Section 3.3.7.2, it makes more sense to handle these cases with a **MRR** operator.

When dealing with a method that is a member of a class, the Method Naked Receiver (**MNR**) is a more specific type of deletion mutation that can be implemented. This operator works by deleting the method call and maintaining the receiver object, as done in Listing 3.34. The application of this operator requires the receiver to have the same type as the method’s return value, avoiding program invalidity due to type mismatches. As a result, it can only be applied to non-void methods.

Listing 3.34: Example of a naked receiver mutation applied to a Dafny program

```

1 class Item {
2   var value: int
3
4   constructor(value: int)
5   {
6     this.value := value;
7   }
8
9   method CreateUpdatedObject(n: int) returns (i: Item)
10  {
11    i := new Item(this.value + n);
12  }
13 }
14
15 method CreateAndUpdateItem(value: int)
16 {
17   var newItem := new Item(value);
18 -   var updatedItem := newItem.CreateUpdatedObject(3);
19 +   var updatedItem := newItem;
20 }

```

3.3.9.3 Field initialization

Finally, there are two operators that cause the fields of a class to be initialized with arbitrary values. One of them can be applied to fields that are initialized at the declaration level. Another acts in the context of fields that are initialized in the class’s constructor. Dafny does not provide specific default values for each type but ensures that variables of certain types, called auto-initializable, always hold legal values. In this light, when initialization values are not specified, Dafny automatically assigns arbitrary values for each of its auto-initializable types.

Listing 3.35’s example consists of the deletion of the initialization values provided upon field declaration and involves several types. By printing each of the variables, we can see the value of `0.0` for the real, `false` for the boolean, and that the set is empty. However, these are just an indication of the value’s arbitrary status and would not be accepted as correct by the verifier, in an assert statement, for instance. Despite this, the operator is still useful in detecting potential spec weaknesses.

This phenomenon cannot be observed for instances of user-defined classes, since they are not auto-initializable. Additionally, only constants are allowed by Dafny to be initialized at the declaration level.

Listing 3.35: Example of a field initialization deletion mutation applied to a Dafny program

```

1 class Person {
2 -   const height: real := 160.0 // 0.0
3 -   const isAdult: bool := true // false
4 -   const favouriteNumbers: set<int> := {1, 2, 3} // {}
5 }
```

Listing 3.36 reflects the same effect but, this time, caused by the deletion of the body of the class’s constructor. This operator is implemented by MuJava [41, 43] in a slightly different way: the entire constructor is deleted, instead of just its body, and Java automatically generates a default constructor with each field initialized with the language’s default. Dafny also provides the automatic constructor creation feature, but these have the peculiarity of being called differently than the ones implemented by the programmer: they are called without the parentheses, e.g., `new Person`, as opposed to `new Person()`. If we were to implement this operator the same way MuJava does, upon constructor deletion, any instantiation would have to be done without parentheses, and constructor calls in other parts of the program, previously done with parentheses, would no longer be accepted, resulting in an invalid program. By simply deleting the body of the constructor, we preserve the desired effect of the mutation, since the fields will still be initialized with arbitrary values, while avoiding the described side effects.

3.3.10 Variable Deletion (VDL)

The variable deletion operator works by deleting all occurrences of a variable. When it occurs as one of the operands in a binary expression, the binary operator must also be deleted, as exemplified in Listing 3.37. Some tools also implement a constant deletion operator, which behaves much the same. It could be applied to Listing 3.37 if, instead of targeting variable `sum`, we targeted constant `n`.

Listing 3.36: Example of a constructor deletion mutation applied to a Dafny program

```

1 class Person {
2   var height: real
3   var isAdult: bool
4   var favouriteNumbers: set<int>
5   constructor() {
6 -   this.height := 160.0;
7 -   this.isAdult := true;
8 -   this.favouriteNumbers := {1, 2, 3};
9   }
10 }

```

Listing 3.37: Example of a variable deletion mutation applied to a Dafny program

```

1 const n := 2
2 method AverageTwoNumbers(a: int, b: int) returns (average: int) {
3 - var sum := a + b;
4 - average := sum / n;
5 }

```

Python supports the following syntax for slicing sequence types, such as lists and strings: `s[start:end:step]`, and MutPy [13] implements a mutation operator that deletes one of the slice definition's elements, i.e., either `start`, `end`, or `step`. In Dafny, there is a similar functionality, which only works for sequences, and does not support `step` but supports `start` and `end`. Therefore, it is possible to implement MutPy's operator for Dafny's sequences, as done in Listing 3.38. This operator, though also dealing with the deletion of a variable, is slightly different than the previous since it does not delete all of its occurrences. In order to differentiate, we will call it Subsequence Limit Deletion (SLD).

Listing 3.38: Example of a subsequence boundary deletion mutation applied to a Dafny program

```

1 method Subsequence() returns (s: seq<int>) {
2   var start := 1;
3   s := [1, 2, 3, 4, 5];
4 - s := s[start..]; // s := [2, 3, 4, 5]
5 + s := s[..]; // s := [1, 2, 3, 4, 5]
6 }

```

3.3.11 Operator Deletion (ODL)

Similar to the variable deletion operator, this one deletes all occurrences of a specific operator, which in Dafny can be arithmetic, relational, conditional, logical, and shift. For binary operators, it is necessary to delete one of the operands as well. The application of this operator is illustrated in Listing 3.39.

Listing 3.39: Example of an operator deletion mutation applied to a Dafny program

```

1 method AverageThreeNumbers(a: int, b: int, c: int) returns (average: int) {
2 - var sum := a + b;
3 - sum := sum + c;
4   average := sum / 3;
5 }

```

3.3.12 Object-Oriented operators

MuJava can be highlighted as the first tool to propose a comprehensive set of mutation operators specifically directed at evaluating the use of OO features such as inheritance, polymorphism, and encapsulation. Some of these operators can be repurposed for Dafny, a programming language with support for OO features. Listing 3.40 presents an example of an OO Dafny program which depicts two classes, both inheriting from the same parent trait. This single case will be used to exemplify each mutation described in this subsection.

Listing 3.40: Base example of a Dafny OO program

```

1 trait Shape {
2   var numSides: int
3 }
4
5 class Rectangle extends Shape {
6   var length: real, width: real
7   constructor(length: real, width: real)
8   {
9     this.numSides := 4;
10    this.length := length;
11    this.width := width;
12  }
13
14  method GetLength() returns (length: real)
15  {
16    length := this.length;
17  }
18
19  method GetWidth() returns (width: real)
20  {
21    width := this.width;
22  }
23
24  method SetLength(length: real)
25  modifies this
26  {
27    this.length := length;
28  }
29
30  method SetWidth(width: real)
31  modifies this
32  {
33    this.width := width;
34  }
35
36  method CheckProportion(width: real, length: real) returns (isProportional: bool)
37  requires width > 0.0 && length > 0.0
38  {
39    var ratio := this.width / width;
40    isProportional := length * ratio == this.length;
41  }
42 }

```

```

43
44 class Triangle extends Shape {
45     constructor()
46     {
47         this.numSides := 3;
48     }
49 }

```

3.3.12.1 Polymorphic Reference Replacement (PRV)

The **PRV** operator replaces the value of an assignment to an object of a parent class with an object of a different child class than the one being used in the original program. This operator can only be used when the parent class has more than one child and there are at least two variables of different child types in the method that is the mutation target. In our running example, `Rectangle` and `Triangle` are both children of the trait `Shape` and in Listing 3.41 one instance of each is declared: variable `rectangle` and `triangle`, respectively. Originally, the `rectangle` variable is assigned to another variable of type `Shape`, however, the mutation causes `triangle` to be assigned instead.

Even though this operator always generates valid programs, it may have side effects when different child classes have different implementations for a common method that is called in the mutation target method or their constructors initialize certain fields, which are used in said method, differently. In the example, each subclass has a different value for the `numSides` field, therefore the mutation will cause the wrong value to be printed.

Listing 3.41: Example of a subclass reference assignment replacement mutation applied to a Dafny program

```

1 method Main()
2 {
3     var shape: Shape;
4     var rectangle := new Rectangle(10.0, 20.0);
5     var triangle := new Triangle();
6 -   shape := rectangle; // numSides = 4
7 +   shape := triangle; // numSides = 3
8     print shape.numSides, "\n";
9 }

```

3.3.12.2 This Keyword Insertion (THI) and This Keyword Deletion (THD)

MuJava presents two operators that involve manipulating the `this` keyword, one consisting of its insertion and another of its deletion. These mutations only make sense in the context of a class method with a parameter that has the same name as one of its class fields. In the first one, depicted in Listing 3.42, `this` is added in front of a usage of the parameter in question, and the variable starts referring to the field instead. The second one, depicted in Listing 3.43, is the opposite: upon removing the keyword, the variable starts referring to the parameter.

Neither of the operators can ever be applied to the Left-Hand Side (**LHS**) of an assignment operation because Dafny does not allow assignments to parameters. For the insertion operator, inserting a **this** keyword would imply the **LHS** of the assignment to be a parameter in the original code, which is not possible. On the other hand, its deletion would cause the mutant to have a parameter in the **LHS** of the assignment, making it an invalid program.

Listing 3.42: Example of a **this** keyword insertion mutation applied to a Dafny program

```

1 method SetLength(length: real)
2 modifies this
3 {
4 - this.length := length;
5 + this.length := this.length;
6 }

```

Listing 3.43: Example of a **this** keyword deletion mutation applied to a Dafny program

```

1 method CheckProportion(width: real, length: real) returns (isProportional: bool)
2 requires width > 0.0 && length > 0.0
3 {
4   var ratio := this.width / width;
5 - isProportional := length * ratio == this.length;
6 + isProportional := length * ratio == length;
7 }

```

3.3.12.3 Accessor Method Replacement (**AMR**) and Modifier Method Replacement (**MMR**)

The last two operators perform the replacement of one method with another with the same signature. One of them acts specifically over accessor methods, those typically starting with **get** and that are used to access fields of a class, and the other acts over modifier methods, those which conventionally start with **set** and are responsible for updating an object's state. These operators are illustrated in Listing 3.44 and Listing 3.45, respectively.

Listing 3.44: Example of an accessor method replacement mutation applied to a Dafny program

```

1 method Main()
2 {
3   var rectangle := new Rectangle(10.0, 20.0);
4 - var length := rectangle.GetLength();
5 + var length := rectangle.GetWidth();
6 }

```

Listing 3.45: Example of a modifier method replacement mutation applied to a Dafny program

```

1 method Main()
2 {
3   var rectangle := new Rectangle(0.0, 0.0);
4 - rectangle.SetLength(10.0);
5 + rectangle.SetWidth(10.0);
6 }

```

3.4 Mutation operators proposed for other languages but excluded for Dafny

The majority of the operators identified for other programming languages that cannot be applied to Dafny consist of mutations to specific syntax constructs that do not exist in Dafny. Some constructs that are available in many languages but not in Dafny are shortcut operators, as mentioned earlier, compound assignment operators, such as `+=`, switch statements, exception-catching blocks, and method overloading. Many operators also involve Python's `None` keyword, which does not exist in Dafny. Furthermore, some operators deal with language-specific classes or libraries, such as Java's `BigInteger`, C#'s `Math`, and C#'s `Language Integrated Query (LINQ)`.

In some cases, operators may deal with syntax constructs that do in fact exist in Dafny, but the behaviour of the mutation itself may be impossible or generate equivalent mutants due to restrictions inherent to Dafny. Below we highlight some other operators that cannot be applied to Dafny due to more complex reasons.

3.4.1 Operator replacement operators

Major's *Operator Replacement Unary (ORU)* cannot be applied to Dafny because it consists of the replacement of unary operators that do not necessarily belong to the same group, for instance between Java's `-` and `~`, whereas in Dafny, the only two unary operators (`-` and `!`) have different type requirements. To be more specific, `-` can only be used in front of a numerical type expression, while `!` can only be used before a boolean or a bit-vector type expression. This makes it impossible to replace one operator with the other and still get a valid program.

3.4.2 Expression replacement operators

Stryker.NET's *Initialization* operator can be partially applied to Dafny when it comes to the empty initialization of collection types, as previously illustrated in Listing 3.22 and Listing 3.23. Even so, the operator is originally also applied to the initialization of objects through the replacement of a parameterized constructor with an empty one. This specific use case cannot be applied to Dafny programs because Dafny only allows its classes to implement one constructor.

The *Method Expression* operator implemented by StrykerJS is a very general method replacement operator which acts over several methods specific to the JavaScript language and belonging to several different domains: strings, arrays, and math. For instance, `toUpperCase()` is replaced with `toLowerCase()`, `startsWith()` is replaced with `endsWith()`, `some()` is replaced with `every()`, `min()` is replaced with `max()`, etc. Dafny provides limited built-in support for manipulating its types. Not only the aforementioned methods do not exist in Dafny, but there are also practically no other built-in methods for which the behaviour described by the operator could be repurposed for.

Stryker.NET implements an operator similar to the last one but specifically directed to string manipulation, the *String Methods* operator. The previous explanation also partially applies to why this operator cannot be implemented in Dafny, another constraint being that strings in Dafny are not objects and, therefore, would have no class methods associated.

3.4.3 Statement deletion operators

Mutmut's *Argument List Mutation* operator performs the deletion of one of the elements from a list of arguments. This mutation makes sense in Python because it is possible to define optional arguments with default values. However, Dafny does not provide this feature and any method call must be done with the correct fixed number of arguments.

3.4.4 Object-Oriented operators

When explaining the restrictions in using Dafny's OO features it is important to note that Dafny's classes are only allowed to extend traits, which can be viewed as abstract superclasses and cannot declare constructors or be instantiated. Methods and fields can both be declared in the trait and used in its subclasses and methods can be implemented either in the trait or in the subclasses.

3.4.4.1 Inheritance

MuJava's inheritance-related operators test three different aspects of this feature in Java: overriding methods, hiding variables, and the use of the `super` keyword. Dafny supports none of these features.

When a method is declared in a trait but no implementation is provided, all subclasses are required to provide their own implementation for that method. However, when the trait implements a method, said implementation is common to all subclasses and alternative ones at the subclass level are not allowed, which means it is not possible to override methods. Regarding hiding variables, when a field is declared in a trait Dafny does not allow a field with the same name to be re-declared in any of its subclasses.

Finally, the `super` keyword does not exist in Dafny. Moreover, since parent classes cannot declare constructors, there is no equivalent mechanism for invoking a superclass's constructor.

3.4.4.2 Polymorphism

MuJava's polymorphism-related operators perform replacement of child types with parent types and vice versa in several different contexts such as variable declaration, parameter declaration, and type casting. While some of these replacements are technically possible, we have concluded that valid mutants are always equivalent to the original program.

Because superclasses cannot be instantiated, a constructor of a subclass always needs to be invoked in order to create an object. Consequently, Dafny always knows which specific type of object is being used, even when it is declared with the type of the superclass. We can observe this in the example of

Listing 3.46. Even though we change the declaration of the type to a more general one, Dafny still treats it as a `Rectangle` using the correct values for its fields and the correct implementation for methods that are declared but not implemented in the parent. The behaviour of the program does not change with the mutation, so this mutant is said to be equivalent and, therefore, incapable of providing insight into the correctness of the original program.

Listing 3.46: Example of child/parent declaration type replacement mutation applied to a Dafny program

```

1 method Main()
2 {
3 - var rectangle: Rectangle;
4 + var rectangle: Shape;
5   rectangle := new Rectangle(10.0, 20.0);
6   print rectangle.numSides, "\n"; // numSides = 4
7 }

```

When applying mutations that replace child with parent types, if the original program uses methods or fields that are defined exclusively in the child then the program becomes invalid. Otherwise, independently of the context in which the replacement occurs and for the reasons outlined above, these types of mutation operators will always generate equivalent mutants.

In type casting contexts, the inverse replacement, i.e., from parent to child, is also performed. This can only be done in Dafny with prior type checking, for instance inside of an `if var is Type` block in which `Type` is the type `var` will be cast into. These type checking statements are unlikely to be present in most programs that do not already perform downcasting, thus these types of operators will almost always generate invalid programs.

To summarize, mutation operators that replace child with parent types, when applied to Dafny, will either generate invalid or equivalent programs and, therefore, have no use being implemented in a mutation testing tool for Dafny programs. We can, however, note that, as presented in Section 3.3.12.1, child-to-child type replacement constitutes a polymorphic mutation that does indeed make sense in Dafny.

3.4.4.3 Static keyword

MuJava proposes one operator to perform insertion of the `static` keyword in front of a class field and another to perform its deletion. In Java, this operator is useful because the behaviour of a class 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, on the other hand, only constant class fields, whose value cannot be updated, can be made static. Having a field be static or not causes internal differences in the program, for example a static field is stored in a single memory location shared between all class instances, yet its behaviour is not altered regardless of how the field is declared. For this reason, these two mutation operators also fall into the category of operators that always generate equivalent mutants in Dafny and should not be implemented.

3.5 Answer to RQ1, Given the set of mutation operators that have been proposed in the literature, which ones could be applied to Dafny programs?

RQ1: From the total of 147 compiled mutation operators, we can apply 90 to Dafny, 30 of them being unique. This includes a wide range of operators involving the replacement of binary operators, the insertion and deletion of unary ones, the replacement of literal values or complete expressions with other literal values, the replacement of language keywords, and the deletion of constructs.

3.6 Summary

The mapping of mutation operators from non-verification-aware languages to Dafny provides a strong foundation for the development of a mutation tool for the language, as will be presented in Chapter 5.

Chapter 4

New Mutation Operators for Dafny

Mutation operators typically try to mimic faults commonly made by developers when writing a program, which may differ depending on the features offered by each programming language. Chapter 3 equipped us with a substantial list of mutation operators that can be implemented in Dafny. While these may be useful as a starting point, revealing faults that may be transversal to common features between several languages, they were all proposed taking the requirements of other languages, in particular non-verification-aware ones, into consideration.

In this chapter, we intend to explore this problem further by studying evidence of real faults in Dafny programs and synthesizing them into a new catalog of mutation operators tailored for the language. This effort will enable us to answer our **RQ2**: *What mutation operators could be synthesized from bugfixes or previous changes to Dafny programs?*

4.1 Methodology

Verification-aware languages are still far less adopted than traditional ones, and as a result, there are not many studies on which mistakes commonly occur in Dafny. The easiest way to study this is by analyzing evidence of bugfixes present in projects publicly available on GitHub.

DafnyBench¹, proposed by Loughridge et al. [40], is currently the largest benchmark for Dafny formal verification with 785 programs and about 53,000 lines of code. 62 of the 785 programs in the benchmark were collected from the Clover dataset [65], 164 from the dafny-synthesis benchmark [49], and the remaining 559 programs were scraped from a total of 120 GitHub repositories found to include Dafny files. However, note that (a) four of the repositories had been renamed at the time their paper was published, (b) two have been renamed or deleted since, (c) three appeared repeated on the list, and one was a subset of another, resulting in the exclusion of the 10 repositories that are summarized in Table 4.1. Additionally, we decided to include the repositories for the AWS encryption SDK² and

¹<https://github.com/sun-wendy/DafnyBench>, accessed 24 Jul 2025.

²<https://github.com/aws/aws-encryption-sdk>, accessed 24 Jul 2025.

Table 4.1: DafnyBench repositories not included in study.

Repository	Reason for exclusion
BinarySearchTree (no link provided)	(a) unavailable upon publishing
CO3408-Advanced-Software-Modelling-Assignment-2022-23-Part-2-A-Specification-Spectacular (no link provided)	(a) unavailable upon publishing
Programmverifikation-und-synthese (no link provided)	(a) unavailable upon publishing
Prog-Fun-Solutions (no link provided)	(a) unavailable upon publishing
https://github.com/AoxueDing/Dafny-Projects , accessed 24 Jul 2025.	(b) unavailable since publishing
https://github.com/Aaryan-Patel-2001/703FinalProject , accessed 24 Jul 2025.	(b) unavailable since publishing
https://github.com/Eggyl15/Dafny , accessed 24 Jul 2025.	(c) repeated
https://github.com/vladstejeroiu/Dafny-programs , accessed 24 Jul 2025.	(c) repeated
https://github.com/isobelm/formal-verification , accessed 24 Jul 2025.	(c) repeated
https://github.com/secure-foundations/iron-sync , accessed 24 Jul 2025. $\subset$ https://github.com/secure-foundations/ironsync-osdi2023 , accessed 24 Jul 2025.	(d) subset

Dafny-EVM projects³, which are widely recognized as key examples of the application of Dafny in the industry. In practice, this left us with a total of 112 repositories to explore.

Evidence of programming errors in repositories may manifest in bug reports (i.e., issues labeled *bug* on GitHub) or commit messages referring to fixes or corrections. 108 out of the 112 repositories contained no bug reports, so our analysis had to be commit-based, with the dataset covering 12,308 commits in total. In order to avoid manually reading the message of all commits, and aiming to achieve replicability in identifying which commit messages hint at bugfixes, we ran the `git log` command together with `grep` and a group of text patterns in each repository. The chosen patterns were the following:

- “fix”: 925 occurrences.
- “correct”, including both the main keyword and other variations such as “correction”, “corrected”, etc.: 146 occurrences.
- “updat”, including variations like “update”, “updating”, etc.: 332 occurrences.
- “chang”, including, for example, the keywords “change” and “changing”: 214 occurrences.

The first two directly refer to fault amending, while the latter may or may not contain corrections, but, due to being commonly used in commit messages, we considered worth collecting for further analysis.

³<https://github.com/Consensys/evm-dafny>, accessed 24 Jul 2025.

In total, we collected 1,475 commits with messages hinting at possible bugfixes — some containing more than one of the patterns listed above — that could potentially be synthesized into mutation operators for Dafny. On average, each commit contained changes to one Dafny file, this number, nonetheless, varying up to 254 files, and around 16 line insertions and 7 line deletions.

Understanding whether a change performed in a commit would make sense as a mutation operator is not straightforward. For this reason, our approach was to make a decision based on discussion between the three authors of this research. Each of the authors was responsible for analyzing one-third (approximately 491) of the commits and registering any changes that appeared to be a potential mutation operator. Afterwards, the team discussed each flagged commit until reaching an agreement on the suitability of any of its changes as mutation operators for Dafny.

Alike the initial individual analysis and the discussion process took several details into consideration. Since our goal is to mutate implementations, changes involving syntax constructs particular to specifications were disregarded. Additionally, some changes required taking full program context into consideration, in order to understand whether they corresponded to a correction in program behaviour — our target case — to a fix to a previously invalid program, or simply to a refactoring. Finally, mutation operators often consist of simple and localized changes, hence their synthesis should exclude ones that involve several logical steps and encompass sizeable portions of code.

4.2 Synthesized operators

The commit review process described in Section 4.1 uncovered 11 new operators suitable to be included in a mutation testing tool for Dafny programs. In this section, we briefly describe each of the operators and in Table C.1 we present some example commits from which we synthesized them.

4.2.1 Extended Conditional Operator Replacement (COR)

The conditional operator replacement was previously introduced in Section 3.3.3.1 as an operator that performs the replacement between the conditional binary operators available in a language. Dafny supports the standard conditional operators `&&` and `||`, common to most traditional programming languages. However, during the study described in this chapter, we found evidence of the replacement of these two operators with other Dafny-specific ones. In reality, Dafny offers a set of additional conditional operators that do not usually exist in traditional languages, `==>` (implication), `<==` (reverse implication), and `<==>` (equivalence), making it possible to extend the previously defined conditional replacement to include them. Listing 4.1 presents an example of the extended application of this mutation operator.

Listing 4.1: Example of a conditional replacement mutation extended to include Dafny-specific operators

```

1 method IsBetween(a: int, b: int, c: int) returns (res: bool) {
2   - return c < a || c > b;
3   + return c < a ==> c > b;
4 }
```

4.2.2 Expression replacement

Section 3.3.8 presented the replacement of expressions with other expressions in specific contexts, such as loops. Beyond that, evidence shows that it is common for, in particular, identifiers to be mistakenly swapped. Below, we describe different operators involving this type of mutation.

4.2.2.1 Variable Expression Replacement (VER) and Field Access Replacement (FAR)

The variable expression replacement operator specifically targets references to variable names, as shown in Listing 4.2. The original variable's type should be considered when choosing a replacement from the variables available in the scope, restricting the operation to ones of the same type. For instance, in the example, replacing the same target variable reference with `message` would lead to an invalid program.

Listing 4.2: Example of a variable expression replacement mutation applied to a Dafny program

```

1 method IsBetween(a: int, b: int, c: int, message: string) returns (res: bool)
2 {
3 - if (c < a || c > b) {
4 + if (b < a || c > b) {
5     res := true;
6     print message;
7 } else {
8     res := false;
9 }
10 }
```

The field access replacement operator also consists of identifier replacement, yet in this case, the targeted expressions constitute accesses to class fields. The operator preserves the class reference portion of the expression, exclusively replacing the field with another one declared within the same class, once again subject to type restrictions. Listing 4.3 depicts an example of the application of this operator.

Listing 4.3: Example of a field access replacement mutation applied to a Dafny program

```

1 class Item {
2     var item: string
3     var price: int
4     var stock: int
5 }
6
7 method GetProfit(item: Item) returns (profit: int)
8 {
9 - profit := item.price * item.stock;
10 + profit := item.price * item.price;
11 }
```

4.2.2.2 Method Call Replacement (MCR) and Datatype Constructor Replacement (DCR)

These two operators work in a similar manner to the preceding two. As their names imply, the method call replacement operator replaces one method (or function) call with another, as illustrated in List-

ing 4.4, and the datatype constructor replacement operator replaces one datatype constructor call with another available for the same datatype, as done in Listing 4.5. Both operators retain the original arguments. With regard to ensuring program validity, they require limiting the replacement to callable elements with the same signature as the original: not only return value, in the case of the MCR operator, but also number, type, and sequence of the arguments.

Listing 4.4: Example of a method call replacement mutation applied to a Dafny program

```

1 method Sum(a: int, b: int) returns (c: int)
2 {
3   c := a + b;
4 }
5
6 method Multiply(a: int, b: int) returns (c: int)
7 {
8   c := a * b;
9 }
10
11 method Main()
12 {
13 - var n := Sum(10, 20);
14 + var n := Multiply(10, 20);
15 }

```

Listing 4.5: Example of a datatype constructor replacement mutation applied to a Dafny program

```

1 datatype MyQuantifier = None | Some(x: set<int>) | All(y: set<int>)
2
3 method Main()
4 {
5 - var selection := Some({1, 2, 3, 4});
6 + var selection := All({1, 2, 3, 4});
7 }

```

4.2.2.3 Method-Variable Replacement (MVR)

Besides replacing variables with other variables and methods with other methods, we also found evidence of the replacement of methods with variables being a common fix. Listing 4.6 exemplifies this type of mutation. In this case, the existing type restriction requires the replacement variable to be of the same type as the target method's return value.

4.2.3 Argument Swapping (SAR)

Given a callable element, i.e., a method, a function, a constructor, etc., another mistake that is commonly made is the swapping of its arguments. The mutation operator that replicates this type of fault swaps two arguments of the same call, as depicted in Listing 4.7. Because the arguments are only swapped in a specific call, its signature being preserved, the mutated program is only valid if the swapped arguments have the same type.

Listing 4.6: Example of a replacement mutation between a method and a variable applied to a Dafny program

```

1 method SumArrayElems(elems: array<int>) returns (sum: int)
2 {
3     sum := 0;
4     var i := 0;
5     while (i < elems.Length)
6     {
7         sum := sum + elems[i];
8         i := i + 1;
9     }
10 }
11
12 method Main()
13 {
14     var n := 4;
15     var a := new int[n][1, 2, 3, 4];
16 - var sum := SumArrayElems(a);
17 + var sum := n;
18     var avg := sum / n;
19 }

```

Listing 4.7: Example of an argument swapping mutation applied to a Dafny program

```

1 method Sum(a: int, b: int) returns (c: int)
2 {
3     c := a + b;
4 }
5
6 method Main()
7 {
8 - var n := Sum(10, 20);
9 + var n := Sum(20, 10);
10 }

```

4.2.4 Tuple Access Replacement (TAR)

In Dafny, tuples can have an arbitrary number of elements, which are accessed using a dot followed by their positional index, as if accessing a class field. This mutation operator performs the replacement of an index used in a tuple element access context by a different index corresponding to another element of the same type as the original, as done in Listing 4.8. Element access mutations can also occur in other collection types, such as arrays, but in these, indexes are represented as literal expressions inside of specific selection expressions and are, therefore, implemented as a part of the standard LVR operator.

Listing 4.8: Example of a tuple access replacement mutation applied to a Dafny program

```

1 method GetLatitude(coords: (real, real, real)) returns (lat: real)
2 {
3 - lat := coords.0;
4 + lat := coords.1;
5 }

```

4.2.5 Conditional Block Extraction (CBE)

One of the operators presented in Section 3.3.9.1 consisted of deleting one of the branches in an if statement. Our study on commonly committed faults in Dafny programs showed that an equally prevalent mistake is the extraction of the content of one of the if statement's branches to the outside scope, while the remaining branches are deleted. There are two main differences between the two operators: the first only deletes one branch, preserving the if statement, while the second only maintains one branch and converts it into a regular code block. Listing 4.9 exemplifies the simplest case of application of this operator and Listing 4.10 showcases its application in the context of a ternary statement.

Listing 4.9: Example of a conditional block extraction mutation applied to an if Dafny statement

```

1 method Find(a: array<int>, key: int) returns (index: int)
2 {
3     index := -1;
4
5     for i := 0 to a.Length
6     invariant 0 <= i <= a.Length
7     {
8 -     if a[i] == key {
9         index := i;
10        break;
11 -     }
12    }
13 }
```

Listing 4.10: Example of a conditional block extraction mutation applied to an if-then-else Dafny statement

```

1 function CreateSeq(a: int) : seq<int>
2 {
3 - if a <= 0 then
4 -     []
5 - else
6     var b, c := a + a, a * a;
7     [a, b, c]
8 }
```

4.2.6 Statement Swapping (SWS)

Another frequently encountered program correction involves swapping two or more lines of code, corresponding to corrections in logical flow affecting program behaviour. While this concept is somewhat broad to be fully translated into a mutation operator, considering the exponential number of mutants that would result from swapping every line and group of lines in a program, it is possible to create simpler operations that replicate part of this problem. For instance, we propose swapping a mutation target statement with either the one immediately above, as done in Listing 4.11, or below it.

Listing 4.11: Example of a statement swapping mutation applied to a Dafny program

```

1 method Factorial(n: int) returns (r: int)
2 {
3   r := 1;
4   var i := 0;
5
6   while i < n
7   {
8 -     i := i + 1;
9     r := r * i;
10 +    i := i + 1;
11   }
12 }

```

4.2.7 Variable Declaration Swapping (SWV)

Lastly, the **SWV** operator reflects another case related to the swapping of program constructs, but this time, applied to the declaration of variables in the same scope, as depicted in Listing 4.12. This operator has a similar effect to the **VER**, since the behaviour of a mutated variable will correspond to that of a different one. However, while **VER** only affects the single place where the variable is replaced, this operator disturbs its behaviour across the entire scope.

Listing 4.12: Example of a variable declaration swapping mutation applied to a Dafny program

```

1 method GetCircleDimensions(radius: real)
2 {
3 - var perimeter := 2.0 * radius * 3.14;
4 - var area := radius * radius * 3.14;
5 + var perimeter := radius * radius * 3.14;
6 + var area := 2.0 * radius * 3.14;
7
8   print "Perimeter: ", perimeter, "\n";
9   print "Area: ", area, "\n";
10 }

```

4.3 Common operations that do not correspond to mutation operators

Despite the analysis of bugfixes in Dafny projects only having revealed 11 new mutation operators, many of the analyzed commits contained changes that, at first glance or for someone unfamiliar with Dafny's syntax, also came across as having potential. We can highlight three groups of common operations: (a) changes that alone do not alter program behaviour, (b) changes inherent to migrations from Dafny 3 to Dafny 4, and (c) code improvements. For each case we present below, Table C.2 lists some examples of commits showcasing that fix.

In the first group, we include operations such as the replacement of a type between `nat`, `int`, and `real`, `nat` being a subset of `int`, and `int` literals being represented differently than `real` literals (`.0` is mandatory). An upcast from `nat` to `int` is always valid, while the opposite is only valid if no negative values are assigned to the original `int` variable. Nevertheless, neither case changes program behaviour

since the values held by the variables remain the same. On the contrary, the replacement between `nat` or `int` and `real` would simply cause an initialization conflict. Another change consists of the replacement between the `var` and `const` keywords, which merely restrict whether a variable can or not be updated, similarly having no effect on program behaviour. Lastly, the addition and deletion of type parameter properties also appear recurrently. This Dafny-specific feature is used to identify different equality, auto-initializability, variance, and cardinality, among other, properties that a type must meet to be passed as a parameter. For example, type `T(==)` supports equality, type `A<+T>` is co-variant, a subtype of `T` is also a subtype of `A<+T>`, and cardinality preserving, the cardinality of `A<+T>` does not exceed the one of `T`, in turn, type `A(0)<!T>` is auto-initializable, non-variant and not necessarily cardinality preserving. Because these pose restrictions on the types passed as parameters, they determine program validity, once again, without changing its behaviour.

Version 4 of Dafny brought some syntactical changes, particularly how functions and other callable elements are declared. Previous to version 4, functions were, by default, `ghost` — only available to be used in specifications and, thus, not compiled — while compilable functions had to explicitly be declared as `function` `method`. In version 4, current version, all functions are compiled, unless declared as `ghost function`. A large number of commits changed functions to ghost functions, function methods to functions, and similar to lemmas and predicates. None of these replacements corresponds to programming errors.

Finally, common code improvement changes include the replacement of the declaration of a callable element as a method, function, predicate, or lemma, and the insertion of `ghost` keywords, which optimizes program compilation by eliminating the need to compile certain elements. A particular case worth highlighting is the replacement between `s.Length` and `|s|`, a very prevalent change that indeed appears as a correction but, upon further investigation, we can conclude only occurs due to a refactoring consisting of turning an array into a set.

4.4 Answer to RQ2, What mutation operators could be synthesized from bugfixes or previous changes to Dafny programs?

RQ2: We found 11 operators suitable to the mutation of Dafny programs: two involving Dafny-specific features (extended **COR** and **DCR**) and nine (**CBE**, **FAR**, **MCR**, **MVR**, **SAR**, **SWS**, **SWV**, **TAR**, **VER**) involving traditional programming constructs, such as statements, variables, method calls, class field accesses, and control flow. The latter also have the potential to be applied in traditional programming languages. In addition to these 11 newly collected operators, we also observed fixes corresponding to mutation operators identified in Chapter 3, validating and emphasizing their inclusion in a mutation testing tool for Dafny.

4.5 Summary

We have detailed the process of synthesizing new mutation operators for the Dafny language, building upon the systematic analysis of 1,475 commits selected from real Dafny projects. The new identified mutation operators allow us to complement traditional mutation testing techniques according to the language's needs. Chapter 5 details the implementation of this combined approach in a mutation testing tool for Dafny.

Chapter 5

Mutation Testing of Dafny Programs

This chapter presents MutDafny¹, a mutation testing tool for Dafny programs, through which we intend to assess whether current specifications are able to detect artificial faults. Our approach starts with a program that verifies and, while keeping its specification unchanged, generates mutated versions of the program. If the specification accepts a mutated program, i.e., if the program continues to verify, it may reveal potential weaknesses or ambiguities in the specification itself.

5.1 Solution overview

The mutation testing pipeline usually comprises two phases: (i) mutant generation—during which programs with minor syntactic variations from the original one are created—and (ii) mutation analysis—during which the mutants are evaluated, in a software testing context, usually through test suite execution. MutDafny integrates two complementary components for mutant generation: a scanner and a mutator. The mutation analysis stage, in our context, corresponds to the verification of the generated mutants, i.e., the specification acts as a test oracle.

Figure 5.1 illustrates an overview of our solution. MutDafny’s scanner performs an initial program analysis and collects a list of mutation targets, i.e., specific locations of the program where a mutation can be applied, and the data necessary to perform each mutation. The mutator component, in turn, is the one that effectively changes the program’s **AST** to create, one by one, each of the mutants corresponding to each of the previously identified mutation targets. This pipeline is integrated with Dafny’s internals and uses its components to classify the mutants.

A simplified version of Dafny’s verification workflow is depicted in Figure 5.2. The input program provided as a `dfy` file is first parsed into an initial **AST**, process during which syntactic validity is established. After that, the resolver is responsible for a series of semantic and type checks that ensure program validity. Only then does the verification process begin. The verifier, being a complex component, still involves other steps such as the translation to the Boogie intermediate language [2] and, afterwards, to a

¹MutDafny is available at <https://github.com/MutDafny/mutdafny>, accessed 24 Jul 2025.

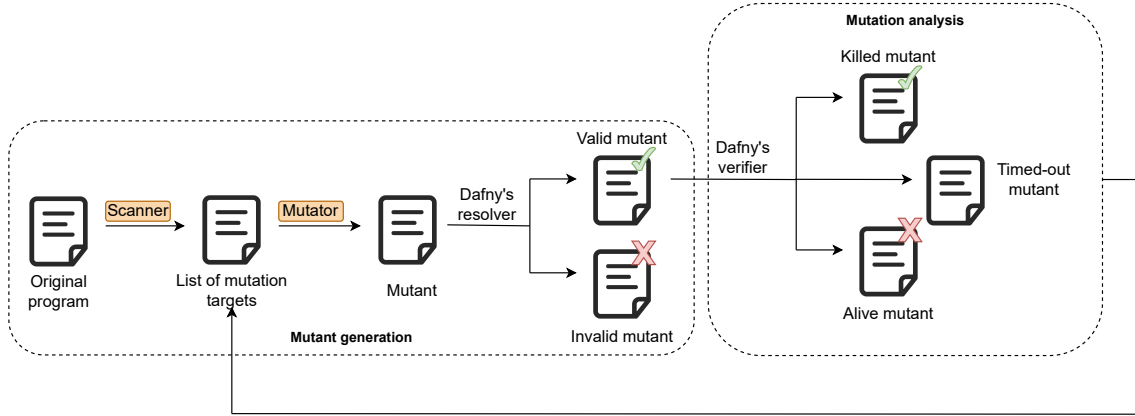


Figure 5.1: Overview of MutDafny's pipeline.

group of SMT formulas that are lastly passed to the Z3 theorem prover [11]. Additionally, the workflow for the compilation from Dafny to other programming languages involves different actions. Considering that MutDafny works at the level between the parser and the verifier, we will not further detail either of these parts of the workflow.

While we try to minimize the generation of invalid mutants, some operators would require complex context analysis to ensure their application would always result in a valid program. Moreover, there is always the risk of edge cases, not accounted for in the implementation of the tool, occurring and resulting in invalid programs. MutDafny applies mutations prior to the program undergoing resolution, which means invalid mutants are later excluded during this process.

At last, all the valid mutants go through Dafny's verifier, and the outcome of the process determines the mutant's final classification status. A mutant that verifies against the original specification hints at a possible specification weakness and, therefore, is said to be alive. An alternative implementation that is correctly detected to be a mutant and fails verification is said to have been killed. Finally, the verifier may not succeed in either accepting or rejecting some of the mutants due to a timeout. Program verification

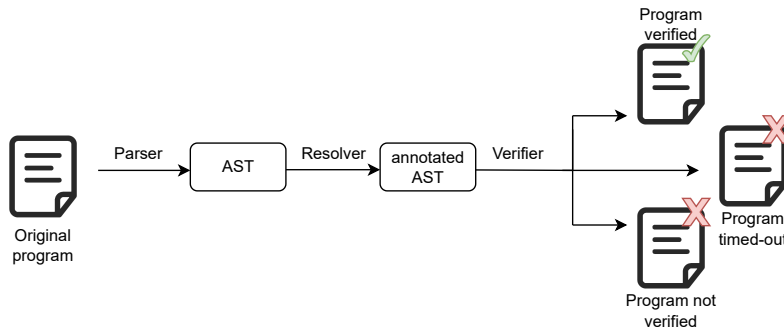


Figure 5.2: Dafny's workflow simplified.

suffers from proof instability, where often small syntactical changes, even if equivalent, can cause a program to stop verifying or to time out due to drastic changes to the SMT solver’s performance [70]. For this reason, we consider timeout mutants as an inconclusive mutation analysis result.

5.1.1 MutDafny’s mutation operators

The operators implemented in MutDafny mostly correspond to the ones listed in Chapter 3 and Chapter 4. However, it is important to highlight three that correspond to groupings of several of the operators presented in Chapter 3:

- **AOR**, **ROR**, **COR**, **LOR**, and **SOR** are grouped into the single Binary Operator Replacement (**BOR**) operator.
- Unary Operator Insertion (**UOI**) encompasses **AOI**, **COI**, and **LOI**.
- Unary Operator Deletion (**UOD**) comprises **AOD**, **COD**, and **LOD**.

We opted for this organization because, within each group, the behaviour is the same, only changing the operators that take part in the operation. In fact, Dafny internally represents the target expression the same way for each operator in the same group, resulting also in the same implementation.

MutDafny, therefore, generates mutants of Dafny programs through the application of a collection of 32 mutation operators. These 32 operators include the 90 from the six programming languages analyzed in Chapter 3 that were deemed relevant in the context of Dafny — which we were able to summarize into 22 (the 30 unique operators mentioned in Chapter 3 considering the groupings described above) — and the 11 operators synthesized from the analysis of program corrections in GitHub Dafny commits, presented in Chapter 4, with one of them (**COR**) being an extension of another previously introduced.

5.2 Mutation target identification

MutDafny’s scanner component is responsible for identifying every possible location in the original program suitable to be mutated, as well as the type of mutation that can be applied in each identified location. This is done during a pre-order traversal of the program’s **AST**. Each mutation operator can be applied to one or a group of specific types of **AST** nodes. As these nodes are reached, the current context and state of the program are further analyzed to evaluate any restrictions that might exist regarding the application of certain operators, and one or more mutation targets corresponding to that program location are stored in a **CSV** file. Each target contains the node’s position where the mutation will be applied, the type of mutation operator to be used, and, depending on the operator, an extra argument providing additional information on how the mutant will be generated. For each node, Dafny internally stores one or more values corresponding to the character position of one of the node’s parsed tokens. These are the values that are later used to find each target node before mutating it.

Figure 5.3 presents an **AST** representation of the example program given in Listing 5.1, consisting of a single assignment operation, and demonstrates how MutDafny compiles the list of mutation targets

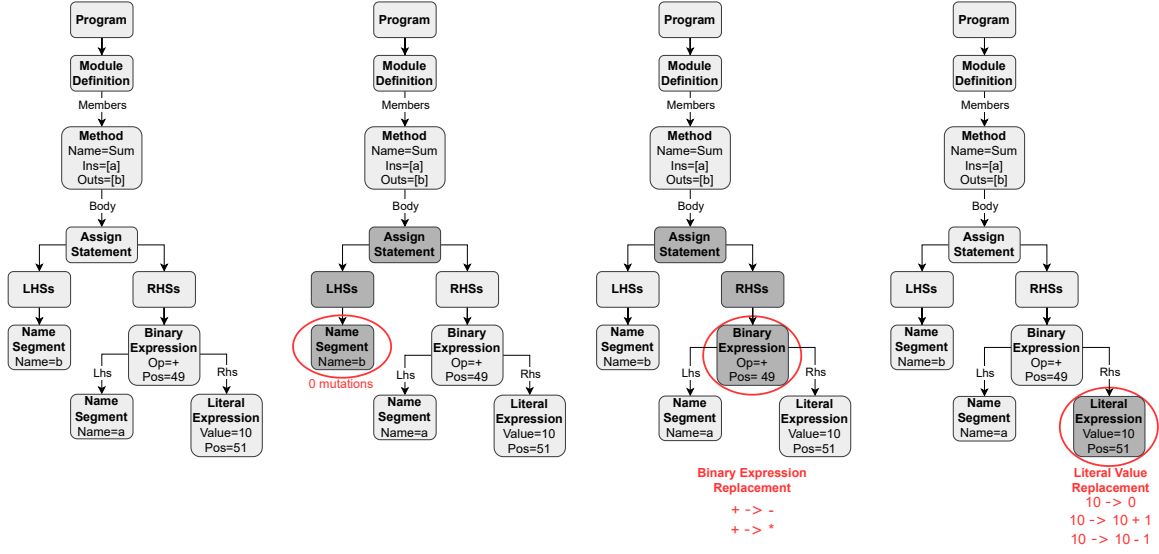


Figure 5.3: Example of the mutation target identification flow for the program in Listing 5.1.

for it. The scanner first visits the **LHS** of the assign statement. It collects information necessary for context analysis of some operators, but because none of our mutation operators mutate the **LHS** of assign statements, no mutation targets are identified for this location. On the Right-Hand Side (**RHS**), it encounters a binary expression, where it is possible to apply a **BOR** mutation. The specific type of mutation to apply (e.g., arithmetic, relational, or conditional) depends on the operator currently used in the target expression. The scanner analyzes the node's data and, upon observing that we are in the presence of an addition, it determines that this operator may be replaced by either a subtraction or a multiplication, as previously delineated in Section 3.3.1.1. Before proceeding with visiting the tree, the two identified mutation targets are stored, as noted in Lines 1 and 2 of Listing 5.2. Next, visiting the binary expression leads to the discovery of a literal expression in the **RHS** of the addition, a node suitable for applying a **LVR** mutation. Once more, the scanner analyzes the node and, due to its value being annotated as an integer, it chooses to increment, decrement, and replace it with 0. The three new targets are stored along with the first two, as noted in Lines 3 to 5 of Listing 5.2.

Listing 5.1: Example of a simple Dafny program

```

1 method Sum(a: int) returns (b: int)
2 {
3   b := a + 10;
4 }

```

Despite the targets being identified as the **AST** is visited, the scanner requires three complete tree traversals, each with its own purpose, to correctly identify all targets. As previously stated, we do not want to insert mutations in parts of the program that correspond to the specification. In order to achieve this, on the one hand, we need to avoid certain statements and expressions when visiting the tree for target identification, such as *assert* statements and *require*, *ensures*, and *invariant* expressions, among

Listing 5.2: List of mutation targets generated for the program in Listing 5.1

```

1 49, BOR, Sub
2 49, BOR, Mul
3 51, LVR, 0
4 51, LVR, 11
5 51, LVR, 9

```

others, as well as any ghost variables, ghost functions, etc. On the other hand, when it comes to non-ghost functions, predicates, and lemmas, it is unclear whether they are being used as an actual part of the program or still as a part of the specification. For this reason, our first tree traversal only visits specification-specific constructs and assembles a catalogue of all functions, predicates, and lemmas that are called in these. This information allows us to avoid visiting specification constructs during the scanning process, thus enabling us to avoid mistakenly identifying targets that are a part of them.

The remaining two tree traversals carried out by MutDafny’s scanner correspond to the target identification process previously described. Because the mutation process is done prior to the program’s resolution, we also try to identify as many targets as possible at this stage of the Dafny pipeline by performing a tree visit. However, for some mutation operators, generating mutation targets involves analyzing the target node’s type information, which is only available post-resolution. This is the case, for example, of the unary operator insertion, which consists of inserting a minus (−) or negation (!) operator in front of every expression where applicable. The minus operator can only be applied in front of a numerical expression, and the negation in front of a boolean one. Therefore, 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 5.4, a portion of the targets are identified during a pre-resolve tree visit, and the remaining are identified during a final post-resolve one:

- **Operators identified during a pre-resolve visit**, requiring two tree visits: **AMR, BBR, BOR, CBE, CBR, LBI, LSR, MMR, ODL, SDL, SLD, SWS, THD, THI, UOD, and VDL**.
- **Operators identified during a post-resolve visit**, requiring three tree visits: **CIR, DCR, EVR, FAR, LVR, MAP, MCR, MNR, MRR, MVR, PRV, SAR, SWV, TAR, UOI, and VER**.

5.3 Mutant generation

Figure 5.5 summarizes the functionality of the mutator component. Each line of the CSV file containing the mutation targets for the program is individually input into an instance of the mutator. The mutation is inserted before the program’s resolution and, following the remaining stages of Dafny’s pipeline, the mutated version of the program, if valid, is either verified, constituting an alive mutant, fails verification, having been killed, or times out, its result being inconclusive.

Regardless of the type of mutation operator to apply, given a target, the first step in generating the corresponding mutant is to traverse the **AST** until the target’s location, i.e., the node to mutate, is reached. MutDafny implements custom visitors to be used for the different types of mutation operators.

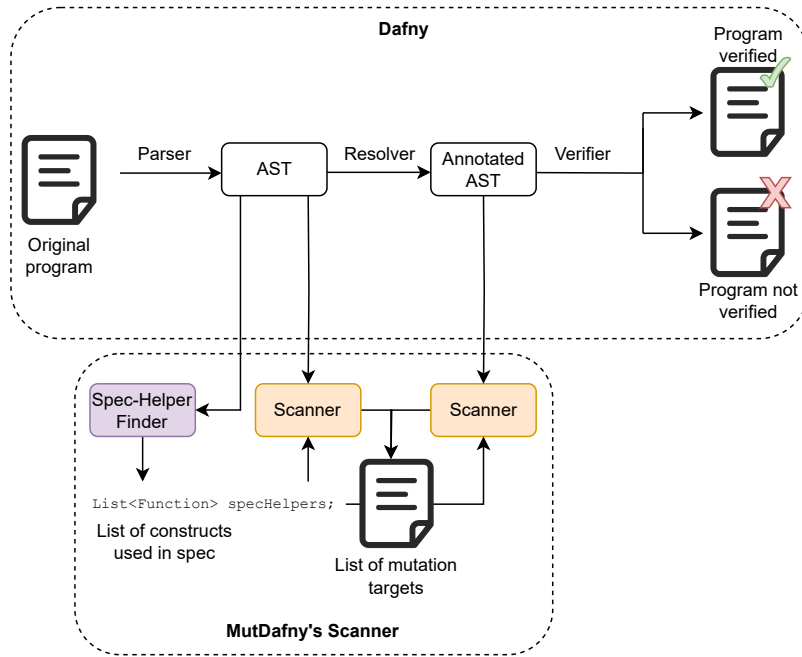


Figure 5.4: Integration of MutDafny's scanner component in Dafny's pipeline.

For instance, when the mutant under generation results from the application of a **BOR** operator, for most nodes, the tree will be parsed as usual, but upon finding a binary expression, the custom visitor for this operator will execute an additional set of actions. The position of the expression's token will be compared to the target's location received by the mutator, and upon finding the target node, the traversal of the tree stops and the mutation process begins.

Mutating the original program involves performing different types of alterations to its **AST**. The simplest change consists of modifying a portion of the target node's data, i.e., one or more of its attributes. This is, for example, the case for **BOR** mutations. As shown in Figure 5.6, after identifying the target binary expression, the mutation itself merely requires changing the value of the expression's operator code to the code provided by the input data's additional argument.

Some mutation operators, however, involve replacing one type of expression or statement with an entirely different type. An example is when a relational or conditional binary expression is replaced with `true` or `false` (**BBR**), which involves the original binary expression being transformed into a literal expression. In practice, and as portrayed in Figure 5.7, the mutator creates a new node of the necessary type containing the appropriate data, and changes the parent node's child reference from the target to the newly created node.

A final group of mutation operators performs the deletion of certain statements or groups of statements. This demands yet another **AST** manipulation approach, specifically the removal of particular nodes, which is achieved through the removal of the target node's reference from the parent in the **AST**

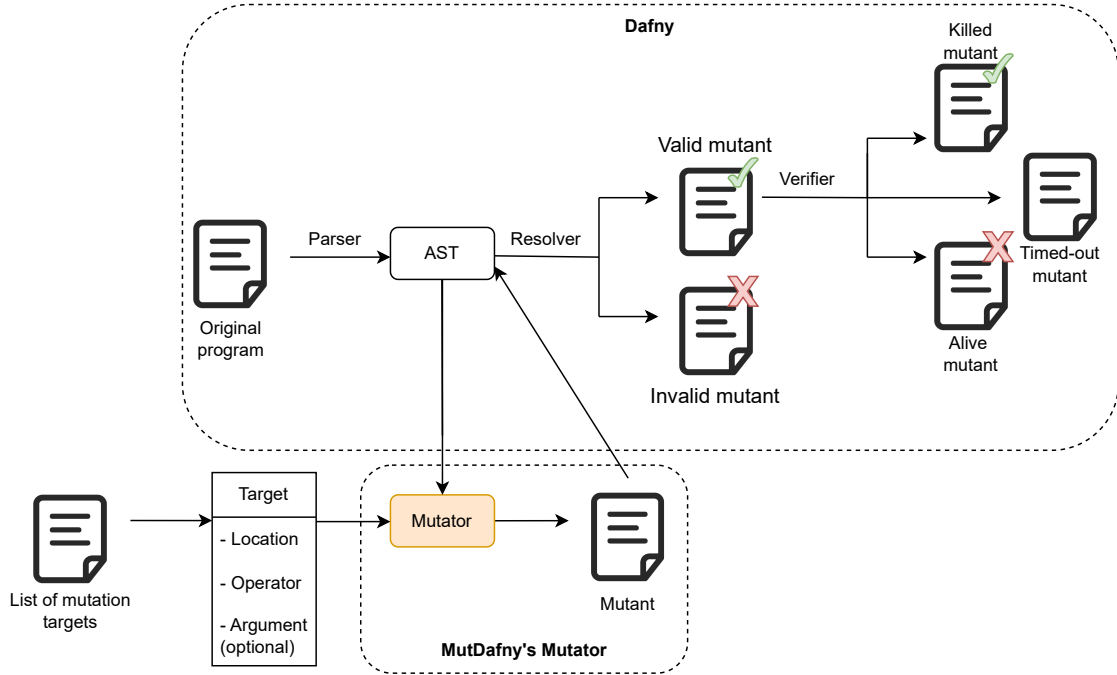


Figure 5.5: Integration of MutDafny's mutator component in Dafny's pipeline.

hierarchical structure. Figure 5.8 represents a program originally containing two assign statements and the application of a **SDL** mutation to the second one.

It is worth noting that, while in the simplest case the mutation is applied directly on the target, these two last types of mutations require, after the target identification, going back to the parent node, which can be of an arbitrary type. This means that our custom visitors for the operators whose implementation falls under these categories need to handle every type of node that may have a target as its child.

5.4 Implementation details

Dafny provides a plugin architecture² for facilitating building tools that are directly integrated with Dafny's flow, without having to replicate parsing, resolution, and verification of Dafny programs. Among other features, it provides an extensible interface for **AST** analysis and manipulation, which allowed us to implement MutDafny as its own standalone tool that uses a plugin to generate mutants without directly changing Dafny's source code.

Figure 5.9 illustrates a simplified view of the subset of the plugin architecture upon which MutDafny is built. The plugin's entry point is the `Microsoft.Dafny.Plugins.PluginConfiguration` abstract class. Here, a set of **AST** rewriters are defined, for later being called by Dafny at the appropriate

²<https://github.com/dafny-lang/dafny/blob/master/docs/DafnyRef/Plugins.md>, accessed 24 Jul 2025.

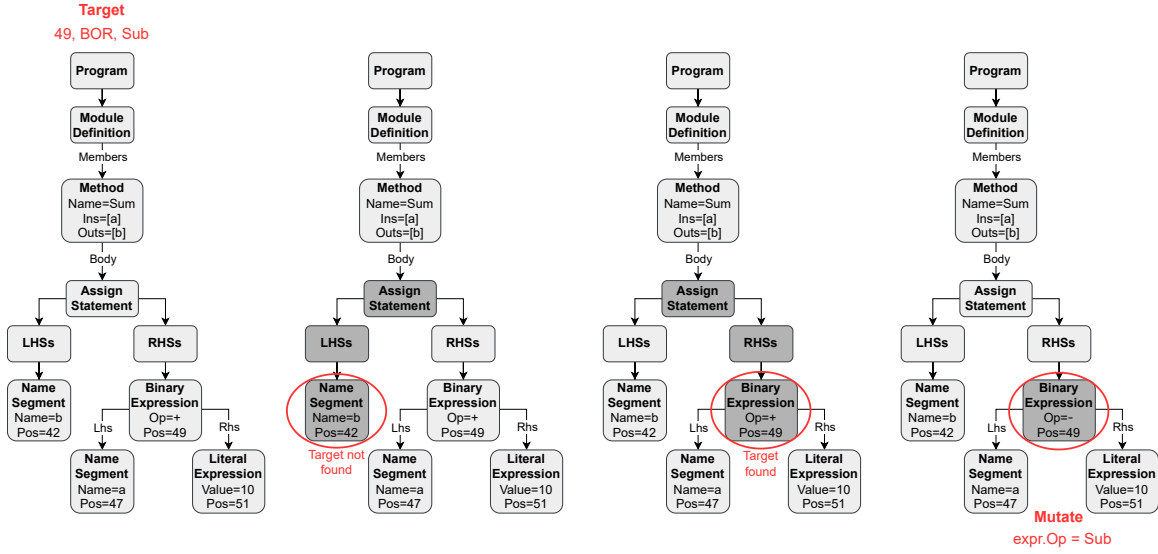


Figure 5.6: Example of the flow for the application of a binary operator replacement mutation.

point in its execution flow. This enables the injection of additional **AST**-related computation steps. Custom rewriters are defined by extending the `Microsoft.Dafny.Plugins.Rewriter` abstract class, which offers several entry point methods for traversing the **AST** at different points of Dafny's execution flow: the `PreResolve(ModuleDefinition)` allows to modify the tree before each of the program's modules starts resolution, the `PostResolve(ModuleDefinition)` provides access to

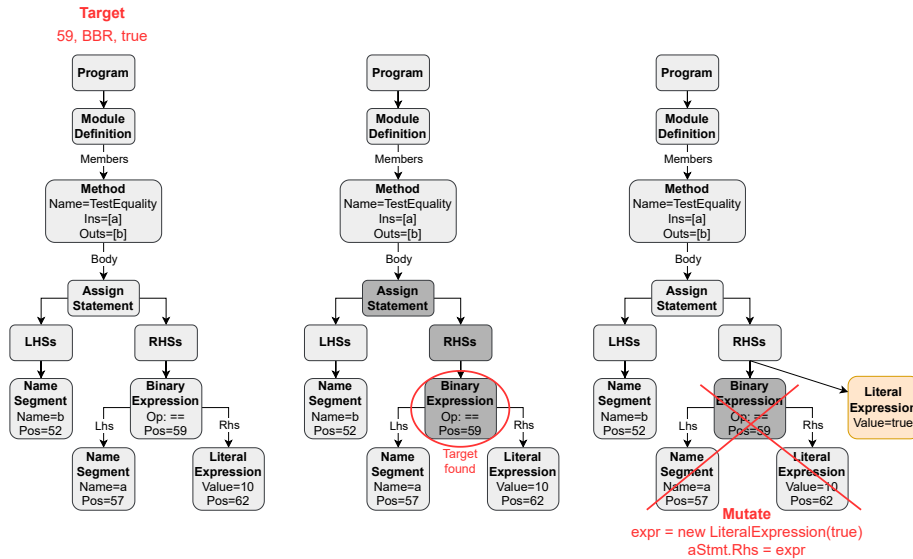


Figure 5.7: Example of the flow for the replacement of a binary expression with a boolean literal.

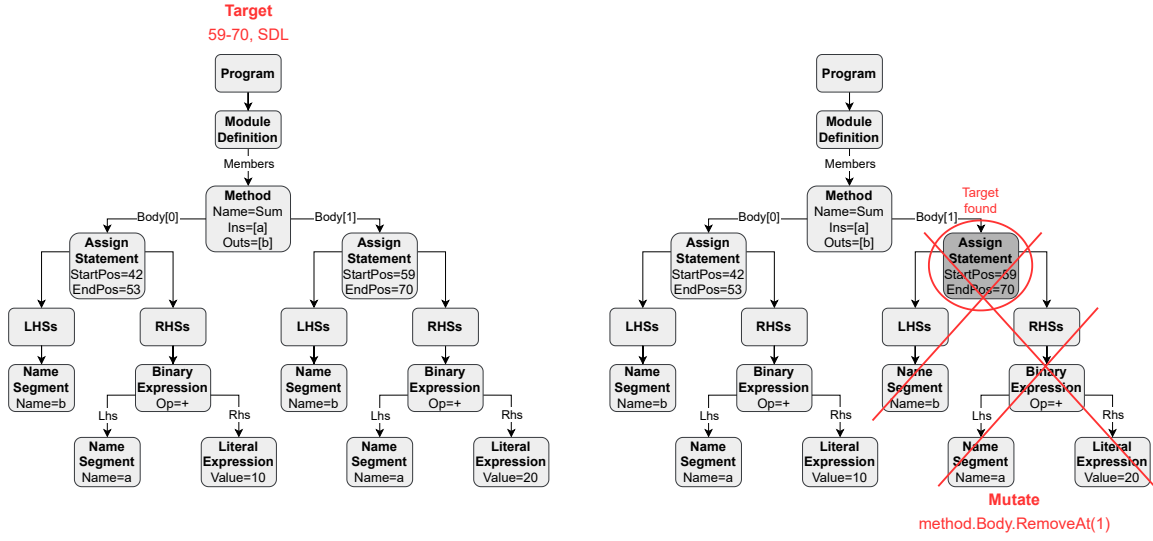


Figure 5.8: Example of the flow for the application of a statement deletion mutation.

each module immediately after its resolution, and `PostResolve(Program)` is called after all modules have been resolved.

MutDafny's scanner and mutator are implemented as two different rewriters. As previously explained, the scanner collects some mutation targets before the program is resolved, and the remaining after this process takes place. This is done by overriding both the `PreResolve(ModuleDefinition)` and the `PostResolve(ModuleDefinition)` methods, each executing one complete visit of the tree. Despite being an extension of the `Rewriter` class, the scanner does not change the tree, but is merely used to analyze it. The mutator, in turn, due to only inserting mutations prior to program resolution, only overrides the `PreResolve(ModuleDefinition)` method.

Because the plugin works as an additional step in Dafny's regular execution flow, after the plugin executes, the program is either going to go through verification or compilation. For this reason, it makes sense to use this in our favor and perform mutation analysis of a particular mutant right after it is gen-

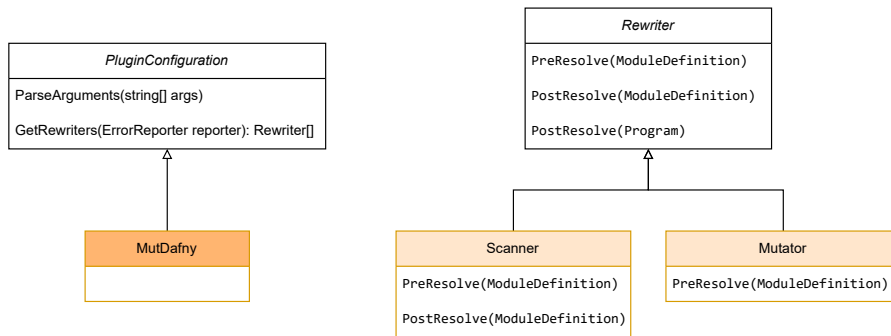


Figure 5.9: Overview of the plugin architecture.

erated. Each mutant is, therefore, generated within one call to Dafny’s `verify` command, corresponding to one instantiation of the plugin. This differs from the usual flow of other mutation testing tools in that, typically, all mutants are generated before the mutation analysis process begins. The program’s scanning for mutation targets is done in a first instantiation of the plugin and, while this initial process only involves analyzing the program, the way to execute it is also by using the plugin in the context of a call to the `verify` command, although in this case the verification process is not really necessary.

As mentioned, the plugin runs inside of a Dafny execution, so it must be passed as an argument to a Dafny command. We override the `ParseArguments` method in order to instruct which type of rewriter — scanner or mutator — to use in the plugin in a particular execution. Additionally, considering the target scanning and mutant generation steps occur in different executions, we are also able to use the plugin’s arguments to pass the data of the mutation target corresponding to the mutant currently under generation. Listing 5.3 shows how the plugin can be used to generate the list of all mutation targets for a particular program (living in a file named `program.dfy`). By default MutDafny generates mutants for all 32 mutation operators. Listing 5.4 shows how MutDafny is instructed to generate a specific mutant of the same program. MutDafny automates the entire process of performing mutation testing on a Dafny program through a shell script³, as exemplified in Listing 5.5. This script first generates the list of mutation targets for the program, and then iterates over it, generating and verifying each of the program’s mutants by passing each target’s data, one by one, to the plugin.

Listing 5.3: Usage of MutDafny’s plugin scanning component.

```
1 dotnet ./dafny/Binaries/Dafny.dll verify program.dfy \
2   --solver-path ./dafny/Binaries/z3 \
3   --plugin ./mutdafny/bin/Debug/net8.0/mutdafny.dll,scan
```

Listing 5.4: Usage of MutDafny’s plugin mutator component for the mutation target listed in Line 1 of Listing 5.2.

```
1 dotnet ./dafny/Binaries/Dafny.dll verify program.dfy \
2   --solver-path ./dafny/Binaries/z3 \
3   --plugin ./mutdafny/bin/Debug/net8.0/mutdafny.dll,"mut 49 BOR Sub"
```

Listing 5.5: Running MutDafny.

```
1 bash run.sh program.dfy
```

5.5 Summary

This chapter presented details on the inner workings of MutDafny through which it achieves mutation testing of Dafny programs. We describe its two-phase pipeline: mutant generation, which further divides into mutation target identification and effective mutation insertion, and mutation analysis, corresponding to mutant verification. Moreover, we outline its integration with Dafny through a plugin architecture. Chapter 6 validates this design on real-world Dafny programs.

³<https://github.com/MutDafny/mutdafny/blob/main/run.sh>, accessed 24 Jul 2025.

Chapter 6

Empirical evaluation

We evaluate the efficiency of MutDafny and the impact of code mutations on the ability of Dafny specifications to detect them by failing program verification. This chapter details our empirical evaluation procedure¹, presents the results collected during it, and discusses the key findings. In a nutshell, in this chapter we aim to answer **RQ3**: *How efficient is MutDafny at creating mutants, and how many are generated overall and by mutation operator?*, and **RQ4**: *How do Dafny specifications perform at killing mutants?*

6.1 Experimental subjects

For our empirical evaluation, we utilize the programs of DafnyBench [40], the current most extensive Dafny verification benchmark, as our experimental subjects. As described in Section 4.1, the benchmark comprises 785 programs, both synthesized and extracted from real-life projects, with a total of about 53,000 lines of code and, on average, 2.12 methods, 1.03 functions, and 1.40 lemmas.

At the time of writing, 40 programs out of the 785 are not verifiable using Dafny 4.10.0, more specifically the version corresponding to commit hash 7159879² and Z3 4.12.6. Since the mutation testing concept involves an initial passing test suite, before employing MutDafny, we attempted to verify each of the 785 programs using the same version of Dafny that supports our tool. We deemed unfit for the application of mutation testing any program that (a) contained any errors regarding program validity, (b) that timed out, (c) that had at least one verification error, or (d) that, while not exactly identifying any verification errors, also did not verify any element of the program (e.g., Dafny program verifier finished with 0 verified, 0 errors). Scenario (a) occurs in three programs e.g., `groupTheory_tmp_tmppmmxvu8h_assignment1`, scenario (b) occurs in nine (e.g., `WrappedEther`), scenario (c) in 24 (e.g., `Clover_insert`), and scenario (d) occurs in five (e.g.,

¹All the scripts and artifacts necessary to replicate our experiment are available in <https://github.com/MutDafny/mutdafny-study-data>, accessed 24 Jul 2025.

²<https://github.com/dafny-lang/dafny/tree/71598790179cf5c37947c0953d295c12d9737335>, accessed 24 Jul 2025.

DafnyExercises_tmp_tmpe6qyevja_QuickExercises_testing2). We identified that scenario (d) occurred, for example, in programs that defined a set of methods marked as `extern`—implemented in an external source in one of the languages Dafny compiles to—and that, though a specification was provided, contained no body to be verified. Table 6.1 summarizes the excluded programs and the reason for their exclusion. This elimination process left us with 745 valid experimental subjects.

Table 6.1: Non-verifiable programs in the DafnyBench [40] benchmark.

Program	# Verified	# Errors	# Timeout
BelowZero	4	1	0
BinaryAddition	6	2	2
Clover_insert	1	1	0
dafny_examples_tmp_tmpe8qotd4ez_leetcode_0027-remove-element	1	1	0
DafnyExercises_tmp_tmpe6qyevja_QuickExercises_testing2	0	0	0
dafny_experiments_tmp_tmpe29_3_3i_circuit	23	0	1
dafny-language-server_tmp_tmpekir0kenl_Test_hofs_SumSum	10	1	0
Dafny-Practice_tmp_tmpehnm4ovh_Pattern Matching		<i>Unhandled exception</i>	
DafnyPrograms_tmp_tmpe74_f9k_c_automaton	2	3	0
DafnyProjects_tmp_tmpe2acw_s4s_CombNK	5	1	0
DafnyProjects_tmp_tmpe2acw_s4s_Power	4	1	0
dafny-synthesis_task_id_566	12	0	1
dafny-synthesis_task_id_632	4	0	1
FMSE-2022-2023_tmp_tmpe6_x_ba46_Lab2_Lab2	16	1	0
FMSE-2022-2023_tmp_tmpe6_x_ba46_Lab3_Lab3	2	3	0
groupTheory_tmp_tmpeppmmxvu8h_assignment1		<i>1 resolution/type errors detected</i>	
iron-sync_tmp_tmpe49o3tyz_Impl_CommitterCommitModel	0	0	0
llm-verified-eval_tmp_tmpe2deqn_i_dafny_3	4	1	0
MFES_2021_tmp_tmpeuljn8zd9_TheoreticalClasses_Power	6	1	0
MFS_tmp_tmpepmnu354t_Praticas_TP9_Power	7	1	0
Prog-Fun-Solutions_tmp_tmpe7_gmnz5f_extra_mod	1	0	1
Program-Verification-Dataset_tmp_tmpegbdrlnu__Dafny_advanced examples_Percentile	9	1	0
Program-Verification-Dataset_tmp_tmpegbdrlnu__Dafny_algorithms and leetcode_examples_bubblesort	5	1	0
Program-Verification-Dataset_tmp_tmpegbdrlnu__Dafny_algorithms and leetcode_examples_relativeOrder	2	1	0
Program-Verification-Dataset_tmp_tmpegbdrlnu__Dafny_algorithms and leetcode_leetcode_ReverseLinkedList	29	0	1
Program-Verification-Dataset_tmp_tmpegbdrlnu__Dafny_algorithms and leetcode_leetcode_stairClimbing	23	3	0

(table continues)

Table 6.1: Non-verifiable programs in the DafnyBench [40] benchmark (continued).

Program	# Verified	# Errors	# Timeout
Program-Verification-Dataset_tmp_tmpgbdrlnu__Dafny_algorithms and leetcode_lib_seq	51	0	1
Program-Verification-Dataset_tmp_tmpgbdrlnu__Dafny_algorithms and leetcode_math_pearson	18	1	0
Program-Verification-Dataset_tmp_tmpgbdrlnu__Dafny_from_dafny main_repo_dafny4_ACL2-extractor	<i>1 resolution/type errors detected</i>		
Program-Verification-Dataset_tmp_tmpgbdrlnu__Dafny_pregel algorithms_skeleton_nondet-permutation	6	2	0
protocol-verification-fa2023_tmp_tmpw6hy3mjp_demos_ch04_toy_consensus	9	5	0
protocol-verification-fa2023_tmp_tmpw6hy3mjp_exercises_chapter04- invariants_ch03exercise03	6	1	0
QS_BoilerPlate1_tmp_tmppa29vtz9__Ex2	8	1	0
sat_dfy_tmp_tmppfcyj8am9_dfy_Seq	2	1	0
t1_MF_tmp_tmppi_sqie4j_exemplos_colecoes_sequences_ex3	1	1	0
test-generation-examples_tmp_tmptwyqofrp_IntegerSet_dafny_Utils	0	0	0
test-generation-examples_tmp_tmptwyqofrp_ParamTests_dafny_Utils	0	0	0
Trab1-Metodos-Formais_tmp_tmp_8fa4trr_circular-array	15	0	1
veri-titan_tmp_tmppbg2iy0kf_spec_crypto_fnnt512	0	0	0
WrappedEther	61	0	1

The 745 program files that constitute our experimental subjects gather a total of 2,229 methods, functions, and similar constructs: 1,317 of these have no pre-conditions, 1,066 no post-conditions, and 829 have neither. This does not necessarily imply that the specification of the programs containing these constructs 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 a specification, mutations affecting them can still contribute to measuring the quality of the overall program’s specification.

6.2 Mutation operators

For clarity of the following sections, we use the 32 operators that MutDafny implements. **AMR**, **BBR**, **BOR**, **CBR**, **CIR**, **EVR**, **LBI**, **LSR**, **LVR**, **MAP**, **MMR**, **MNR**, **MRR**, **ODL**, **PRV**, **SDL**, **SLD**, **THD**, **THI**, **UOD**, **UOI**, and **VDL** were proposed for other mutation testing tools. **CBE**, **DCR**, **FAR**, **MCR**, **MVR**, **SAR**, **SWS**, **SWV**, **TAR**, and **VER** were the operators newly found in Chapter 4.

6.3 Experimental procedure

Our experimental procedure consists of running MutDafny against each program from DafnyBench that was deemed valid for applying mutation testing. While the default of MutDafny’s scan command is to

generate every possible mutation target for a program using all of the available mutation operators, we also allow generating a list of mutation targets for a single specific operator, passed as an argument to the plugin, and, consequently, only generating mutants resulting from the application of said mutation operator. Since it is crucial to measure the performance and effectiveness of each implemented mutation operator individually, for the most part, we conducted our experiments individually for each operator. To be more specific, for each experimental subject, we perform mutation testing over it by applying each of our operators individually in successive iterations. This phased process also facilitated an iterative collection of results in parallel with the tool’s implementation.

The iterative result collection leads both MutDafny’s scanner and mutator to be run 32 times for each program, once for each implemented mutation operator. For the mutator component, the results we are able to collect using this phased approach are equivalent to the ones we would have obtained by running the tool with all the operators enabled, since each mutant is generated individually and, after the 32 executions, every possible mutant has been generated. However, running the scanner with every operator enabled is not the same as running it once for each, because each execution traverses the **AST**, so we are facing one traversal of the tree against 32. In order to attain more faithful results regarding the tool’s runtime when every operator is considered, in addition to the 32 scans executed for each program, we also collect data relative to a default scan, which collects all targets for all mutation operators in a single execution.

The tight integration of our tool in Dafny’s execution flow introduces challenges in measuring its runtime. Determining the end-to-end runtime is essential for providing an accurate depiction of the usage of the tool. On the other hand, it is also of interest to precisely quantify what portion of the runtime is related to our implementation and how much of it, in reality, corresponds to Dafny’s usual execution time. To achieve this, we pinpointed the specific locations of Dafny’s source code where the plugin’s methods are called and wrapped them inside an elapsed time counter, as exemplified in Listing 6.1. Since MutDafny relies on Dafny’s resolver and verifier for different tasks, and we want to analyze the impact of each on the overall tool’s runtime, we do the same for the locations where each phase of Dafny’s pipeline begins and ends, as shown in Listing 6.2.

Listing 6.1: Alterations to Dafny’s source code to measure plugin’s runtime

```

1 public ModuleSignature Resolve(ModuleResolver resolver, CompilationData compilation) {
2     var module = ModuleDef;
3     var rewriters = RewriterCollection.GetRewriters(
4         resolver.Reporter, resolver.ProgramResolver.Program
5     );
6
7     + MyStopwatch.Start();
8     foreach (var rewriter in rewriters) {
9         rewriter.PreResolve(module);
10    }
11    + MyStopwatch.Stop();
12    + MyStopwatch.IncreasePluginTime();
13 }

```

Listing 6.2: Alterations to Dafny’s source code to measure each execution pipeline’s phase runtime

```

1 public Compilation() {
2 +   MyStopwatch.Start();
3   ParsedProgram = ParseAsync();
4 +   MyStopwatch.Stop();
5 +   MyStopwatch.SetParsingTime();
6
7 +   MyStopwatch.Start();
8   Resolution = ResolveAsync();
9 +   MyStopwatch.Stop();
10 +   MyStopwatch.IncreaseResolutionTime();
11 }
12
13 public static async Task<int> HandleVerification(DafnyOptions options) {
14   var resolution = await compilation.Resolution;
15
16 +   MyStopwatch.Start();
17   if (resolution != null) {
18     Subject<CanVerifyResult> verificationResults = new();
19     compilation.VerifyAllLazily().ToObservable().Subscribe(verificationResults);
20   }
21 +   MyStopwatch.Stop();
22 +   MyStopwatch.SetVerificationTime();
23 }

```

To accelerate the experiment runtime, we parallelized some workflow components. For the scanner component, each job consists of scanning one program for mutation targets using one mutation operator, which means we are able to simultaneously generate the list of targets for multiple of the 745 programs that constitute our experimental subjects. For the mutator component, each job consists of generating the mutants for one program, similarly allowing us to apply different mutations to several of the programs in the dataset and run mutation analysis on the resulting mutants at the same time.

The entire experiment, consisting of running the scanner individually for each mutation operator and once with all enabled, and running the mutator for each identified mutation target, was executed 10 times, as it is commonly done in other studies, to, for example, measure execution times [56]. Following this process, for each program and mutation operator, we removed 20% of the records corresponding to the highest scanning time and 20% to the lowest as suggested by others [56], i.e., the two highest and the two lowest. We did the same for the data relative to each generated mutant. The exclusion of extreme values aims to eliminate any outliers caused by, e.g., cache effects, and increase the robustness of our results. The values reported in our analysis reflect the mean of the remaining data records.

The experimental procedure was run on a cluster running Ubuntu 20.04 operating system, Linux kernel version 5.15, with 1GB of RAM, and an Intel(R) E5-2450 CPU at 2.1GHz. As done for the verification of DafnyBench’s programs, we use the version of Dafny 4.10 corresponding to commit hash 7159879³, and Z3 4.12.6.

³<https://github.com/dafny-lang/dafny/tree/71598790179cf5c37947c0953d295c12d9737335>, accessed 24 Jul 2025.

6.4 Experimental metrics

The output dataset resulting from running this experiment comprises a collection of mutants of the DafnyBench programs, one CSV file (`scan-data.csv`) containing runtime measurements and mutation target data for all of the executed program scans, and one CSV file (`mut-data.csv`) containing runtime measurements for all of the generated mutants, alongside data identifying the applied mutation and its analysis status:

- `scan-data.csv`:
 - `program_name`: the input program from DafnyBench.
 - `mutation_operator`: the acronym of the used mutation operator, or ALL if they were all simultaneously considered.
 - `parsing_time`: the time in milliseconds for Dafny to parse the input program.
 - `plugin_time`: the time in milliseconds for the plugin to collect the mutation targets.
 - `resolution_time`: the time in milliseconds for Dafny to resolve the input program.
 - `verification_time`: the time in milliseconds for Dafny to verify the input program.
 - `number_of_targets`: the number of mutation targets found in the program for the operator in question.
 - `scan_time`: the total time in milliseconds to run MutDafny’s scanner.
- `mut-data.csv`:
 - `program_name`: the program from DafnyBench under mutation.
 - `mutation_position`: the position of the **AST** node to be mutated.
 - `mutation_operator`: the mutation operator applied to generate the mutant.
 - `mutation_argument`: the additional mutation argument, if any.
 - `status`: the result of running mutation analysis on the mutant: invalid, survived, timeout, or killed.
 - `parsing_time`: the time in milliseconds for Dafny to parse the input program.
 - `plugin_time`: the time in milliseconds for the plugin to generate the mutant.
 - `resolution_time`: the time in milliseconds for Dafny to resolve the mutant.
 - `verification_time`: the time in milliseconds for Dafny to verify the mutant.
 - `mut_time`: the total time in milliseconds to run MutDafny’s mutator.

We are interested in using this data to analyze the overall number of mutants our tool is capable of generating, and their distribution regarding the mutation operator used for the generation of each. We use the data on mutant status to compute the mutation score of the programs’ specifications, as well as to calculate the percentage of invalid mutants our tool generates. Finally, we assess the overall tool’s runtime alongside the runtime of each stage: plugin execution, mutant generation, and mutation analysis. Since the collected data regarding runtime refers to each individual execution of the scanner or mutator, we need to compute the runtime values at the program level using these smaller component runtimes.

- For each program p , its mutation score is given by:

$$score = \frac{\# \text{ mutants with status killed}}{\# \text{ mutants} - \# \text{ mutants with status invalid} - \# \text{ mutants with status timeout}}$$

- For each program p , the time to generate all of its mutants using operator op is given by:

$$runtime = scan_time + \text{sum}(\text{mut_time} - \text{verification_time})$$

Where $scan_time$ is the result from running the scanner on program p with operator op , and mut_time and $verification_time$ are the results from running the mutator on each target generated by the previous scan.

- For each program p , the time to generate all of its mutants is given by:

$$runtime = scan_time + \text{sum}(\text{mut_time} - \text{verification_time})$$

Where $scan_time$ is the result from running the scanner on program p with operator ALL , and mut_time and $verification_time$ are the results from running the mutator on each target generated by the previous scan.

- For each program p , the time to perform mutation testing over it is given by:

$$runtime = scan_time + \text{sum}(\text{mut_time})$$

Where $scan_time$ is the result from running the scanner on program p with operator ALL , and mut_time is the result from running the mutator on each target generated by the previous scan.

In the computation of the mutant generation runtime, we do not include the time spent verifying the mutant because this corresponds to the mutation analysis runtime and only takes place subsequently. The verification time is, however, considered when computing the total runtime of the tool. Regarding the resolution time, we deem this should be accounted for as part of the mutant generation time due to its role in determining mutant validity.

6.5 Threats to validity

Based on the guidelines reported by Wohlin et al. [67], we have taken all reasonable steps to mitigate the effect of potential threats described in detail in this section.

6.5.1 Threats to construct validity

We identify four key constructs in our experimental process: the tool's integration with Dafny, its efficiency and mutant generation capacity, and a specification's effectiveness in detecting mutations to the

original code. For each of these constructs, we present precise definitions and a set of measures to ensure the experiment strictly follows these definitions.

Regarding the tool’s integration with Dafny’s standard workflow, we detail both the tool’s pipeline and how it is integrated into Dafny’s, pinpointing the specific points where this happens, e.g., the point immediately before resolution for the insertion of a mutation, the point immediately after resolution for determining the validity of the generated mutant, the coincidence of the mutation analysis process with the verification of the mutant, etc. The exact definition of when each process begins and ends will enable us to properly analyze the collected results. Following this model’s definition, we also formalize how the runtime of each stage is computed, presenting mathematical formulas that capture the measurement logic. This corresponds to the foundations for quantifying tool’s efficiency. Additionally, though we collected our results through parallelized executions, we set aside each individual stage’s runtime and report the aggregate values as if the experiments had been run sequentially. This ensures we avoid presenting misleading results that depend on specific parallelization advantages and can easily vary with a different execution setup.

In order to secure meaningful results when it comes to the tool’s mutant generation capacity, we run our experiments individually on each operator. The overall results may be highly influenced by a subset of the mutation operators exhibiting a larger weight, for instance, ones that can be applied more often and, therefore, generate more mutants, or whose application suffers from a considerably higher runtime. While these potential biases will still influence the overall results, the analysis of the results collected for each individual operator facilitates the interpretation of the former.

Finally, for the assessment of a specification’s effectiveness in detecting mutations, we import the construct of mutation score, deeply standardized in mutation testing literature. Furthermore, to properly ensure precision in the computation of the mutation score of Dafny programs, we meticulously define what the status of a mutant after going through mutation analysis is considered to be for each possible result, as well as when we consider the specification to have indeed detected a mutation.

6.5.2 Threats to internal validity

We identify several threats to internal validity inherent in the nature of the experiment, and we adopt a series of measures to ensure that our results are not distorted by any hidden factors. We pay close attention to performance maturation, repeated-testing bias, and experimental subject mortality.

The first threat arises from CPU optimizations that can artificially reduce runtime in repeated executions compared to initial runs. We mitigate this warm-up distortion and are able to achieve more accurate results by executing the entire experiment 10 times and removing runtime outliers. Another common threat occurs upon the repeated application of an experimental procedure to the experimental subjects, their state being continuously changed. In the case of our experiment, we successively apply mutations to the programs of DafnyBench. However, by saving the mutant in a separate program file, we are able to leave the original program unchanged and always apply new mutations to the same base program. Finally, we deal with programs not getting to the end of the pipeline in different ways. Invalid

and non-verifiable experimental subjects were excluded prior to the experiment in order to avoid mistakenly identifying possible mutants of those programs as invalid or killed. Mutants whose verification naturally times out are tracked and, later, considered as killed for mutation score computation purposes. Additionally, the execution of several parts of the experiment in parallel implies that occasionally jobs would time out as well, which were monitored and resubmitted until finalized.

6.5.3 Threats to external validity

The ability to generalize the results from this study to external environments lies in the representativeness of our experimental subjects. We can trust DafnyBench to be a representative subset of the Dafny program population due to it comprising both synthesized and real-life programs, the latter constituting 75% of the dataset. Not only do real Dafny programs account for the majority of the dataset, but considering these were collected from a GitHub scrape with minimum filtering, such as the removal of programs that did not verify or that presented no specification, the dataset also contains a high portion of the overall population of Dafny programs publicly available. The programs have varying levels of complexity, and they all contain specifications, allowing us to evaluate their response to the tool.

6.6 Experimental results

Given the data collected in the experimental procedure previously described, we now proceed to analyze and discuss the results, attempting to answer our research questions. For RQ3, we focus our analysis on the number and types of mutants generated for DafnyBench’s programs and on runtime data, as these reflect the tool’s mutation generation rate. For RQ4, we analyze the status of the mutants and mutation score data, as this is the measure that will enable us to understand the volume of mutants identified by the Dafny specifications.

6.6.1 Answer to RQ3, How efficient is MutDafny at creating mutants, and how many are generated overall and by mutation operator?

Figure 6.1 shows the distribution of the number of mutants generated per program using each of the specific operators implemented in MutDafny. In addition, the first column of Table 6.3 provides the total number of mutants generated across the 745 experimental subjects, also segregated by mutation operator. In total, MutDafny generated over 87,000 mutants for the programs of DafnyBench.

We can clearly highlight the **BOR**, **EVR**, **LVR**, **UOI**, and **VER** operators as being responsible for the generation of a higher number of mutants per program. In fact, for around half of the programs, between five and 20 mutants are generated using each of these operators, and for some, the number even goes up to 200. Table 6.3 also shows that these are the operators that overall generate the most mutants, each generating over 10,000. This is somewhat expected since these operators target some of the syntactic constructs that most commonly occur in programs. Namely, the **BOR** operator targets, among

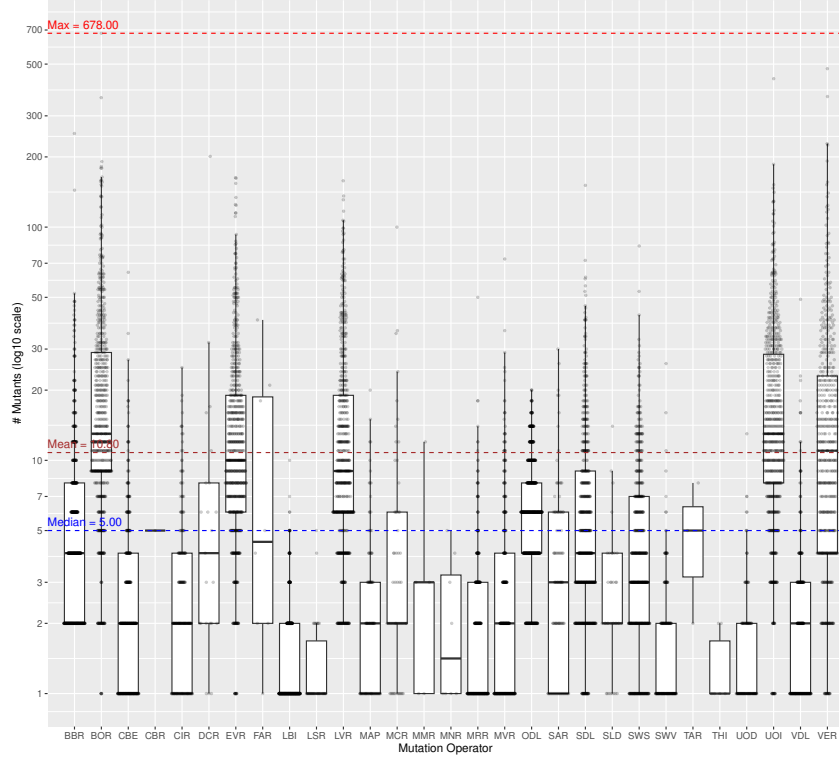


Figure 6.1: Distribution of number of mutants per program and per mutation operator.

others, arithmetic, relational, and conditional binary expressions; the **LVR** targets primitive literals, from which numerical ones are highly prevalent; both the **EVR** and **UOI** target expressions as a whole, which constitute the basic unit for program writing; and, finally, **VER** targets variables.

Three of the operators — **AMR**, **PRV** and **THD** do not appear in this plot or any of the following because no mutation targets, i.e., no suitable locations, were found for the application of these two operators in any of the 745 programs of the dataset. This happens because the operators can only be applied in very specific scenarios. The first targets methods whose sole behaviour is returning the values of class fields. The second requires the existence, in the same scope, of two variables that are instances of different classes, children of a common parent class. The third entails an access to a class field using a `this` keyword within a method with an argument with the same name as the field. These apparently occur so rarely in Dafny that they are not even contemplated in the dataset, which is reasonable considering these operators were all proposed for a purely **OO** programming language.

Aside from these three, we can highlight the **CBR**, **MMR**, **MNR**, **TAR**, and **THI** operators as ones that generated a very low number of mutants: in total, five, 20, 18, 15, and eight, respectively. Similarly to the first, these are not prevalent due to their specificity in application context, the **MMR** and **THI** also being **OO** operators. The **CBR** operator targets match case statements, which are fairly common in Dafny, but requires the utilization of a default case, which rarely appears in DafnyBench’s programs. **TAR** involves tuples, which only occur in three of the dataset’s programs, and the **MNR** requires a call

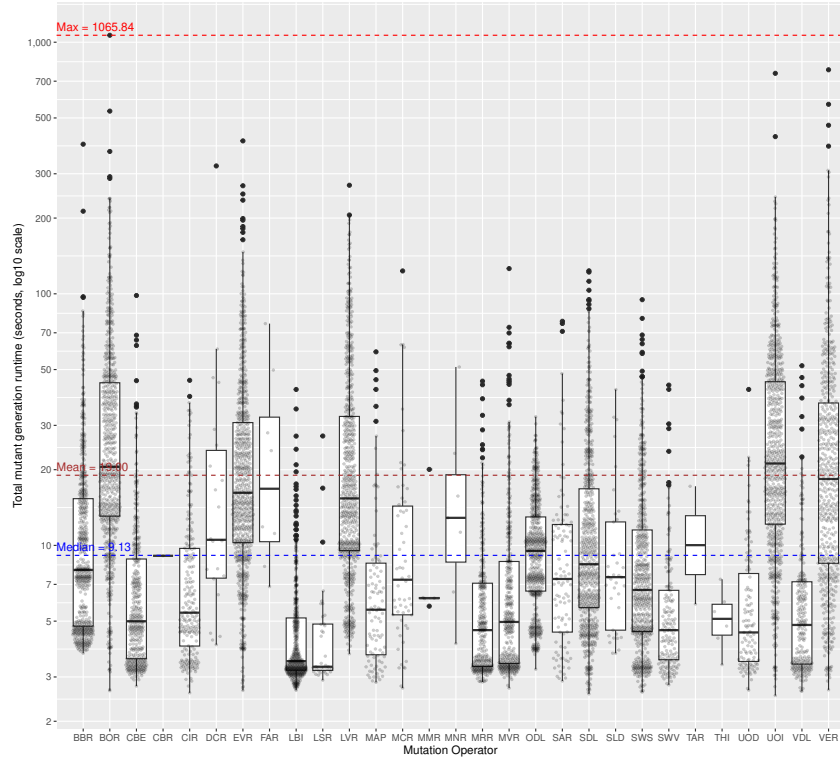


Figure 6.2: Distribution of total mutant generation runtime per mutation operator.

to a class method where the return value is also an object of the same class. Except for **TAR**, these operators were collected from other programming languages, and while in Chapter 3 we showed they could be applied to Dafny, they appear to be rarely used in this context.

Figure 6.2 illustrates the distribution of the mutant generation runtime per program using each of MutDafny’s mutation operators. In comparison to Figure 6.1, this plot demonstrates a similar outline, namely the same five operators that were identified as the ones responsible for the generation of a higher number of mutation operators — **BOR**, **EVR**, **LVR**, **UOI**, and **VER** — are also the ones that, when employed, lead to a higher mutant generation runtime. This stems from the fact that more executions are required in order to generate more mutants, inevitably increasing the total runtime. For most programs, the runtime of the mutant generation process using one of these five operators varies between 10 and 50 seconds, while the overall median considering any of the implemented operators is 9.13 seconds.

On the other hand, Figure 6.3, which normalizes the previous results by mutant count, capturing the distribution of the average mutant generation time per operator, shows that the five most used operators, which also present a higher total mutant generation runtime, actually seem to present among the lowest average mutant generation runtimes. These lower runtimes, for most cases lower than two seconds per mutant, reflect the operators’ more straightforward implementation. We can highlight the **MMR** and **MNR** operators as the ones with higher average mutant generation runtime. We believe the increased runtime is due to the verification overhead of the scanning process. This cost is present in all executions,

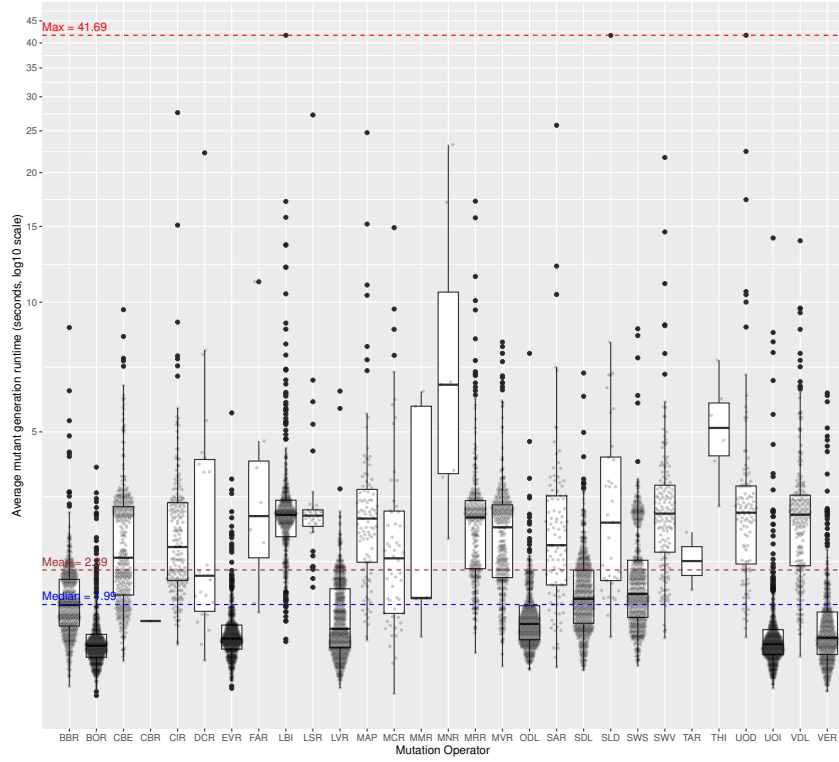


Figure 6.3: Distribution of average mutant generation runtime per mutation operator.

but in most cases it is not so impactful as it is split among the several mutants generated. In most programs, however, both these operators only generate one single mutant, causing it to bear the entirety of the scanner runtime.

Figure 6.4 exhibits the distribution of the runtime of the mutation testing process of each Dafny-Bench program. The median total runtime across all experimental subjects is 243.12 seconds, i.e., four minutes, while the mean is notably higher at 660.95 seconds, almost triple. This suggests that a subset of the programs suffered from a significantly higher mutation testing runtime, in the worst case, close to eight hours to generate and verify all of the mutants for one single program. We can, therefore, consider the median as a more representative value of the total runtime value under typical conditions, with about half of the results falling between two and 10 minutes.

The mutant generation and mutation analysis processes, the two individual phases comprised in the mutation testing pipeline, present similar median runtimes. In the case of mutation analysis, however, we once again observe a mean runtime over three times higher. Table 6.2 presents the three programs with the higher mutation testing runtimes, corresponding to the only programs with this value exceeding 10,000 seconds. This data shows that the cases with the highest overall runtime also correspond to those with the highest mutation analysis runtime, and, though the mutant generation runtime is also considerably above average, it is still one order of magnitude lower. We can conclude that, for most of the

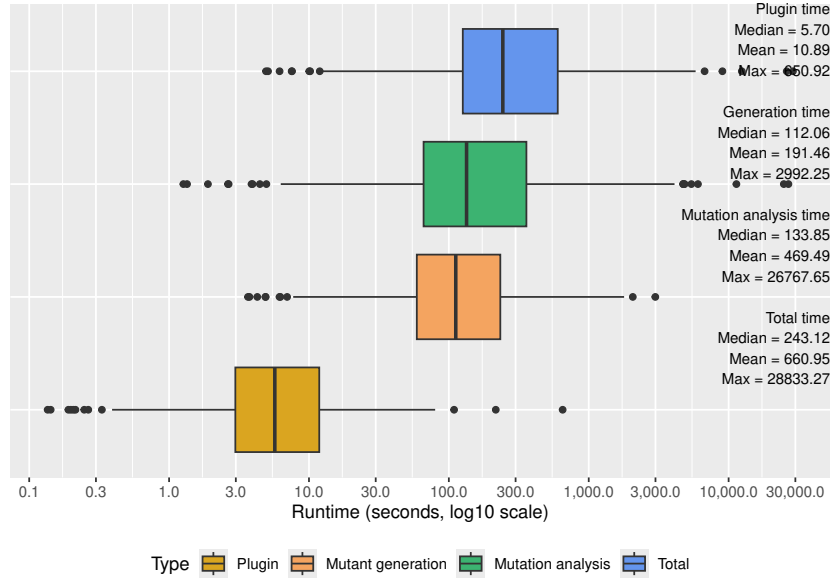


Figure 6.4: Plugin, mutant generation, mutation analysis, and total runtime to perform mutation testing of a Dafny program.

Table 6.2: Programs with mutation analysis and total runtime in the 10^4 order of magnitude.

Program	Mutant generation time (seconds)	Mutation analysis time (seconds)	Total runtime (seconds)
dafleet_tmp_tmpa2e4kb9v_0001-0050_0005-longest-palindromic-substring	1,918.84	26,144.95	28,063.79
Program-Verification-Dataset_tmp_tmpgbdrlnu_Dafny_lightening_verifier	1,183.84	24,328.91	25,512.76
dafny-rope_tmp_tmpl4v_njmy_Rope	1,014.83	11,082.69	12,097.52

executions, the total runtime is approximately equally divided between mutant generation and verification. In contrast, extreme runtimes seem primarily due to abnormal executions of Dafny’s verifier.

In half of the programs in our dataset, MutDafny takes between one and four minutes to generate all of the mutants for each. Even so, that time does not accurately reflect the runtime of our implementation. In reality, the plugin does not require minutes to execute its group of **AST** traversals. It most frequently scans and mutates the program in between three and 10 seconds, not exceeding this value in 75% of the cases and, according to our estimate, only accounting for around 5% of the total mutant generation runtime. The remaining 95% of the time consists of mutant resolution, which, despite being a regular part of Dafny’s execution, is crucial for the identification of invalid mutants. Overall, Dafny’s internal performance, including both the resolution and verification stages, increases the median value of the

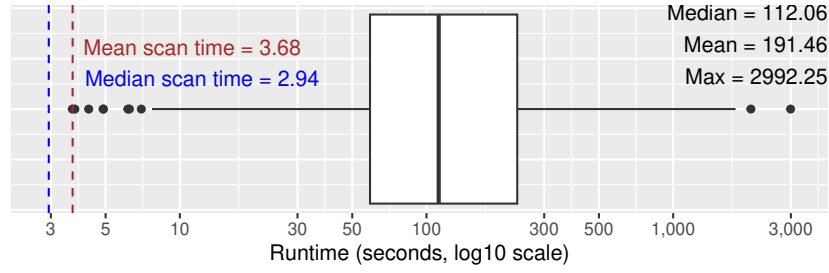


Figure 6.5: Comparison between the average scan runtime and the overall mutant generation runtime.

tool’s runtime by over 40 times. Considering Dafny’s internal capabilities are essential to the mutation testing pipeline, this constitutes a bottleneck to its application in Dafny programs.

Figure 6.5 depicts, on average, the portion of time corresponding to the scanning of the original program for mutation targets rather than actually mutating them. The plot shows that the scan time is almost negligible regarding the total mutant generation runtime. Because the scanner is only run once per program, while the mutator is run several times, once for each generated mutant, its runtime will already be a small portion of the total plugin runtime. Considering the plugin runtime itself only constitutes around 5% of the total mutant generation runtime, the scanner’s single execution becomes an insignificant fraction of the overall process.

RQ3: The analysis of the collected experimental results, on the one hand, revealed considerable runtime values associated with the application of mutation testing in Dafny programs. These run-times (2-10 minutes) may be acceptable for the analysis of individual files but raise scalability concerns for multi-file projects. On the other hand, we can affirm MutDafny’s efficiency to be affected mainly by Dafny’s own workflow. Furthermore, we observed that, while MutDafny presents a substantial mutant generation capability, some operators seem to contribute little or nothing at all to this, raising questions regarding their suitability in the context of Dafny and whether the cost of their inclusion overshadows their practical relevance.

6.6.2 Answer to RQ4, How do Dafny specifications perform at killing mutants?

Table 6.3 lists, for every mutation operator implemented in MutDafny, the total number of mutants generated with that operator, and how many of those were killed, survived, resulted in an invalid Dafny program, or timed out. The tool generates a total of 87,941 mutants, with only around 4% being invalid, and the number of timeouts during mutation analysis almost insignificant. While in Section 6.6.1 we analyzed the operators responsible for generating the highest and lowest number of mutants, now we will look further into their statuses after mutation analysis.

The fact that all of the mutants generated using the **CBR** operator remain alive is immediately apparent, so we manually reviewed each of them, aiming to understand this phenomenon. The five mutants happen to have all been generated for the same program, and the mutation directly changes the value

Table 6.3: Number of generated, killed, survived, invalid, and timeout mutants.

Operator	# Mutants	# Killed	# Survived	# Invalid	# Timeout
AMR	0	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
BBR	4,180	2,997 (71.70%)	1,051 (25.14%)	124 (2.97%)	8 (0.19%)
BOR	16,389	11,569 (70.59%)	3,562 (21.73%)	1,177 (7.18%)	81 (0.49%)
CBE	1,300	923 (71.00%)	362 (27.85%)	15 (1.15%)	0 (0.00%)
CBR	5	0 (0.00%)	5 (100.00%)	0 (0.00%)	0 (0.00%)
CIR	724	526 (72.65%)	159 (21.96%)	39 (5.39%)	0 (0.00%)
DCR	341	243 (71.26%)	94 (27.57%)	4 (1.17%)	0 (0.00%)
EVR	11,376	9,256 (81.36%)	1,652 (14.52%)	417 (3.67%)	51 (0.45%)
FAR	93	60 (64.52%)	26 (27.96%)	7 (7.53%)	0 (0.00%)
LBI	723	615 (85.06%)	106 (14.66%)	1 (0.14%)	1 (0.14%)
LSR	53	30 (56.60%)	23 (43.40%)	0 (0.00%)	0 (0.00%)
LVR	10,397	7,691 (73.97%)	2,684 (25.82%)	0 (0.00%)	22 (0.21%)
MAP	311	166 (53.38%)	108 (34.73%)	37 (11.90%)	0 (0.00%)
MCR	411	104 (25.30%)	301 (73.24%)	6 (1.46%)	0 (0.00%)
MMR	20	10 (50.00%)	10 (50.00%)	0 (0.00%)	0 (0.00%)
MNR	18	1 (5.56%)	15 (83.33%)	2 (11.11%)	0 (0.00%)
MRR	834	385 (46.16%)	379 (45.44%)	69 (8.27%)	1 (0.12%)
MVR	1,305	260 (19.92%)	1,041 (79.77%)	2 (0.15%)	2 (0.15%)
ODL	4,258	1,345 (31.59%)	1,712 (40.21%)	1,193 (28.02%)	8 (0.19%)
PRV	0	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
SAR	458	305 (66.59%)	148 (32.31%)	2 (0.44%)	3 (0.66%)
SDL	4,715	3,272 (69.40%)	1,122 (23.80%)	309 (6.55%)	12 (0.25%)
SLD	113	76 (67.26%)	36 (31.86%)	0 (0.00%)	1 (0.88%)
SWS	3,022	1,151 (38.09%)	1,630 (53.94%)	231 (7.64%)	10 (0.33%)
SWV	368	184 (50.00%)	131 (35.60%)	53 (14.40%)	0 (0.00%)
TAR	15	8 (53.33%)	2 (13.33%)	5 (33.33%)	0 (0.00%)
THD	0	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
THI	8	5 (62.50%)	1 (12.50%)	2 (25.00%)	0 (0.00%)
UOD	217	172 (79.26%)	45 (20.74%)	0 (0.00%)	0 (0.00%)
UOI	14,636	12,301 (84.05%)	2,123 (14.51%)	145 (0.99%)	67 (0.46%)
VDL	749	315 (42.06%)	267 (35.65%)	167 (22.30%)	0 (0.00%)
VER	10,902	8,679 (79.61%)	2,023 (18.56%)	93 (0.85%)	107 (0.98%)
<i>Total</i>	87,941	62,649 (71.24%)	20,818 (23.67%)	4,100 (4.66%)	374 (0.43%)

returned by the target method. Since this specific target method, despite declaring a set of preconditions, does not contain any postconditions, there is no way for the mutation to its return value to be detected. We can, therefore, confirm that the mutation operator functions as intended and that this outlier is merely due to its low usage. In fact, these five surviving mutants do indeed signal a weakness in the target method's specification. Besides **CBR**, we can also highlight the **MCR**, **MMR**, **MNR**, **MRR**, **MVR**, and **SWS** operators as producing a moderately high number of surviving mutants. To better understand

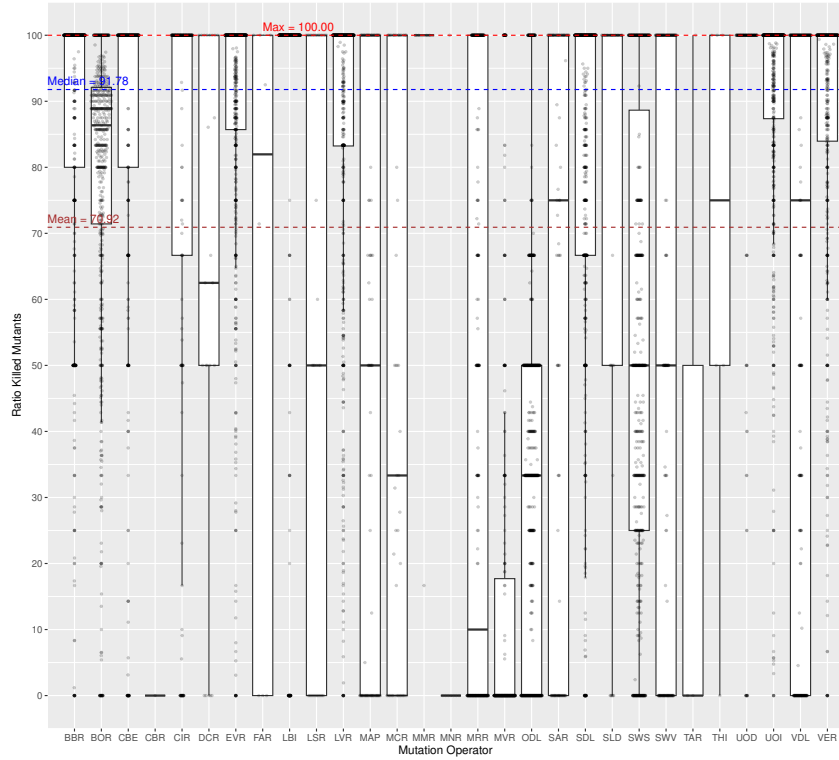


Figure 6.6: Ratio of killed mutants per mutation operator.

whether this percentage reflects actual hints at specification weaknesses or simply indicates that these operators are not particularly helpful to the application of mutation testing in Dafny, for example, if they generate many equivalent mutants, a deeper study of the alive mutants would be required.

Regarding invalid mutants, approximately 30% of these originate from the application of the **BOR** operator. Nevertheless, this is the operator that, overall, generates the highest number of mutants, with only 7% of them being invalid. Cases of invalid arithmetic replacement occur in chaining and `forall` expressions and are challenging to determine at mutation time. We accept these as a limitation of a first prototype and as room for future improvement. **ODL** and **VDL** come second as the operators generating the most invalid mutants. These two operators perform the deletion of every occurrence of a specific binary operator or variable, respectively, in the program, corresponding to the operators with the most complex implementation logic. This explains, to some degree, the high percentage of invalid mutants.

Figure 6.6 illustrates the distribution of the ratio of mutants generated by each operator that were killed per program. The figure shows that the operators **BBR**, **BOR**, **CBE**, **EVR**, **LVR**, **UOD**, **UOI**, and **VER** are the ones that consistently generate a higher percentage of killed mutants per program. To be more specific, over half of the specifications kill at least 70% of the mutants generated by these operators. On the other hand, the mutants generated by the **FAR**, **LSR**, **MAP**, **MCR**, **MRR**, **SAR**, **SWV**, and **VDL** operators appear to exhibit a very inconsistent kill ratio, with certain programs demonstrating a high mutant detection capability, while, in others, the mutations seem to go unnoticed by the specifications.

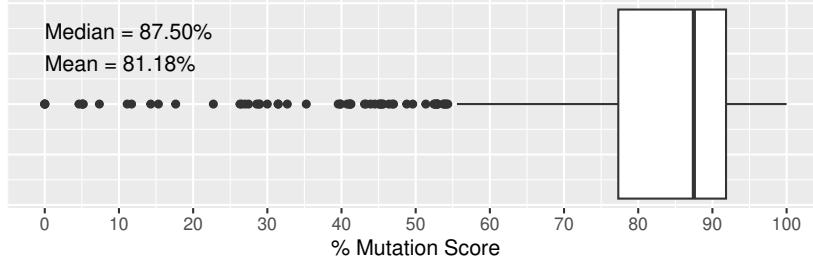


Figure 6.7: Mutation score distribution.

Table 6.4: Programs with a mutation score of 0%.

Program	Suitable for mutation	Mutants
dafny-aoc-2019_tmp_tmpj6suy_rv_parser_split	3 methods with 1 statement each	36
groupTheory_tmp_tmppmmxvu8h_tutorial2	3 statements	13
Program-Verification-Dataset_tmp_tmpgbdrlnu__Dafny_advanced examples_Refinement	2 statements	4
Program-Verification-Dataset_tmp_tmpgbdrlnu__Dafny_from dafny main repo_dafny0_ModulePrint	2 statements	5

Figure 6.7 illustrates the overall distribution of the mutation score per program. The mean mutation score value is 81.18%, and the median value is 87.5%, meaning that half of the programs kill at least 87.5% of the mutants generated by MutDafny.

In addition, 43 programs exhibit a mutation score of 100%, while four programs do not kill any mutants, manifesting a score of 0%. We inspected these four outliers more closely and, as detailed in Table 6.4, some of these programs already generate a very low number of mutants, and, apart from that, they all contain few statements where mutations can be applied. Small programs mean that sometimes there is not a lot of program behaviour to be constrained by a specification, even if a considerable number of mutants can be generated, as is the case with the program with 36 mutants. Even though all of the programs in DafnyBench provide specifications, we observed that some of the mutations applied to the four programs in question targeted particular methods or functions with no specification and that were also not used anywhere else in the program (e.g., `Main` methods). We also uncovered a group of mutants generated by changes to unused variables, constituting a group of equivalent mutants.

From the 20,818 surviving mutants, only 9,689 affect methods with postconditions. The remaining 11,129 can either be weak due to the lack of specification in the mutated component, hint at a weakness in a different component calling the mutated one, or be equivalent to the original program. Being unfeasible to manually analyze 9,689 mutants to distinguish whether they were not killed due to a weak specification or for being equivalent to the original program, we focused on those mutating methods with postconditions. This way, if a weakness is present in any of these, it most likely will affect the specification of the mutated method, simplifying the analysis process. We randomly selected 24 files,

and manually analyzed all their mutants in methods with postconditions, 284 in total. In these 24 files, we found five specification weaknesses. From the 284 analyzed mutants, 77 hinted at these weaknesses and were successfully killed upon strengthening the specifications. 157 turned out to be equivalent to the original program, and we could not draw meaningful conclusions from the remaining 50.

In `BinarySearchTree_tmp_tmp_bn2twp5_bst4copy`, the `insert`⁴ 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`⁵ 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 was made to the input tree, neither method's specification would be able to detect said incorrect behaviour. In `dafny-synthesis_task_id_126`⁶, the postcondition 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 postcondition would not be able to detect it. The purpose of `dafny-synthesis_task_id_161`⁷ 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 is 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. `dafny-synthesis_task_id_2`⁸ and `dafny-synthesis_task_id_249`⁹ are very similar to `dafny-synthesis_task_id_161`, computing the unique elements common to two input arrays, and the weakness being the same.

RQ4: The reported mutation score values show that Dafny specifications are indeed effective at detecting mutations, as 71.24% of the mutants do not pass verification with the original specification, and the programs present a median 87.5% mutation score. However, even one single mutant may be enough to detect a particular aspect of program behaviour not constrained by the specification, constituting a weakness. The discovery of five weak specifications in our experimental subjects validates the usefulness of MutDafny in aiding the identification of weaknesses. For this reason, any surviving mutants should always be manually inspected as a way of achieving a higher confidence level over the quality of the spec.

⁴https://github.com/sun-wendy/DafnyBench/blob/main/DafnyBench/dataset/ground_truth/BinarySearchTree_tmp_tmp_bn2twp5_bst4copy.dfy#L58, accessed 24 Jul 2025.

⁵https://github.com/sun-wendy/DafnyBench/blob/main/DafnyBench/dataset/ground_truth/BinarySearchTree_tmp_tmp_bn2twp5_bst4copy.dfy#L66, accessed 24 Jul 2025.

⁶https://github.com/sun-wendy/DafnyBench/blob/main/DafnyBench/dataset/ground_truth/dafny-synthesis_task_id_126.dfy#L1, accessed 24 Jul 2025.

⁷https://github.com/sun-wendy/DafnyBench/blob/main/DafnyBench/dataset/ground_truth/dafny-synthesis_task_id_161.dfy#L7, accessed 24 Jul 2025.

⁸https://github.com/sun-wendy/DafnyBench/blob/main/DafnyBench/dataset/ground_truth/dafny-synthesis_task_id_2.dfy#L7, accessed 24 Jul 2025.

⁹https://github.com/sun-wendy/DafnyBench/blob/main/DafnyBench/dataset/ground_truth/dafny-synthesis_task_id_249.dfy#L7, accessed July 2025.

6.7 Summary

The conducted empirical evaluation revealed key insights regarding the applicability of mutation testing to Dafny specifications. Its application to the 745 benchmark programs resulted in the generation of over 87,941 mutants, providing us with comprehensive results to address our two core research questions.

Chapter 7

Conclusion and Future work

7.1 Conclusion

This thesis presented a methodology for the detection of specification weaknesses through the mutation of Dafny programs, an in-depth study of mutation operators suitable for the language, and culminated in the introduction of MutDafny, a prototype mutation testing tool for Dafny.

The tool was evaluated on a dataset of 745 Dafny programs, showing that the methodology can be successfully applied, namely to real-world programs. The results demonstrate that Dafny specifications present a high mutant detection capability, though our partial analysis of alive mutants also revealed the presence of weaknesses. For this reason, we believe that MutDafny has the potential to deliver a meaningful contribution to the field of formal verification in improving the confidence of these systems, while also expanding the understanding of how specification weaknesses affect verification outcomes.

7.2 Future work

For the application of mutation testing in Dafny to become feasible, optimization is essential. There is little room to work on reducing the mutant generation or mutation analysis runtimes since these processes are dominated by Dafny’s own runtime. A more reliable strategy would be to reduce the number of mutants generated per program, entailing a reduction in the number of programs to resolve and verify as well. There are several approaches proposed in the mutation testing literature that could help us achieve this by avoiding the generation of irrelevant mutants and preserving those adding insight into the quality of the specification.

In traditional mutation testing, where the goal is to assess the quality of a test suite, a mutant in a section of the code that is not reached by the tests is impossible to kill. The concept of test coverage is used to avoid mutating these portions of the code [31]. Similarly, one type of irrelevant mutation in the context of Dafny is when the target is located in a portion of the code that is not exercised by the spec. The identification of the lines of code not covered by the specification, on the one hand, also serves as a

mechanism to signal a potential incompleteness of the spec, and on the other, improves performance by circumventing the generation of mutants that are already known to be survivors.

Mutants that are equivalent to the original program are also impossible to kill, and furthermore, they skew the mutation score by contributing with false negatives. Some studies [23, 55] have successfully minimized their occurrence in other programming languages. Additionally, reducing the number of mutants equivalent to each other and not only to the original program, if feasible, can also help reduce the overall number of mutants to generate and verify. Due to Dafny's incompleteness, programs that satisfy a specification may fail to verify while a semantically equivalent mutant may succeed. Consequently, while a single successful verification allows us to discard the remaining equivalent mutants, a single failure does not provide enough information to do so. New strategies should be studied to reduce the number of equivalent mutants that undergo verification while avoiding eliminating potentially verifiable ones. This introduces challenges uncommon in mutation testing for traditional programming languages.

References

- [1] Cezar-Constantin Andrici and Ștefan Ciobâcă. Verifying the dpll algorithm in dafny. *Electronic Proceedings in Theoretical Computer Science*, 303:3–15, September 2019. ISSN 2075-2180. doi: 10.4204/eptcs.303.1. URL <http://dx.doi.org/10.4204/EPTCS.303.1>. Cited on page 9.
- [2] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: a modular reusable verifier for object-oriented programs. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, FMCO’05, page 364–387, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3540367497. doi: 10.1007/11804192_17. URL https://doi.org/10.1007/11804192_17. Cited on pages 10 and 49.
- [3] Martin Brain and Elizabeth Polgreen. A pyramid of (formal) software verification. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 14934 LNCS:393 – 419, 2025. doi: 10.1007/978-3-031-71177-0_24. Cited by: 0; All Open Access, Hybrid Gold Open Access. Cited on page 87.
- [4] Arno Broeders, Ruud Hermans, Sylvia Stuurman, Lex Bijlsma, and Harrie Passier. Improving students’ code correctness and test completeness by informal specifications. *arXiv preprint arXiv:2309.02221*, 2023. Cited on page 88.
- [5] Tej Chajed, Joseph Tassarotti, Mark Theng, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying the DaisyNFS concurrent and crash-safe file system with sequential reasoning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 447–463, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL <https://www.usenix.org/conference/osdi22/presentation/chajed>. Cited on page 9.
- [6] YuTing Chen and Carlo A. Furia. Robustness testing of intermediate verifiers. In Shuvendu K. Lahiri and Chao Wang, editors, *Automated Technology for Verification and Analysis*, pages 91–108, Cham, 2018. Springer International Publishing. ISBN 978-3-030-01090-4. Cited on pages 7, 10, 11, and 88.
- [7] Zheng Cheng and Massimo Tisi. Incremental deductive verification for relational model transformations. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 379–389. IEEE, 2017. Cited on page 88.
- [8] Zheng Cheng, Rosemary Monahan, and James F Power. Formalised emftvm bytecode language for sound verification of model transformations. *Software & Systems Modeling*, 17:1197–1225, 2018. Cited on page 88.
- [9] Janice Chin and David Pearce. Finding bugs with specification-based testing is easy! *arXiv preprint arXiv:2103.00032*, 2021. Cited on page 88.

- [10] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. PIT: a practical mutation testing tool for Java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 449–452, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343909. doi: 10.1145/2931037.2948707. URL <https://doi.org/10.1145/2931037.2948707>. Cited on pages 13, 21, 23, 89, 90, 91, 92, 93, 94, and 95.
- [11] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78800-3. Cited on page 50.
- [12] Alex Denisov and Stanislav Pankevich. Mull it over: Mutation testing based on llvm. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 25–31, 2018. doi: 10.1109/ICSTW.2018.00024. Cited on pages 14, 89, 90, 93, 94, and 95.
- [13] Anna Derezińska and Konrad Hałas. Analysis of mutation operators for the python language. In Wojciech Zamojski, Jacek Mazurkiewicz, Jarosław Sugier, Tomasz Walkowiak, and Janusz Kacprzyk, editors, *Proceedings of the Ninth International Conference on Dependability and Complex Systems DepCoS-RELCOMEX. June 30 – July 4, 2014, Brunów, Poland*, pages 155–164, Cham, 2014. Springer International Publishing. ISBN 978-3-319-07013-1. Cited on pages 14, 31, 89, 90, 91, 92, 93, 94, 95, 96, and 97.
- [14] Alastair F. Donaldson, Dilan Sheth, Jean-Baptiste Tristan, and Alex Usher. Randomised testing of the compiler for a verification-aware programming language. In *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 407–418, 2024. doi: 10.1109/ICST60714.2024.00044. Cited on pages 7, 9, 10, 11, and 87.
- [15] Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K Lahiri. Formalizing natural language intent into program specifications via large language models. *arXiv preprint arXiv:2310.01831*, 2023. Cited on page 88.
- [16] Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K. Lahiri. Can large language models transform natural language intent into formal method postconditions? *Proc. ACM Softw. Eng.*, 1(FSE), July 2024. doi: 10.1145/3660791. URL <https://doi.org/10.1145/3660791>. Cited on pages 7 and 88.
- [17] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. An empirical study on the correctness of formally verified distributed systems. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys ’17, page 328–343, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349383. doi: 10.1145/3064176.3064183. URL <https://doi.org/10.1145/3064176.3064183>. Cited on pages 2 and 87.
- [18] Miroslav Gavrilov, Kyle Dewey, Alex Groce, Davina Zamanzadeh, and Ben Hardekopf. A practical, principled measure of fuzzer appeal: A preliminary study. page 510 – 517, 2020. doi: 10.1109/QRS51102.2020.00071. Cited by: 2. Cited on page 88.
- [19] Rohit Gheyi, Márcio Ribeiro, Beatriz Souza, Marcio Guimarães, Leo Fernandes, Marcelo d’Amorim, Vander Alves, Leopoldo Teixeira, and Baldoino Fonseca. Identifying method-level

- mutation subsumption relations using z3. *Information and Software Technology*, 132:106496, 2021. ISSN 0950-5849. doi: <https://doi.org/10.1016/j.infsof.2020.106496>. URL <https://www.sciencedirect.com/science/article/pii/S095058492030238X>. Cited on pages 7 and 11.
- [20] Samuel Jiménez Gil, Manuel I. Capel, and Gabriel Olea Olea. Automatic test cases generation from formal contracts. *Information and Software Technology*, 172, 2024. doi: 10.1016/j.infsof.2024.107467. Cited by: 0; All Open Access, Hybrid Gold Open Access. Cited on page 87.
- [21] Samuel Jiménez Gil, Manuel I. Capel, and Gabriel Olea Olea. Automatic test cases generation from formal contracts. *Information and Software Technology*, 172:107467, 2024. ISSN 0950-5849. doi: <https://doi.org/10.1016/j.infsof.2024.107467>. URL <https://www.sciencedirect.com/science/article/pii/S0950584924000727>. Cited on pages 6 and 7.
- [22] Eli Goldweber, Weixin Yu, Seyed Armin Vakil Ghahani, and Manos Kapritsos. IronSpec: Increasing the reliability of formal specifications. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 875–891, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3. URL <https://www.usenix.org/conference/osdi24/presentation/goldweber>. Cited on pages 7, 8, 11, and 87.
- [23] Mahdi Houshmand and Samad Paydar. Tce+: An extension of the tce method for detecting equivalent mutants in java programs. In Mehdi Dastani and Marjan Sirjani, editors, *Fundamentals of Software Engineering*, pages 164–179, Cham, 2017. Springer International Publishing. ISBN 978-3-319-68972-2. Cited on page 79.
- [24] Cătălin Hrițcu. *The Quest for Formally Secure Compartmentalizing Compilation*. PhD thesis, ENS Paris; PSL Research University, 2019. URL <https://theses.hal.science/tel-01995823/>. Cited on page 88.
- [25] Catalin Hritcu, Chantal Keller, Aseem Rastogi, Pierre-Yves Strub, and Nikhil Swamy. F: Smt-based verification for ml. URL <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=011b249ffa298088415alded2c4e8790a20ad>. Cited on page 88.
- [26] Ahmed Irfan, Sorawee Porncharoenwase, Zvonimir Rakamarić, Neha Rungta, and Emina Torlak. Testing dafny (experience paper). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2022*, page 556–567, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393799. doi: 10.1145/3533767.3534382. URL <https://doi.org/10.1145/3533767.3534382>. Cited on pages 6, 7, and 87.
- [27] Ru Ji and Meng Xu. Finding specification blind spots via fuzz testing. volume 2023-May, page 2708 – 2725, 2023. doi: 10.1109/SP46215.2023.10179438. Cited by: 0. Cited on page 87.
- [28] Ru Ji and Meng Xu. Finding specification blind spots via fuzz testing. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2708–2725, 2023. doi: 10.1109/SP46215.2023.10179438. Cited on page 7.
- [29] Yue Jia and Mark Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011. doi: 10.1109/TSE.2010.62. Cited on page 2.

- [30] René Just. The major mutation framework: efficient and scalable mutation analysis for Java. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014*, page 433–436, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450326452. doi: 10.1145/2610384.2628053. URL <https://doi.org/10.1145/2610384.2628053>. Cited on pages 13, 89, 90, 91, 93, and 95.
- [31] René Just, Franz Schweiggert, and Gregory M. Kapfhammer. 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)*, pages 612–615, 2011. doi: 10.1109/ASE.2011.6100138. Cited on pages 13, 78, 89, 90, 91, and 93.
- [32] Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. From c to interaction trees: specifying, verifying, and testing a networked server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019*, page 234–248, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362221. doi: 10.1145/3293880.3294106. URL <https://doi.org/10.1145/3293880.3294106>. Cited on page 87.
- [33] Nikolai Kosmatov, Artjom Plaunov, Subash Shankar, and Julien Signoles. Combining analyses within frama-c. In *Guide to Software Verification with Frama-C: Core Components, Usages, and Applications*, pages 423–455. Springer, 2024. Cited on page 88.
- [34] Shuvendu K. Lahiri. Ai-assisted user intent formalization for programs: Problem and applications (invited talk). In *Proceedings of the 1st ACM International Conference on AI-Powered Software, AIware 2024*, page 179, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706851. doi: 10.1145/3664646.3676274. URL <https://doi.org/10.1145/3664646.3676274>. Cited on pages 7 and 87.
- [35] Shuvendu K. Lahiri. Evaluating llm-driven user-intent formalization for verification-aware languages, 2024. URL <https://repositum.tuwien.at/handle/20.500.12708/200786>. Cited on pages 7, 10, 12, and 88.
- [36] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-17511-4. Cited on page 1.
- [37] Ai Liu and Shaoying Liu. Enhancing the capability of testing-based formal verification by handling operations in software packages. *IEEE Transactions on Software Engineering*, 49(1):304 – 324, 2023. doi: 10.1109/TSE.2022.3150333. Cited by: 5. Cited on page 87.
- [38] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 21558–21572. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/43e9d647ccd3e4b7b5baab53f0368686-Paper-Conference.pdf. Cited on page 7.

- [39] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. volume 36, 2023. Cited by: 37. Cited on page 87.
- [40] Chloe Loughridge, Qinyi Sun, Seth Ahrenbach, Federico Cassano, Chuyue Sun, Ying Sheng, Anish Mudide, Md Rakib Hossain Misu, Nada Amin, and Max Tegmark. DafnyBench: A Benchmark for Formal Software Verification, 2024. URL <https://arxiv.org/abs/2406.08467>. Cited on pages x, 4, 39, 59, 60, and 61.
- [41] Yu-seung Ma and Jeff Offutt. Description of class mutation mutation operators for java. 2014. URL <https://cs.gmu.edu/~offutt/mujava/mutopsClass.pdf>. Cited on pages 13, 30, 95, 96, 97, and 98.
- [42] Yu-seung Ma and Jeff Offutt. Description of mujava’s method-level mutation operators. Update 2016. URL <https://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf>. Cited on pages 11, 13, 15, 89, 90, 91, 92, 95, and 96.
- [43] Yu-Seung Ma, Jeff Offutt, and Yong Kwon. Mujava: An automated class mutation system. *Softw. Test., Verif. Reliab.*, 15:97–133, 06 2005. doi: 10.1002/stvr.308. Cited on pages 13, 30, 95, 96, 97, and 98.
- [44] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. Mujava: a mutation system for java. In *Proceedings of the 28th International Conference on Software Engineering, ICSE ’06*, page 827–830, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933751. doi: 10.1145/1134285.1134425. URL <https://doi.org/10.1145/1134285.1134425>. Cited on page 13.
- [45] Enrique Martin-Martin, Manuel Montenegro, Adrián Riesco, Juan Rodríguez-Hortalá, and Rubén Rubio. Verification of the ros navfn planner using executable specification languages. *Journal of Logical and Algebraic Methods in Programming*, 132, 2023. doi: 10.1016/j.jlamp.2023.100860. Cited by: 3; All Open Access, Hybrid Gold Open Access. Cited on page 87.
- [46] Hugo Rafael Fecha Martins. Automated program repair of dafny programs. 2022. Cited on page 88.
- [47] Bertrand Meyer. *Design by contract*. Prentice Hall Upper Saddle River, 2002. Cited on page 1.
- [48] Martin Mirchev, Andreea Costea, Abhishek Kr Singh, and Abhik Roychoudhury. Assured automatic programming via large language models. *arXiv preprint arXiv:2410.18494*, 2024. Cited on page 88.
- [49] Md Rakib Hossain Misu, Cristina V. Lopes, Iris Ma, and James Noble. Towards AI-Assisted Synthesis of Verified Dafny Methods, 2024. URL <https://arxiv.org/abs/2402.00247>. Cited on page 39.
- [50] Iman Saleh Moustafa. Formal specification and verification of data-centric web services. 2012. URL <https://vtechworks.lib.vt.edu/items/9f883197-038d-4dd5-97ba-b7f7dcd257>. Cited on page 88.
- [51] Tim Nelson, Elijah Rivera, Sam Soucie, Thomas Del Vecchio, John Wrenn, and Shriram Krishnamurthi. Automated, targeted testing of property-based testing predicates. *Art, Science, and Engineering of Programming*, 6(2), 2022. doi: 10.22152/programming-journal.org/2022/6/10. Cited by: 2; All Open Access, Bronze Open Access, Green Open Access. Cited on page 87.

- [52] Safwan Omari, Kshitiz Basnet, and Mohammad Wardat. Investigating large language models capabilities for automatic code repair in python. *Cluster Computing*, 27(8):10717 – 10731, 2024. doi: 10.1007/s10586-024-04490-8. Cited by: 0. Cited on page 87.
- [53] Jonathan S Ostroff. Validating software via abstract state specifications. *York University, Tech. Rep. EECS-2017-02*, 2017. URL <https://www.eecs.yorku.ca/research/techreports/2017/EECS-2017-02.pdf>. Cited on page 88.
- [54] Pavel Panchekha and Emina Torlak. Automated reasoning for web page layout. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 181–194, 2016. Cited on page 88.
- [55] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. 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*, volume 1, pages 936–946, 2015. doi: 10.1109/ICSE.2015.103. Cited on page 79.
- [56] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Ranking programming languages by energy efficiency. *Science of Computer Programming*, 205:102609, 2021. ISSN 0167-6423. doi: <https://doi.org/10.1016/j.scico.2021.102609>. URL <https://www.sciencedirect.com/science/article/pii/S0167642321000022>. Cited on page 63.
- [57] Guillaume Petiot, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. Stady: Deep integration of static and dynamic analysis in frama-c. 2014. URL <https://inria.hal.science/hal-00992159/>. Cited on page 88.
- [58] Neha Rungta. A billion smt queries a day. In *International Conference on Computer Aided Verification*, pages 3–18. Springer, 2022. Cited on page 88.
- [59] Mooly Sagiv. Systematic testing. URL <https://www.cs.tau.ac.il/~msagiv/courses/sp/testing.pdf>. Cited on page 88.
- [60] Iman Saleh. *Formalizing Data-Centric Web Services*. Springer, 2015. Cited on page 7.
- [61] Iman Saleh. *Formalizing Data-Centric Web Services*. Springer, 2015. Cited on page 88.
- [62] Ana B. Sánchez, Pedro Delgado-Pérez, Inmaculada Medina-Bulo, and Sergio Segura. Mutation testing in the wild: findings from github. *Empirical Softw. Engg.*, 27(6), November 2022. ISSN 1382-3256. doi: 10.1007/s10664-022-10177-8. URL <https://doi.org/10.1007/s10664-022-10177-8>. Cited on pages 13 and 14.
- [63] Nicholas Smallbone. *Lightweight verification of functional programs*. Chalmers Tekniska Hogskola (Sweden), 2013. Cited on page 88.
- [64] Rebecca Smith. *Computer Science Education at Scale: Providing Personalized and Interactive Learning Experiences within Large Introductory Courses*. PhD thesis, Rice University, 2019. Cited on page 88.
- [65] Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. Clover: Closed-Loop Verifiable Code Generation, 2024. URL <https://arxiv.org/abs/2310.17807>. Cited on page 39.

- [66] Alex Usher, Alastair Donaldson, and Cristian Cadar. fuzz-d: Random Program Generation for Testing Dafny, 2023. URL <https://www.imperial.ac.uk/media/imperial-college/faculty-of-engineering/computing/public/2223-ug-projects/fuzz-d-Random-Program-Generation-for-Testing-Dafny.pdf>. Cited on pages 7, 9, 10, 11, and 88.
- [67] C. Wohlin, P. Runeson, M. Hst, M. Ohlsson, B. Regnell, and A. Wessln. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012. ISBN 3642290434. Cited on page 65.
- [68] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, EASE '14, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450324762. doi: 10.1145/2601248.2601268. URL <https://doi.org/10.1145/2601248.2601268>. Cited on page 7.
- [69] M.R. Woodward. Mutation testing—its origin and evolution. *Information and Software Technology*, 35(3):163–169, 1993. ISSN 0950-5849. doi: [https://doi.org/10.1016/0950-5849\(93\)90053-6](https://doi.org/10.1016/0950-5849(93)90053-6). URL <https://www.sciencedirect.com/science/article/pii/0950584993900536>. Cited on page 2.
- [70] Yi Zhou, Jay Bosamiya, Yoshiki Takashima, Jessica Li, Marijn Heule, and Bryan Parno. Mariposa: Measuring smt instability in automated program verification. In *2023 Formal Methods in Computer-Aided Design (FMCAD)*, pages 178–188, 2023. doi: 10.34727/2023/isbn.978-3-85448-060-0_26. Cited on page 51.

Appendix A

Complete list of records retrieved from literature database search

Table A.1: Records retrieved from literature database search (see Section 2.1). “S” stands for Scopus, “SD” for ScienceDirect, “GS” for Google Scholar, “IEEEEX” for IEEE Xplore, and “ACM” for ACM Digital Library.

Ref.	Title	Source(s)	Included
[14]	Randomised Testing of the Compiler for a Verification-Aware Programming Language	IEEEEX, GS	Yes
[26]	Testing Dafny (experience paper)	ACM, GS	No
[34]	AI-Assisted User Intent Formalization for Programs: Problem and Applications (Invited Talk)	ACM, GS	No
[17]	An Empirical Study on the Correctness of Formally Verified Distributed Systems	ACM, GS	No
[32]	From C to interaction trees: specifying, verifying, and testing a networked server	ACM	No
[3]	A Pyramid Of (Formal) Software Verification	S, GS	No
[52]	Investigating large language models capabilities for automatic code repair in Python	S, GS	No
[20]	Automatic test cases generation from formal contracts	S, SD, GS	No
[22]	IronSpec: Increasing the Reliability of Formal Specifications	S, GS	Yes
[45]	Verification of the ROS NavFn planner using executable specification languages	S, SD, GS	No
[39]	Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation	S, GS	No
[27]	Finding Specification Blind Spots via Fuzz Testing	S, GS	No
[37]	Enhancing the Capability of Testing-Based Formal Verification by Handling Operations in Software Packages	S, GS	No
[51]	Automated, Targeted Testing of Property-Based Testing Predicates	S, GS	No

(table continues)

Table A.1: Records retrieved from literature database search (continued).

Ref.	Title	Source(s)	Included
[18]	A Practical, Principled Measure of Fuzzer Appeal: A Preliminary Study	S, GS	No
[6]	Robustness Testing of Intermediate Verifiers	S, GS	Yes
[66]	fuzz-d: Random Program Generation for Testing Dafny	GS	Yes
[35]	Evaluating LLM-driven User-Intent Formalization for Verification-Aware Languages	GS	Yes
[59]	Systematic Testing	GS	No
[46]	Automated Program Repair of Dafny Programs	GS	No
[25]	F: SMT-Based Verification for ML	GS	No
[4]	Improving students' code correctness and test completeness by informal specifications	GS	No
[57]	StaDy: Deep Integration of Static and Dynamic Analysis in Frama-C	GS	No
[58]	A billion SMT queries a day	GS	No
[48]	Assured Automatic Programming via Large Language Models	GS	No
[9]	Finding Bugs with Specification-Based Testing is Easy!	GS	No
[15]	Formalizing natural language intent into program specifications via large language models	GS	No
[7]	Incremental deductive verification for relational model transformations	GS	No
[16]	Can Large Language Models Transform Natural Language Intent into Formal Method Postconditions?	GS	No
[61]	Formalizing Data-Centric Web Services	GS	No
[50]	Formal Specification and Verification of Data-Centric Web Services	GS	No
[64]	Computer Science Education at Scale: Providing Personalized and Interactive Learning Experiences within Large Introductory Courses	GS	No
[53]	Validating software via abstract state specifications	GS	No
[63]	Lightweight verification of functional programs	GS	No
[33]	Combining Analyses Within Frama-C	GS	No
[54]	Automated reasoning for web page layout	GS	No
[8]	Formalised EMFTVM bytecode language for sound verification of model transformations	GS	No
[24]	The Quest for Formally Secure Compartmentalizing Compilation	GS	No

Appendix B

Mutation operators proposed in the literature for non-Dafny programs

Table B.1: Compilation of mutation operators that have been proposed in a group of selected tools for non-Dafny programs (see Section 3.2). “Yes” means that the operator can be applied to Dafny, “No” means it cannot, “Partially” means that only a subset of the behaviour described by the operator can be applied to Dafny.

* Even though the operator can be applied to Dafny, it makes no sense to implement as it always generates programs that are either invalid or equivalent to the original one.

Operator	Description	Tool(s)	Dafny?
Arithmetic Operator Replacement (AOR)	Replacement of an arithmetic operator with another	Major [30, 31], MuJava [42], PIT [10], MutPy [13], Mutmut ¹	Yes
Arithmetic Operators	Replacement of an arithmetic operator with another	StrykerJS ² , Stryker.NET ³	Yes
Arithmetic Base	Replacement of an arithmetic operator with another	Go-Mutesting ⁴	Yes
Math	Replacement of an arithmetic operator with another	Mull [12]	Yes
Increments	Replacement between ++ and --	PIT [10]	No

(table continues)

¹https://github.com/boxed/mutmut/blob/main/mutmut/__init__.py, accessed 24 Jul 2025.

²<https://stryker-mutator.io/docs/mutation-testing-elements/supported-mutators/>, accessed 24 Jul 2025.

³<https://stryker-mutator.io/docs/stryker-net/mutations/>, accessed 24 Jul 2025.

⁴<https://github.com/avito-tech/go-mutesting?tab=readme-ov-file#list-of-mutators>, accessed 24 Jul 2025.

Table B.1: Mutation operators that have been proposed in the literature for non-Dafny programs (continued).

Operator	Description	Tool(s)	Dafny?
Relational Operator Replacement (ROR)	Replacement of a relational operator with another	Major [30, 31], MuJava [42], PIT [10], MutPy [13], Mutmut ¹	Yes
Relational Operator Replacement (ROR)	Replacement of boolean conditions with <code>true</code> or <code>false</code>	MuJava [42]	Yes
Conditionals Boundary	Replacement of a relational operator with its boundary counterpart (e.g., <code><</code> is replaced with <code><=</code> and vice versa)	PIT [10]	Yes
Expression Comparison	Replacement of a relational operator with its boundary counterpart (e.g., <code><</code> is replaced with <code><=</code> and vice versa)	Go-Mutesting ^{4 5}	Yes
Negate Conditionals	Replacement of a relational operator with its complement (e.g., <code><</code> is replaced with <code>>=</code> and vice versa)	PIT [10]	Yes
Negate Condition	Replacement of a relational operator with its complement (e.g., <code><</code> is replaced with <code>>=</code> and vice versa)	Mull [12]	Yes
Conditional Negated	Replacement of a relational operator with its complement (e.g., <code><</code> is replaced with <code>>=</code> and vice versa)	Go-Mutesting ⁴	Yes
Equality Operators	Replacement of each relational operator with its boundary counterpart or its complement	StrykerJS ² , Stryker.NET ³	Yes
Conditional Operator Replacement (COR)	Replacement of a conditional operator with another	Major [30], MuJava [42]	Yes
Logical Connector Replacement (LCR)	Replacement of a conditional operator with another	MutPy [13]	Yes
Logical Operators	Replacement of a conditional operator with another	StrykerJS ² , Stryker.NET ³	Yes
And/Or Test Mutation	Replacement between <code>and</code> and <code>or</code>	Mutmut ¹	No
And-Or Replacement	Replacement between <code>&&</code> and <code> </code>	Mull [12]	Yes
Conditional Operator Replacement (COR)	Replacement of boolean conditions with <code>true</code> or <code>false</code>	Major [30], PIT [10]	Yes
Conditional Expression	Replacement of boolean conditions with <code>true</code> or <code>false</code>	StrykerJS ²	Yes

(table continues)

⁵<https://github.com/zimmski/go-mutesting?tab=readme-ov-file#list-of-mutators>, accessed 24 Jul 2025.

Table B.1: Mutation operators that have been proposed in the literature for non-Dafny programs (continued).

Operator	Description	Tool(s)	Dafny?
Boolean Keyword Replacement	Replacement of boolean conditions with <code>true</code> or <code>false</code>	Mutmut ¹	Yes
Conditional Operators	Replacement of boolean conditions with <code>true</code> or <code>false</code> in the ternary operator	Stryker.NET ³	Yes
Expression Remove	Replacement of boolean expressions that contain <code>&&</code> and <code> </code> with <code>true</code> or <code>false</code>	Go-Mutesting ^{4 5}	Yes
Logical Operator Replacement (LOR)	Replacement of a logical operator with another	Major [30, 31], MuJava [42], PIT [10], MutPy [13], Mutmut ¹	Yes
Shift Operator Replacement (SOR)	Replacement of a shift operator with another	Major [30, 31], MuJava [42], Mutmut ¹	Yes
Math	Replacement of each arithmetic, logical and shift operators with a specific one from the same group	PIT [10]	Yes
Bitwise Operators	Replacement of each shift and logical operator with a specific one from the same group	Stryker.NET ³	Yes
Arithmetic Bitwise	Replacement of each shift and logical operator with a specific one from the same group	Go-Mutesting ⁴	Yes
Assignment Operator Replacement (ASR)	Replacement of an assignment operator with another	MuJava [42], MutPy [13], Mutmut ¹	No
Assignment Statements	Replacement of an assignment operator with another	Stryker.NET ³	No
Assignment Expression	Replacement of an assignment operator with another	StrykerJS ²	No
Arithmetic Assign Invert	Replacement of an assignment operator with another	Go-Mutesting ⁴	No
Arithmetic Assignment	Replacement of an assignment operator with <code>=</code>	Go-Mutesting ⁴	No
Operator Replacement Unary (ORU)	Replacement of a unary operator with another	Major [30, 31]	No

(table continues)

Table B.1: Mutation operators that have been proposed in the literature for non-Dafny programs (continued).

Operator	Description	Tool(s)	Dafny?
Unary Operators	Replacement of a unary operator with another	StrykerJS ² , Stryker.NET ³	No
Operator Replacement (ORU)	Unary Replacement of a short-cut arithmetic operator with other unary arithmetic operators	MuJava [42]	No
Update Operators	Replacement of a short-cut arithmetic operator with another	StrykerJS ² , Stryker.NET ³	No
Arithmetic Operator Insertion (AOI)	Insertion of short-cut and unary arithmetic operators	MuJava [42]	Partially
Unary Operator Insertion (UOI)	Insertion of short-cut and unary arithmetic operators	PIT [10]	Partially
Negation (ABS)	Replacement of a numerical variable with its negation	PIT [10]	Yes
Arithmetic Operator Deletion (AOD)	Deletion of short-cut and unary arithmetic operators	MuJava [42], PIT [10], MutPy [13]	Partially
Remove Increments	Deletion of variable increments	PIT [10]	No
Invert Negatives	Replacement of a negative variable with its positive	PIT [10]	Yes
Conditional Operator Insertion (COI)	Insertion of unary conditional operators	MuJava [42], MutPy [13]	Yes
Not Keyword Insertion	Insertion of the keyword <code>not</code> to negate specific boolean expressions	Mutmut ¹	No
Boolean Literals	Replacement of a boolean literal with its complement and negation of boolean expressions	StrykerJS ² , Stryker.NET ³	Yes
Conditional Operator Deletion (COD)	Deletion of unary conditional operators	MuJava [42], MutPy [13]	Yes
Not Keyword Deletion	Deletion of the keyword <code>not</code>	Mutmut ¹	No
Logical Operator Insertion (LOI)	Insertion of unary logical operators	MuJava [42]	Yes
Logical Operator Deletion (LOD)	Deletion of unary logical operators	MuJava [42], MutPy [13], Mutmut ¹	Yes
Bitwise Operator (OBBN)	Replacement of a bitwise operator with another and deletion of the operator	PIT [10]	Yes
Null-coalescing Operators	Deletion of the coalescing operator or swapping of the operands	Stryker.NET ³	No
Optional Chaining	Deletion of the optional chaining operator	StrykerJS ²	No

(table continues)

Table B.1: Mutation operators that have been proposed in the literature for non-Dafny programs (continued).

Operator	Description	Tool(s)	Dafny?
Literal Value Replacement (LVR)	Replacement of a literal value with another	Major [30, 31]	Yes
Constant Replacement (CRP)	Replacement of a constant with another	MutPy [13]	Yes
Inline Constant	Replacement of an inline constant with another or by its increment	PIT [10]	Yes
Scalar Value Replacement	Replacement of a numerical value with another	Mull [12]	Yes
Numbers Incrementer	Increments a numerical constant	Go-Mutesting ⁴	Yes
Number Mutation	Increments a numerical constant	Mutmut ¹	Yes
Numbers Decrementer	Decrements a numerical constant	Go-Mutesting ⁴	Yes
String Literals	Replacement of a non-empty string with an empty one and of an empty string with a default value	StrykerJS ² , Stryker.NET ³	Yes
String Mutation	Insertion of "XX" after the first character and before the last character of a string	Mutmut ¹	Yes
Regular Expressions	Application of a slight variation to a regular expression	StrykerJS ² , Stryker.NET ³	No
None Keyword Replacement	Replacement of the <code>None</code> keyword with an empty string	Mutmut ¹	No
Expression Value Replacement (EVR)	Replacement of an expression with a literal value	Major ⁶	Yes
Expression Mutation	Replacement of an expression with <code>None</code>	Mutmut ¹	No
Subscript Mutation	Replacement of the index or slicing expression of a subscript operation with <code>None</code>	Mutmut ¹	No
Exception Swallowing (EXS)	Insertion of an empty <code>catch</code> block	MutPy [13]	No
Return Values	Replacement of a return value with a literal	PIT [10]	Yes
Empty Returns	Replacement of a return value with an "empty" default value	PIT [10]	Yes
True Returns	Replacement of primitive and boolean return values with <code>true</code>	PIT [10]	Yes
False Returns	Replacement of primitive and boolean return values with <code>false</code>	PIT [10]	Yes
Null Returns	Replacement of a return value with <code>null</code>	PIT [10]	Yes

(table continues)

⁶<https://mutation-testing.org/docs.html>, accessed 24 Jul 2025.

Table B.1: Mutation operators that have been proposed in the literature for non-Dafny programs (continued).

Operator	Description	Tool(s)	Dafny?
Primitive Returns	Replacement of primitive return values with 0	PIT [10]	Yes
Lambda Mutation	Replacement of the return expression of a lambda with 0 or None	Mutmut ¹	Partially
Loop Break	Replacement of a loop's <code>break</code> statement with <code>continue</code> and vice versa	Go-Mutesting ⁴	Yes
Break Continue Replacement	Replacement of a loop's <code>break</code> statement with <code>continue</code> and vice versa	MutPy [13]	Yes
Continue Keyword Replacement	Replacement of a loop's <code>continue</code> statement with <code>break</code>	Mutmut ¹	Yes
Break Keyword Replacement	Replacement of a loop's <code>break</code> statement with <code>return</code>	Mutmut ¹	Yes
Loop Condition	Replacement of a <code>while</code> loop's condition with <code>1 < 1</code>	Go-Mutesting ⁴	Yes
Loop Range Break	Insertion of a <code>break</code> at the beginning of a range loop	Go-Mutesting ⁴	Yes
Constructor Call	Replacement of a constructor call with <code>null</code>	PIT [10]	Yes
Array Declaration	Replacement of an array initializer with an empty one	StrykerJS ²	Yes
Collection Expressions	Replacement of a non-empty array with an empty one and of an empty array with the language default	Stryker.NET ³	Partially
Initialization	Replacement of an array, list, collection, dictionary or class initializer with an empty one	Stryker.NET ³	Partially
Non-Void Method Call	Deletion of call to non-void method and replacement of its return value with the language default	PIT [10]	Yes
Replace Call	Deletion of call to a function that returns a numerical value and replacement of its return value with the value 42	Mull [12]	Yes
Argument Propagation (Experimental)	Replacement of a method call with one of its arguments of matching type	PIT ⁷	Yes
Argument Mutation	Insertion of "XX" at the end of the name of an argument	Mutmut ¹	No
Checked Statements	Replacement of a call to <code>checked</code> with its argument	Stryker.NET ³	No

(table continues)

Table B.1: Mutation operators that have been proposed in the literature for non-Dafny programs (continued).

Operator	Description	Tool(s)	Dafny?
Switch (Experimental)	Replacement of switch statement labels with default one and of default label with one provided by the user	PIT ⁷	Yes
Method Expression	Replacement of each of the language's methods with a specific one	StrykerJS ²	No
Big Integer (Experimental)	Replacement of a big integer method with another	PIT ⁷	No
String Methods	Replacement of each of the string class methods with a specific one or a default string value	Stryker.NET ³	No
Math Methods	Replacement of each of the math class methods with a specific one	Stryker.NET ³	No
Linq Methods	Replacement of each LINQ method with a specific one	Stryker.NET ³	No
Deepcopy Keyword Replacement	Replacement of the <code>deepcopy</code> keyword with <code>copy</code>	Mutmut ¹	No
Statement Deletion (SDL)	Deletion of a statement (e.g., <code>return</code> , <code>break</code> , assignment, method call, etc.)	Major [30], MuJava [42]	Yes
Statement Remove	Deletion of assignment, increment, decrement and expression statements	Go-Mutesting ^{4 5}	Partially
Removal	Deletion of a statement or of an entire function block	Stryker.NET ³	Yes
Block Statement	Deletion of an entire function block	StrykerJS ²	Yes
Branch Case	Deletion of an entire <code>case</code> block	Go-Mutesting ^{4 5}	Yes
Branch If	Deletion of entire <code>if</code> and <code>else if</code> branches	Go-Mutesting ^{4 5}	Yes
Branch Else	Deletion of entire <code>else</code> branches	Go-Mutesting ^{4 5}	Yes
Exception Handler Deletion (EHD)	Deletion of a <code>catch</code> block	MutPy [13]	No
Object Literal	Removal of an object's properties	StrykerJS ²	No
Member Variable (Experimental)	Deletion of assignments to member variables, as a result members will be initialized with language default	PIT ⁷	Yes
Java-Supported Default Constructor Creation (JDC)	Deletion of an implemented default constructor so that it is automatically generated	MuJava [41, 43]	Yes
Void Method Call	Deletion of call to void method	PIT [10]	Yes
Remove Void Function	Deletion of call to void function	Mull [12]	Yes

(table continues)

⁷<https://pitest.org/quickstart/mutators/>, accessed 24 Jul 2025.

Table B.1: Mutation operators that have been proposed in the literature for non-Dafny programs (continued).

Operator	Description	Tool(s)	Dafny?
Naked Receiver (Experimental)	Replacement of a method call with a naked receiver, i.e., deletion of the method call's receiver	PIT ⁷	Yes
Decorator Deletion (DDL)	Deletion of call to decorator of a function or method	MutPy [13], Mutmut ¹	No
Variable Deletion (VDL)	Deletion of all occurrences of a variable	MuJava [42]	Yes
Argument List Mutation	Deletion of one of the elements from a list of arguments	Mutmut ¹	No
Constant Deletion (CDL)	Deletion of all occurrences of a constant	MuJava [42]	Yes
Slice Index Remove (SIR)	Deletion of one of the arguments of a slice definition, i.e., in <code>collection[start:end:step]</code> , deletion of <code>start</code> , <code>end</code> or <code>step</code>	MutPy [13]	Partially
Operator Deletion (ODL)	Deletion of all occurrences of an operator	MuJava [42]	Yes
Hiding Variable Deletion (IHD)	Deletion of a hiding variable	MuJava [41, 43], MutPy [13]	No
Hiding Variable Insertion (IHI)	Insertion of a hiding variable	MuJava [41, 43]	No
Overriding Method Deletion (IOD)	Deletion of an overriding method from a subclass so that the parent's implementation is used	MuJava [43], MutPy [13]	No
Overriding Method Calling Position Change (IOP)	When the subclass's overriding method calls the parent's implementation, this call is moved either to the first or last line of the method	MuJava [41, 43], MutPy [13]	No
Overriding Method Rename (IOR)	Changes the name of the parent's method that is overridden by the child so that the child's implementation does not affect the parent's	MuJava [41, 43]	No
Super Keyword Insertion (ISI)	Inserts the keyword <code>super</code> to the reference to a subclass's field or method that also exists in the parent	MuJava [41, 43]	No
Super Calling Insertion (SCI)	Inserts the keyword <code>super</code> to the reference to a subclass's field or method that also exists in the parent	MutPy [13]	No
Super Keyword Deletion (ISD)	Deletes the keyword <code>super</code> from the reference to a subclass's field or method that also exists in the parent	MuJava [41, 43]	No

(table continues)

Table B.1: Mutation operators that have been proposed in the literature for non-Dafny programs (continued).

Operator	Description	Tool(s)	Dafny?
Super Calling Deletion (SCD)	Deletes the keyword <code>super</code> from the reference to a subclass's field or method that also exists in the parent	MutPy [13]	No
Explicit Call of a Parent's Constructor Deletion (IPC)	Deletes the <code>super</code> constructor calls from a subclass's constructor	MuJava [41, 43]	No
New Method Call with Child Class Type (PNC)	Replacement of parent constructor call with child constructor call and vice versa	MuJava [41, 43]	No
Instance Variable Declaration with Parent Class Type (PMD)	Replacement of the declared type of an object with the parent	MuJava [41, 43]	Yes*
Parameter Variable Declaration with Child Class Type (PPD)	Replacement of the declared type of a parameter with the parent	MuJava [41, 43]	Yes*
Type Cast Operator Insertion (PCI)	Casting of the type of an object reference to its parent or child	MuJava [41, 43]	Yes*
Cast Type Change (PCC)	In a type casting, replacement of the type that a variable is to be cast into to its parent or child	MuJava [41, 43]	Yes*
Type Cast Operator Deletion (PCD)	Deletion of a type casting of an object to its parent or child	MuJava [41, 43]	Yes*
Reference Assignment with Other comparable Type (PRV)	Replacement of a child reference assignment to a parent with a child reference of a different type	MuJava [41, 43]	Yes
Overloading Method Contents Change (OMR)	Replacement of the body of an overloaded method with the body of another method with the same name	MuJava [41, 43]	No
Overloading Method Deletion (OMD)	Deletion of an overloaded method	MuJava [41, 43]	No
Argument Order Change (OAO)	Changes the order of the arguments in an invocation to an overloaded method	MuJava [41, 43]	No
Argument Number Change (OAN)	Changes the number of arguments in an invocation to an overloaded method	MuJava [41, 43]	No
This Keyword Insertion (JTI)	Insertion of the <code>this</code> keyword in front of the use of a parameter that has the same name as a field	MuJava [41, 43]	Yes
This Keyword Deletion (JTD)	Deletion of the <code>this</code> keyword in front of the use of a field that has the same name as a parameter	MuJava [41, 43]	Yes
Static Modifier Insertion (JSI)	Insertion of the <code>static</code> keyword in front of a class field	MuJava [41, 43]	Yes*

(table continues)

Table B.1: Mutation operators that have been proposed in the literature for non-Dafny programs (continued).

Operator	Description	Tool(s)	Dafny?
Static Modifier Deletion (JSD)	Deletion of the <code>static</code> keyword in front of a class field	MuJava [41, 43]	Yes*
Member Variable Initialization Deletion (JID)	Deletion of a field initialization upon declaration so that a default value is used	MuJava [41, 43]	Yes
Reference Assignment and Content Assignment Replacement (EOA)	Replacement of an assignment of a pointer reference with a clone of the object	MuJava [41, 43]	No
Reference Comparison and Content Comparison Replacement (EOC)	Replacement between reference comparison and comparison using the <code>equals</code> method	MuJava [41, 43]	No
Accessor Method Change (EAM)	Replacement between accessor (<code>get</code>) methods with the same signatures	MuJava [41, 43]	Yes
Modifier Method Change (EMM)	Replacement between modifier (<code>set</code>) methods with the same signatures	MuJava [41, 43]	Yes

Appendix C

Mutation operators synthesized from bugfixes to Dafny programs

Table C.1: Synthesized mutation operators and commit evidence of their relevance (see Section 4.2).

Operator	Commits
Extended Conditional Replacement (COR)	• https://github.com/joaopascoalafariafeup/DafnyProjects/commit/01e0ae46668e7fe6cbd41dea4f1cdf4cc5602b37#diff-ccc646a86fd2fbb88304404cbd1cb7cdf789f1f5d2c52ebba9c607b8ad087e70L235-R235, accessed 24 Jul 2025. • https://github.com/Consensys/evm-dafny/commit/4946bec67d3678a4debb86aa3870c293fe531ccc#diff-5f537a3009d26ba61bcb2a3b39a3c4d0dbc32fff56fd177427b7e5a245dd6a3fL61-R61, accessed 24 Jul 2025. • https://github.com/secure-foundations/ironsync-osdi2023/commit/3960302a9311172864ff2151b3def6b5cc6a6af8#diff-44466bc6573dcb76f489fad45d5e5e9b8c4a4d3efe63d87f7634a03b6f3e830dL234-R234, accessed 24 Jul 2025.
Variable Expression Replacement (VER)	• https://github.com/dafny-lang/Dafny-VMC/commit/27a66014fd3a9e65763465f3d595a5ad5e2ef7f0#diff-9ead0207f306a1ba28e42076fd59fe8bb6802fa2faa5288b6e43bc42d95f3b9cL28-R28, accessed 24 Jul 2025.

(table continues)

Table C.1: Synthesized mutation operators and commit evidence of their relevance (continued).

Operator	Commits
Field Access Replacement (FAR)	• https://github.com/secure-foundations/ironsync-osdi2023/commit/9420f923c31e13564ddf0f740ae27ac439f40936#diff-8b6fe65b7c3a6ba702032935fb409c82b5a20dac0efcd4517966f82c1b980ee2L196-R196, accessed 24 Jul 2025.
Method Call Replacement (MCR)	• https://github.com/aws/aws-encryption-sdk/commit/36d1fad02be3a36d382941dbc94812644ec5c47f#diff-d6402a5709ccca6c7b42036edacf38b136517d3b5d6206557dec0f8123522adL258-R258, accessed 24 Jul 2025. • https://github.com/Consensys/evm-dafny/commit/5326b088ed62ca67fa77e0e66302a1d79539a0b5#diff-d1508b6d3858ba7c1491a640e6e58f2e7a6c034e9c3fd0b8ac662dae366cefdL529-R529, accessed 24 Jul 2025.
Method-Variable Replacement (MVR)	• https://github.com/dafny-lang/Dafny-VMC/commit/ef3152c851ed989aae375dff558271b6c20d8e48#diff-66dcba1192e097932aec9462cdb8e65134fc25c93afcfa612894b2c94e64af5bL99-R99, accessed 24 Jul 2025. • https://github.com/dafny-lang/Dafny-VMC/commit/e84ba787e1e4ec8143bf17741ff7fd950ef23474#diff-66dcba1192e097932aec9462cdb8e65134fc25c93afcfa612894b2c94e64af5bR83, accessed 24 Jul 2025.
Datatype Constructor Replacement (DCR)	• https://github.com/secure-foundations/ironsync-osdi2023/commit/1d7eabcd10d3c9424d7cb915a0bc819987c772c6#diff-dd4cf8cbf6aaf5bb44a285c7bec0890c1092315f2b6a3548b0bfa4ad4f28c2a0L666-R666, accessed 24 Jul 2025.

(table continues)

Table C.1: Synthesized mutation operators and commit evidence of their relevance (continued).

Operator	Commits
Argument Swapping (SAR)	• https://github.com/aws/aws-encryption-sdk/commit/b409ea5a014bbdb8e0ff7c02783b452cc2c39476#diff-f945451ea4d5cc43ed01a97ffb67bfabf7fc17727818427f892b90a7ce2c04e7L55, accessed 24 Jul 2025. • https://github.com/secure-foundations/ironsync-osdi2023/commit/1cee55f84b05c675ad8438d8ffef9f19acd79767#diff-e0feb703d3fe24aa0b33e358eb7e14d8d69cca0584ddb42656f2d5f203f63ae3L182-R182, accessed 24 Jul 2025. • https://github.com/secure-foundations/ironsync-osdi2023/commit/2211e713e672c42b91359d9e555b05f8d442c205#diff-380737698e7297cadaa983d1d89d85be927e379e057910d77c21601e32d50a88L128-R128, accessed 24 Jul 2025.
Tuple Access Replacement (TAR)	• https://github.com/aws/aws-encryption-sdk/commit/4c9d5d6ea7ee3f5cb70d5f2b0fbb6bd886e95c08#diff-00af0f655962fd28a7232ec2053eadda1257bb4587d838f9957a75c069e80437L64-R64, accessed 24 Jul 2025.
Conditional Block Extraction (CBE)	• https://github.com/dafny-lang/Dafny-VMC/commit/1f590fdced9e259d1517773413e5a249cf56484#diff-b2a4d785c925588b8ae085aa7dbf14c524ff7dda0a3b3bafa8c6fd949fba990dL15-R19, accessed 24 Jul 2025. • https://github.com/benreynwar/SiLemma/commit/4da22d6abe75ccbc744134270e39a359a8ec6b4e#diff-0eff5d0d272121340e6e9d6bad6e03612a3057426e284b56cd185a1524d7937aL572-R577, accessed 24 Jul 2025.
Statement Swapping (SWS)	• https://github.com/dafny-lang/libraries/commit/5ecd461c8895b3ee76c3e41b7d8cbefea6619ddb#diff-3a16ac84e5bda345c586253ae5dbaa34730085d17a587393650195f5248eea80L46-R48, accessed 24 Jul 2025.

(table continues)

Table C.1: Synthesized mutation operators and commit evidence of their relevance (continued).

Operator	Commits
Variable Declaration Swapping (SWV)	• https://github.com/Consensys/evm-dafny/commit/58abe25070e2855c32466bf4f69cd8c1b166d9eb#diff-eec5c501c95e783482e4e062ad3c9c30e947c394e0388bc197ad2306a5840ea5L315-R316, accessed 24 Jul 2025.

Table C.2: Commit evidence of common fixes that do not translate into mutation operators (see Section 4.3).

Fix	Commits
Replacement between nat and int	• https://github.com/secure-foundations/ironsync-osdi2023/commit/d84e4d4bac0046a1879c2d7c5b0ad18ea718de1c, accessed 24 Jul 2025. • https://github.com/alexiadorneles/pucrs-metodos-formais-t1/commit/02402793a5999d44b69a5d4e1a7bc2c778960493, accessed 24 Jul 2025.
Replacement between var and const	• https://github.com/Zyfarok/fv2020-tms/commit/3083575c2106338623c43e2254923806a19ae7bb#diff-2bdcf6001ee40bc28ba7be323fd8b7a0991dfac32c421f85f63d10abe206a912L19-R31, accessed 24 Jul 2025. • https://github.com/Zyfarok/fv2020-tms/commit/8ba7880e4cb0cf062bd5c4d7b7e11963a6657d2f#diff-2bdcf6001ee40bc28ba7be323fd8b7a0991dfac32c421f85f63d10abe206a912L47-R47, accessed 24 Jul 2025.
Insertion and deletion of type parameter properties	• https://github.com/secure-foundations/ironsync-osdi2023/commit/e27b26cb56bf765898cd660f4351e4aeed194e1f#diff-7091ca089a282c06587ac4406db89d238ea977e5e47b93c290d769c1221f2be3L14-R14, accessed 24 Jul 2025. • https://github.com/secure-foundations/ironsync-osdi2023/commit/88e36dc542f65d0e9a62898aec72a8f64d4329eb#diff-a9864f26422d6ce53993dd9307556857a4b7c77687ed70f2ab091565b9933850L4-R37, accessed 24 Jul 2025.

(table continues)

Table C.2: Commit evidence of common fixes that do not translate into mutation operators (continued).

Fix	Commits
Dafny 3 to Dafny 4 migration	• https://github.com/namin/dafny-sandbox/commit/21e148e267228a52ef474c53194d2ab052471915#diff-4698b1118ee26951776954c2d28642a7dc6ad3abb38743d0034655ffdf302364L112-R112, accessed 24 Jul 2025.
Insertion and deletion of ghost keywords	• https://github.com/secure-foundations/ironsync-osdi2023/commit/0da7158cbfa434403b6d04e926945b85ef5a591e, accessed 24 Jul 2025. • https://github.com/secure-foundations/ironsync-osdi2023/commit/de87232fa504a4bb6e3d0079f38ac3d8f8874da7#diff-10aacdf18deb66491a2e4bb876eaeff2e7979b41d427563229817337b5302753L412-R420, accessed 24 Jul 2025.
Replacement between <code>s.Length</code> and <code> s </code>	• https://github.com/aws/aws-encryption-sdk/commit/3c0df96e01d8d6bfb88dbc50fe9179836e821bb9#diff-7fb14c1aed7f9bc20b0aebab879c64bd71c751d518182040c30d61fe58e371a1L29-R29, accessed 24 Jul 2025. • https://github.com/volodeyka/veri-sparse/commit/4be18940936258676c1f50b4f212bf95fe5a730f#diff-94003c6bac5d0e9766f14ac90e774a3191ee8b556fb6887a7a5baf9aef3f0820L62-R64, accessed 24 Jul 2025.