FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Automatic Specification Repair in Contract Programming

Alexandre Abreu

Master in Informatics and Computing Engineering

Supervisor: Nuno Macedo

Co-supervisor: Alexandra Mendes

September 8, 2023

Automatic Specification Repair in Contract Programming

Alexandre Abreu

Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

President: Prof. João Carlos Pascoal Faria
Referee: Prof. Hugo José Pereira Pacheco
Referee: Prof. Nuno Filipe Moreira Macedo

September 8, 2023

Abstract

Software verification using formal methods is a growing area. It is vital for code deemed critical by their owners, as it provides a rigorous indication that they follow the provided specification. These two aspects of code have different purposes, the specification is only used to explain what the implementation, which is the real program to be deployed, should do.

Programmers make mistakes, and that is why it is so important to be able to give feedback and help them find those. Plenty of research deals with code repair, focusing on fixing implementations assuming the specifications are correct, but what if it is the opposite? We need more research on providing alternative specifications that fix details missing during contract definition, considering that the implementation is proper.

In this thesis, we will provide a tool that receives Dafny files that fail to verify and suggests corrections to their specifications so that they can be compiled, exploring proposed specification repair techniques. This program can automatically offer changes to the contracts, dialogizing with Dafny, and be considered adequate by programmers. We also expect it to be able to fix relevant examples and use those to verify if the approach is effective and efficient.

A program like this can be valuable for programmers working on Dafny and make software verification easier to learn and implement, which could result in more significant adoption of those techniques in industry. When paired with implementation repair, these techniques make the computer aid the programmers in errors that may occur on every part of the code.

We have developed a prototype tool for analysing and repairing *Dafny* programs. And evaluated it, taking its limitations into consideration. Our evaluation demonstrates reasonable runtimes and effectiveness, represented by a good coverage of programs that can be fixed.

Keywords: Contract Repair, Contract Programming, Software Repair, Dafny, Formal Methods

Acknowledgments

I would like to, first and foremost, thank my supervisors, whose openness to discussions and help with deciding essential aspects of the project were fundamental to its execution. Professors Nuno Macedo and Alexandra Mendes were always thoughtful and helpful. They tried to assist and encourage me throughout the journey, even in moments of tension when I did not know how to proceed with the work and through all the concerns and problems we encountered. I am grateful for their aid and trust in me for this topic.

I want to thank the individuals and institutions that made this possible. FEUP supplied me with the infrastructure needed during all these five years. INESC TEC supported this work and, with my supervisors, provided me with opportunities to publish this work, a valuable experience.

Also, I am grateful for my family, which was a source of support and incentive during my entire academic path; Maria, for always being there for me and helping me to put my head in place and persist; and my friends, who created an enjoyable process out of these couple of years.

I must also note that this work was financed by national funds through FCT (Fundação para a Ciência e Tecnologia - Foundation for Science and Technology) under the project "SpecRep - Constraint-based Specification Repair"[1] (EXPL/CCI-COM/1637/2021).

[1] https://haslab.github.io/SpecRep/

"If debugging is the process of removing software bugs,
then programming must be the process of putting them in."

Edsger W. Dijkstra

Contents

Abbreviations and Symbols

AST	Abstract syntax tree
CPU	Central processing unit
DbC	Design by contract
ID	Identification
IDE	Integrated development environment
OOP	Object-oriented programming

Chapter 1

Introduction

1.1 Context

It is no secret that bugs can cause highly negative consequences [96]. They happen when the program implementation does not follow our expectations, leading to incorrect behaviour. That is one of the essential practical motivations for creating tools that help their prevention.

Contract programming, also known as *design by contract* or *DbC*, is a way of programming based on the idea of providing formal specifications (also known as contracts because they correspond to obligations and guarantees) to software components. Created with reliability in mind [63], it aids programmers in trying to achieve correct and robust software.

Contracts provide ways to tell what obligations and guarantees are provided by a function cal or method procedure — usually stateful. The relevant languages provide verifiers responsible for checking if the contracts are followed by the implementation, forcing the programmer to match them. It is possible to trust one of the two parts, assuming we trust the other. For example, if a person trusts the specifications and the verifier approves their code, they know they can trust the implementation. However, getting the specification right can be particularly tricky [52], even though it is often way more complicated to get the implementation right, as an abstract specification is usually much more succinct and readable.

Dafny is an imperative language based on contract programming, including elements desirable for a programmer coming from that background, like mutable state, references, and objects, which makes it possible to get verification without losing performance. It is also very accessible [56] because it combines key features of formal methods in a programming language, which makes it a vital tool that any programmer or student interested in the area can use.

1.2 Problem

Most studies focus on fixing implementations, assuming the specification is correct. We try to improve the research in the other direction by assuming the specification needs fixes and the implementation does not.

Even though specifications are generally shorter and more straightforward, they can be misdescribed. Also, the implementation and specification have different formats and goals, thus, needing different techniques for automatic repair. These inaccurate descriptions can sometimes happen, so researching techniques to fix contracts could be valuable. A good example is when programmers have a reference implementation of an algorithm and are still analysing their contract options. So, when the programmers trust the implementation to be correct, suggestions to change the contracts based on it can be useful.

There is a minimal amount of research in the topic of repairing contracts. Just one tool has been developed for contract repair assuming implementations to be correct [75]. Furthermore, as far as we are aware, no tool has been proposed for any kind of automatic software repair for *Dafny*.

1.3 Goals

This project aims to create a tool that helps programmers fix faulty *Dafny* programs when the specifications are wrong. The program should automatically provide relevant contract fixes that match the implementation, making the program verifiable, and it should be able to work with *Dafny* source code as input.

A straightforward example can be done with a method for subtraction on natural numbers:

```
method sub(a: nat, b: nat) returns (d: nat) {
  d := a - b;
}
```

This method looks very simple, but it cannot be verified by *Dafny*. The result can fall outside the `nat` range because it can be a negative number. There are two different ways of fixing this:

1. Changing the specification to require `a` to be greater than or equal to `b`:

```
method sub(a: nat, b: nat) returns (d: nat)
  requires a >= b
{
  d := a - b;
}

```

2. Changing the implementation, so the return type of `sub` is `int`:

```
method sub(a: nat, b: nat) returns (d: int) {
  d := a - b;
}

```

Both solutions are valid and depend on the programmer's preference and the method's usage. Solutions to fix this automatically can help people to write code quicker and without errors.

We can also note what parts of the program belong to the specification, in *Dafny*, they always follow one keyword, so we can easily split the code in two. But, the specification includes information necessary only to the verification process, being completely ignored by the compilation phase, which uses all information from the implementation part. Types can look like they belong to the specification, specially when the language only uses type annotations, like *Python*. But they influence the memory used by the variables and influence the source code, thus, they are part of the implementation. Even if the language has dynamic typing, it still has to define the low-level representation of the data.

1.4 Plan

For that purpose, we studied the state of the art, selected relevant approaches, adapted one to our context, and implemented a tool. With a prototype done, we evaluated it, verifying if it has reasonable performance, provides the desired results, and how relevant its repairs are.

Although we have yet to achieve the goal of making our proposed tool part of a plug-in, the tool is ready to be integrated into one. Also, the current program depends on user-provided tests.

The proposed program can significantly improve coverage of fix suggestions, by handling the specification fixes, it can be paired with a tool that suggests fixes to the implementation, increasing the domain of possible fix suggestions. The result is more programmer support for the construction of correct software.

The work presented here was also accepted for publication in the 5th International Workshop on Automated and Verifiable Software System Development [1].

1.5 Document Structure

Chapter 2 explains the current state-of-the-art of the topic and how the proposed approaches can or cannot be used when creating this program. Chapter 3 discusses the language used and its platform, including analysing its essential parts for implementing a plug-in. Chapters 4 and 5 describe the design of our solution and how it was implemented. Chapter 6 analyses and discusses the results of our experimental evaluation. Finally, Chapter 7 concludes the document and gives some final remarks about the possibilities of future work.

Chapter 2

Literature Review

In this chapter, we start by summarizing and explaining some important areas and concepts that are used in our work, and give an overview of their current state. In Section 2.1, we introduce contract programming, the basis of contract generation, *Dafny*, and one of the approaches for software verification. It goes on to explain the difference between static and dynamic analysis (Section 2.2), a main concept for all the subsequent sections, which give an overview of the current state-of-art of: Software repair (Section 2.3), focused on fixing implementations, which provide the first studies on repairing code, which is the basis of the contract repair research as well; Test generation (Section 2.4), a fundamental topic when dealing with some kinds of dynamic analysis of programs; Contract repair (Subsection 2.5.1), our main area of interest; Contract inference (Subsection 2.5.2), similar area capable of providing interesting techniques that could be applied to contract repair; and invariant detection (Section 2.6), an important area when dealing with some approaches to fix generation, including the one we follow. Where invariants are assertions that can happen at any program point, but contracts specify the interface of functions or methods.

For each one of those areas we explain their main concepts, give a few examples of some of their tools or algorithms, being more descriptive for the ones that are actually used in our work, and explain some challenges and properties of different approaches.

2.1 Contract Programming

Contract programming [65] [63] is a way of programming that prescribes the use of precise and formal specifications of the expected program behaviour, taking place as assertions, also known as contracts, which indicate obligations and guarantees that are provided by the software components. Centered around reliability, contract programming aids programmers in writing correct and robust software.

One of the initial concepts that led to contract programming was the notion of defensive programming, based on the realization that programs will have bugs, so we can try to develop proactive actions to try to fix them as soon as possible and with less effort [87].

With programmers focusing on software reliability, the idea of verifying code started to become increasingly more appealing. Developing techniques to verify if code was "correct" would make software development more efficient and reliable [70]. But they also imply effort in the creation of formal specifications of the program behaviour, so one could verify the concrete behaviour follows these specifications. This is a point of discussion between proponents and objectors of software verification, as it is hard to measure the trade-off from projects [93], which are combined with different preferences of programmers. Another important point is that checking whether an implementation satisfies its specification is a problem that has no fully general solution, so there will always be cases where a specific tool cannot verify a program, even if it is correct and follows its contract [88].

So, *contract programming* was born, and it was specially relevant for object-oriented programming [63], as: it incentivizes the reuse of code, which means an unexpected behaviour in a small function can have huge consequences more frequently, even though this could happen nonetheless; one of the main selling points of object orientation is the perceived improvement, by some people, of software quality, and reliability is a good property to be included in those quality parameters; its framework is a specially good environment for the discussion of reliability, coming from the theory of abstract data types, also relevant to functional programming, which can be of special interest when dealing with this type of property [40].

The formal specification language and the programming language can be distinct, and the verifier and compiler can be different programs that do not interact with each other. In spite of that, some languages, like *Eiffel* [64] and *Dafny* [55], provide a way to specify the code using the same language as their implementation, or a functional subset of it, and verify the code during compilation. Thus, forcing the programmer to make code that follows the defined contracts.

Formal verification has also been tested in industry, and it is used by some big companies like *Amazon Web Services* [72], using, for example, *TLA+* [71] and *Dafny* [81] to write contracts and model-checking techniques for verification; and *Facebook* [14], which uses tools like the static analyser *INFER* for software verification, and has experienced the integration of fast iteration-based development and formal methods. This industry adoption is a good sign for the practicality of formal methods, to the point of them spending more resources to improve it and create useful tools.

It must be noted that formal methods come with effort and learning curve that is not justifiable for every project, but its reliability and maintenance cost reduction are worth for other cases, including when the stakeholders consider failures to be critical (having possibility of human injury, equipment failure, reputation loss, major financial consequences, among others). The bigger effort in formal methods is often discussed, but most projects do not provide data about timescales and costs [93].

A challenge of the area is that most programmers are not used to writing those specifications, so they need to be careful to design them to capture the important aspects of the system [72]. The specification is arguably harder to write in some cases, as the implementation can be very succinct, making stakeholders ponder about the effort and results of such an approach to software

development.

2.2 Dynamic and Static Analysis

In the following sections, we discuss techniques that include some sort of testing or analysis of a program's behaviour. These are mainly divided in two types: *dynamic* and *static*, with some tools using both techniques in different aspects or steps.

Their names give a good intuition about how they work. Dynamic analysis is based on the verification of program results to understand its behaviour [6], which means that it tries to get information about the program by running it. Static analysis, however, tries to analyse the program just from the source-code itself, deriving information from it without running the program itself [24].

We can find that both techniques are somewhat complementary. Each one has their pros and cons, as expected, and this synergy is used by many researchers and developers by employing them together [25]. We discuss some of their differences now.

As dynamic analysis tries to generalize the behaviour of a program by running it a limited amount of times, this may not be fully representative of the code behaviour as it is based on particular executions. Static techniques, on the other hand, analyse the code in a way that does not need to generalize the behaviour from particular observations, as it tries to encompass all possible executions of the code.

Also, by requiring the program to be executed with real arguments, dynamic analysis depends on the generation of program arguments, or the usage of fixed inputs. While the opposite is true for static analysis, because it does not rely on concrete values for parameters nor code execution. This independence may be desirable, as it avoids some overhead of code execution, which may be substantial in some cases. But, in other situations, one may want to take into consideration the frequency of different arguments in practice, which is only available by using dynamic techniques [66], as this is useful if they want to give more weight to common values. This is not useful in our specific problem, as we try to validate code, and the errors may precisely be caused by odd and rare arguments.

On the other hand, the current trends of development and deployment of code, which include using dynamically linked libraries or even *Java* bytecode, make static analysis less powerful, because it depends on large parts of the program being available for it to analyse [66]. Also, because this type of technique analyses the entirety of a program, it may generate analyses that are too complex to be easily understandable or appliable.

It is also important to note that static analysis is undecidable [53], so even with infinite resources we would still have to make compromises and approximations. This means that this kind of analysis cannot always take all the program information into consideration, but it tries to analyse to the greatest degree achievable.

2.3 Software Repair

Programming errors happen, and programmers spend a lot of time trying to find out what is their cause while debugging [92]. That is why different techniques were developed to deal with bugs, including *automatic software repair*, for which a lot of research has been made since its initial proposal. Indeed, a large body of knowledge addressing the automatic repair of programs exists (as opposed to the repair of their contracts or formal specifications) [37][34].

The need to describe accurately what *bugs* are and their cause, which is so fundamental for this area of study, led to some concept definitions that are commonly used in the field [68]:

- A **failure** is an observance of unexpected behaviour;
- An **error** is an incorrect program state that is generated by the program, and generates a **failure** following the continuation of the code execution if it becomes observable to the user;
- A **fault** is an incorrectly written code that leads to an **error**.

One of the approaches to automatic software repair is focused on fixing programs with faults by locating the problem in the code and changing it. This is desirable because it helps programmers to write code with fewer faults, relieving them of the effort and time that would otherwise be spent looking for their root. The usage of the term software repair for techniques that eliminate faults became the standard.

There are other approaches that do not eliminate the fault, but try to recover from them at runtime. One of such techniques is runtime repair [68], which is more concerned with runtime errors and how to make software survive them [79]. They use different techniques to allow the program to heal itself or to continue without being killed by an internal error and minimizing their consequences.

We can realize a program is faulty when the implementation of the code creates a program that deviates from the expected behaviour. So, automatic software repair depends implicitly on knowing the intended behaviour of a program, so it can be corrected in case the program is detected as faulty. This is achieved by the use of an oracle [68], which is consulted to verify what is correct behaviour and if an action was expected.

Trying to understand whether a program conforms to the expected behaviour and the root cause for a failure is a complex task. There are two main analysis modes for software, that use different oracle types:

- **Dynamic** techniques depend on tests, which are usually used as oracles; by running the program with concrete arguments and verifying if the tests pass or fail, we determine if the program is correct. There are other alternatives, like the one used in property-based testing [31] where properties are used as the oracle, while still depending on tests;

- **Static** techniques try to understand if the code is correct or where the error is by analysing the source-code itself. Even though dynamic tools are more common, some tools use the formal specifications as oracles, falling into the static category. Examples include *AutoFix* [76], which fixes *Eiffel* code, and uses their own contracts as oracle; the work developed by Gophinat et al. [36], which uses the *Alloy* tool to describe the behaviour of *Java* programs; and *HINTS* [61], focused on providing hints to students on exercises, by using a correct solution as oracle.

Test suites are common practice in programming. They can easily show failures, but, as seen before, alone, tests demand a lot from the programmer when something breaks. The steps to manually fix code after a failure are:

1. Find out what error causes the failure, which can also be achieved with the help of a debugger;
2. Understand which is the fault that leads to the error and failure;
3. Analyse the test and related code to ponder about possible fixes;
4. Check if a fix is valid by changing the faulty code and verifying if the program passes the tests;
5. Choose a fix to apply to the source-code.

Even though in this project we are focused on repair of software with contracts, research has also been done for automatic implementation repair tools for code with no annotations. For example, Chen et al. [17] presents a tool that can fix some errors by analysing the states of the program at different points and using dynamic and static techniques to locate and fix them.

Most methods of automatic software repair fall under the following categories [37]:

- **Heuristic**: tools of this type try to localize the fault, generate possible fixes for it [47], and validate them to find possible solutions;
- **Constraint-based**: this method starts by using symbolic execution, or other approaches, to construct a *repair constraint*, an expression that should be satisfied by the code. It then generates solutions for that constraint, by using techniques such as contract solving. Examples include *Semfix* [73] and *Angelix* [62];
- **Learning-based**: this technique makes use of machine learning algorithms, which can be used in different ways, like ranking candidate fixes by using a model trained on a corpus of correct code [60], inferring fixes with a model trained using successful patches from the source-code history [59] [11], and using models to predict the fixed code from the faulty one [39];

Moreover, another similar study area is the inference of contracts (see Section 2.5.2). This approach can not be used alone to fix programs without contracts by first inferring them and then applying techniques for contract programs, because the contracts will be derived from the implementation, creating a loop that makes the oracle (the contracts) not detect the code as faulty.

2.4 Tests Generation

We can create tests to verify the behaviour of the program, which can be done dynamically, by running the program and analysing its results, or statically, by analysing the code.

Test suits are sets of test cases, and a case contains pre-conditions, inputs and expected results [32]. They are partial in the sense that they cannot realistically test every possible input in most cases, which means important tests can always be unconsidered and that we cannot fully test a program dynamically [46]. This is something to take into consideration, because tests cannot provide all the information we may want. Although, static techniques like symbolic execution may provide a way to fully test a program [5].

Their generation is one of the most demanding aspects of software, and one of the most important, as their impact ranges over the entire program effectiveness and efficiency [3]. So, automated generation of tests is appealing, because it uses the computer's help to remove some of this burden from programmers.

Barr et al. [8] describe many crucial characteristics of tests generation in their survey. Also noting that the generation of tests is a complex problem that needs to be studied more when compared to other aspects of testing. Types of test oracles include:

- **Implicit**: implies using previous knowledge to find problematic sections of code, relying on no knowledge of the program behaviour, instead, focusing on behaviours that are, in almost all cases, considered incorrect, like buffer overflows or segmentation faults. Some tools of this type [13] [85] use program crashes as a way to classify tests;

- **Derived**: uses information derived from different sources like documentation, and system executions, among others;

- **Specified**: uses formal specification (like contracts) to provide tests. This type of oracle can take advantage of *DbC* languages like *Dafny*, which, by providing a source of knowledge to determine whether a test passed or not, using its components' post-conditions, can help make tests less dependent on human interaction. This happens because the tests do not need further indications about their expected outputs, instead, using the contracts to classify them as passing or failing, resulting in tests relying only on the initial state for a function call.

Different tools can also verify the path of the execution [35] to try to automatically generate tests that make all, or most, possible paths be executed. This improves the coverage of the test cases, which can improve their usefulness. This can also be used to statically generate tests, and many tools that do this are available, e.g. *CUTE* and *jCUTE* [84] developed for *C* and *Java*,

respectively, use concrete and symbolic execution to avoid redundancy in test cases, a technique called concolic testing; *Klee* [12] uses symbolic execution focusing on test coverage; *S2E* [18] uses complex analysis techniques achieving the investigation of multiple execution paths at once and being able to analyse binaries, which means it works on proprietary software.

Many other, more specific, techniques to generate or improve tests were presented as well. Regression testing can be used to select the tests from a suite to get more effective, and time and space efficient ones to re-test a new version of a software [78].

An example of such a tool that generates tests is *AutoTest* [19]. To generate input data for tests, the tool creates random objects of the needed classes, it does this by first calling a random constructor, followed by a call to a random routine with side effects, with the goal of adding some variation to the pool of objects of that class. This strategy was design as a temporary solution, but, according to Ilinca Ciupa and Andreas Leitner, it provided "surprisingly good results" [19], not necessarily expected because of the simplistic nature of the strategy, being designed as an ad-hoc solution.

Dafny also includes a module that generates tests, named *DafnyTestGeneration*, which will be explained in Section 3.4.2. This module is specially focused on finding counter-examples to components that fail to verify.

2.5 Contract Generation

Different areas dealing with contract analysis have been studied, some of them deal with generation of contracts, for example, by repairing faulty ones or inferring them from the code. This section discusses these two approaches.

2.5.1 Contract Repair

The research on contract repair is valuable, for example, in scenarios where the verification fails, but we can assume the implementation is correct, so, suggestions to repair the contracts based on the implementation would be helpful. Examples of these use cases include: programmers having a reference implementation for an algorithm and still analysing its contract options; precisely documenting the guarantees of the environment where the software will be deployed by suggesting appropriate specifications; helping in the educational context, where it is common for students to have an implemented program and be asked to provide a specification, being able to use some help in the form of hints. This is specially relevant because students are known to struggle when writing formal specifications [52].

By verifying their software, programmers can be sure that it follows the specification. But, when a program fails to verify, it either means the verifier was not smart enough to prove the contract is followed, or that there is a mismatch between specification and implementation, indicating that at least one of them is not correct. Even though most software repair research assumes the implementation has a problem, the specifications can be the problem and the false sense of security can cause issues [57].

The only tool already proposed for contract repair, *SpeciFix*, was made for *Eiffel* [75], but its fix generation techniques can be applied to other languages. One of the main limitations of its algorithm is that the generalization of particular executions depends a lot on the quality of the generated tests. It suggests changes to specifications so that every contract is not violated by the implementation, which is used only for the generation of tests, applying the following steps:

- Automatic generation of test cases using *AutoTest* [19] (some tests are used only for validation purposes to avoid overfitting);
- Use weakening and strengthening techniques to provide fixes for all the trace routines;
- Validate fixes by verifying if the fix passes all the tests or at least does not break a previously passing test;
- Rank the fixes; the least restrictive ones are preferred.

Another work has focused on repairing formal specifications but without considering implementations as oracles, including for repairing OCL constraints given inconsistent information bases [20] and for repairing faulty Alloy specifications using test cases [89] and reference specifications [10][95][16]. As far as we are aware, no work has focused on the repair of Dafny, either program implementations or their contracts.

Because, as of now, only one tool has been developed for contract repair using implementations as oracle, more research in this area needs to be conducted, which may result in the developing of more techniques. Relevant applications from different problems may exist, e.g. proof automation is very similar in some aspects to program repair [80], so, tools made for one purpose may have applications in the other context.

2.5.2 Contract Inference

Instead of fixing contracts that already exist, we can also generate them *ex nihilo*. This may be harder to do, but it can provide similar outcomes when compared to contract repair, namely, helping programmers to get better contracts for their use cases [82].

Inferring contracts from code has practical consequences, as most software is created without them. That is the reason for the development of tools like *AutoInfer* [90], which can infer contracts from programs written in *Eiffel*. *AutoInfer* works roughly by doing the following:

- Automatically generates tests using AutoTest [19];
- Analyses the changes in variables, to create *change profiles*;
- Using machine learning techniques, it extracts correlations between variables;
- Collects all the generated contracts and returns them.

Other techniques include the use of genetic algorithms to detect invariants, like the one developed for EvoSpex [67]. This tool uses mutation to generate pairs of expressions that may indicate incorrect behaviour of the program (called invalid pairs), the values to be mutated are valid behaviours of the program detected dynamically (called valid pairs). This approach has some drawbacks, like the effectiveness of the generation of the invalid pairs, which the authors try to mitigate using other strategies, like making the valid pairs as encompassing as possible.

However, there are still many challenges in the area. One of the most critical problems is finding the sufficient and necessary pre-conditions to prove the post-conditions of a method [22]. This has a clear consequence of automatically generating pre-conditions given a function and its post-condition, which saves valuable time from programmers working with contracts.

2.5.3 A Fundamental Problem

Another problem that we need to take into consideration is the *The Assertion Inference Paradox*, described by Furia & Meyer [33], which can be summarized in the following points:

- The goal of validation is to verify the correctness of programs;
- We verify the correctness of programs by comparing their implementation to the specifications created by the programmers;
- We can infer a specification from the implementation;
- We can use those generated specifications to verify the system.

This creates a vicious cycle: we have a specification being checked for an implementation that was responsible for the specification in the first place, which means the verification will succeed. This problem may not be observable when inferring intermediate annotations necessary for automated verification, as there is a defined goal, to help the verifier prove the post-condition follows from the pre-condition and the implementation.

We can then ask ourselves what was the point of having this specification in the first place. We can justify our goal to repair specifications with the following assumptions:

- Sometimes programmers make mistakes on the specification, and the generation of fixes can be used to remove those mistakes — this is specially useful for students. But this depends on the user checking the fixes and analysing if any of them matches what the program is supposed to do;
- Other times, errors happen at the program implementation, and specification fixes can be used as a clue for more advanced programmers to realize they have an implementation problem. For example, a programmer could understand that the fix to the contract is explaining to them what their implementation is doing, and finally realize what is wrong with it.

As it is possible to see, our work has a fundamental assumption that the generated contracts are made to and will be analysed by programmers.

2.6 Invariant Detection

One of the approaches to repair contracts, including the one used by *SpeciFix* [75], is to generate expressions to weaken and strengthen contracts. Those are expressions that should be part of the contract but are not. One way to search for them is by detecting invariants in entry and exit points of functions, as they would provide information that could be contained in the pre- and post-condition of the function, respectively. That is why we introduce invariant detection in this section.

2.6.1 Definition

An invariant, in the context of a program, is a valid expression that always evaluates to true at a particular program point. Loop invariants are one of the most used types of invariants, representing recurring assertions that can be always observed during each beginning and end of loop iterations. Examples of places where we can find those are *assert* statements and contract specifications [28]. This information is very useful for software verification, repair, and fault localization [2]. They are very used when creating formal specifications [28], e.g. pre- and post-conditions are invariants that need to be followed by the component they describe.

Most programs do not include this kind of information, and, instead of expecting programmers to annotate their programs with invariants, different techniques of automatically detecting them were developed [28]. Invariant detection tools and algorithms using static and dynamic techniques have been proposed.

The accuracy of dynamic detection is dependent on the coverage and quality of the tests [8] and it can be a somewhat computationally expensive process. This is also why combining dynamic invariant detection with static verification of those detected expressions may be desirable [74], that way we can really be sure the invariants are accurate.

Challenges of this area include, for example, the complexity of programs or structures (e.g. detection of invariants for sequences) [51], which can make the invariants more complex as well, causing them to be harder for tools to detect. The limited data used to detect them may be a problem for dynamic detection if the user does not choose representative values for the tests.

An interesting problem is the inference of loop invariants, which are statements that evaluate to true at each execution step of a specific loop, i.e. it needs to hold on entry and exit points of every iteration of the loop. Their automatic generation is one of the biggest challenges for full-automation of proofs [33] using the Floyd-Hoare approach.

The invariant needs to be a weakened form of the post-condition of the method the loop is inserted on, and this observation leads to possible implementations to take advantage of that [33]. But even without taking this observation into account, many approaches were developed for this specific problem.

Even though this is on a lower level part of the program when compared to our main goal (i.e. loops, instead of methods and functions), it can give us important results. As a matter of fact, tools like *Daikon* can provide invariants for any point in a program, including loops, as explained below.

2.6.2 The Daikon System

A well-known example of an invariant detection program, *Daikon* [27], receives as input something similar to an execution trace and generates a set of invariants that are likely to be followed using statistical analysis. It detects properties that can be observed in different points of the program, following its execution.

Some front-ends are available for some languages like *C*, *C#*, *Java*, and *Eiffel*. They provide a way to generate the input trace from the code, even though the arguments must still be manually supplied by the user, the generation of the file is done automatically and passed to the Daikon program. This is achieved by changing the source-code to log the values of function calls to a file. To get traces for *Dafny*, users would have to generate the input by themselves, as no *Daikon* front-end is available for this programming language.

Daikon has a list of basic invariants, which were deemed relevant by its authors, and it works by testing them against the points provided [4]. This list of invariants can also be expanded by programmers, by explaining to Daikon how to test for them. The following is a list of invariants included in the tool [28], where x, y are variables and a, b, c are computed constants:

- The value of a variable x is uninitialized;
- Constant: $x = c$;
- Small set of values: $x \in \{a, b, c\}$;
- Range: $a \leq x \leq b$, $a \leq x$, $x \leq b$;
- Non-zero : $x \neq 0$;
- Modulus: $x \equiv c \ (mod\ a)$, $x \not\equiv c \ (mod\ a)$ (this is reported if all the other values but c are verified);
- Any of the above invariants over $x + y$;
- Any of the above invariants over $x - y$;
- Linear relationship: $y = a \times x + b$, $z = a \times x + b \times y + c$;
- Comparison: $x = y$, $x < y$, $x \leq y$, $x > y$, $x \geq y$, $x \neq y$;
- Unary function: $y = f(x)$, and f is absolute value, negation or bitwise complement;
- Binary function: $z = f(x, y)$, and f is min, max, multiplication, and, or, gcd, a comparison, exponentiation, rounding, division, modulus, left or right shift;
- Lexicographic order for sequences: $a \leq x \leq b$, $a \leq x$, $x \leq b$;
- Non-decreasing, non-increasing or equal values for a sequence;
- Invariants that evaluate to true for all elements of a sequence;

- Linear element-wise relationship over two sequences;
- Comparison element-wise over two sequences;
- Subsequence: y is a subsequence of x;
- Reverse: $y = rev(x)$;
- Membership: $y \in x$.

Daikon also creates more variables, e.g. $size(s)$, $min(s)$, $max(s)$, $s[i]$ for a sequence s and a numeric value i. These can be used in the expressions above to create invariants like $size(s) > 2$.

These invariants are reported if they are deemed statistically relevant by the tool, which observes the number of samples and confidence level — each invariant has different implementations of its own way to calculate this level. Daikon also takes into consideration the redundancy, so it provides more relevant invariants, by not reporting unnecessary ones.

One of the challenges of the tool is that it is not reasonable to test all the invariants for every element of a sequence, which can be solved, for example, by sampling it [4].

The simplistic nature of *Daikon*'s algorithm means that it can only detect some types of invariants. Even though we can add more types of patterns, the tool is the same, and the number of patterns is very influential on its detection of invariants, which may be undesirable when dealing with completely generic code. Another point of configuration is that *Daikon* allows users to specify a Boolean variable that can be used to produce two different sets of invariants, according to their concrete evaluation (see Section 5.3).

Daikon has been one of the most used tools for the detection of invariants, some projects that used it include: *ESC/Java* [74], where it is used to generate annotations for *Java* code to try to verify the code cannot generate runtime errors; Harder et al. [43] use it to help generate, augment and minimize test suites, the invariants are used to generate the code's behaviour abstraction; Toh Ne Win and Michael Ernst [69] use it to help with theorem-proving algorithms, by using the runtime properties of the program to guide the solvers.

One property of *Daikon* is that it detects too many invariants, including many that are not useful. Some approaches were suggested for trying to make them more relevant [26]. Some of these techniques can also improve the time costs of invariance detection, which is an important point.

Several tools make use of, and improve on, *Daikon*. For example, *iDiscovery* [94] uses symbolic execution to improve the quality of invariants computed by *Daikon* by following a feedback loop of instrumenting the generated invariants into the code, symbolically executing the code to generate new tests, and feeding the new tests into *Daikon* to refine the results. *Daikon* has been used and extended in many other contexts which include the detection of logic vulnerabilities in web applications [30], invariants for relational databases [21], distributed systems [38], robotic systems [45], and Ethereum smart contracts [58].

2.6.3 Inference with DySy

Csallner et al. [23] focus on making a tool that provides fewer invariants than *Daikon*, by providing less redundant invariants and more relevant ones. Their approach is based on a mix of static and dynamic analysis, and symbolic execution, which is based on executing the program without real input, but symbols representing them. The proposed tool, *DySy*, prefers a static-analysis overhead in favour of the ability to summarize invariants, unlike *Daikon*, which outputs many invariants, including redundant and irrelevant ones.

The tool works by doing the symbolic execution of methods to detect invariants, also detecting which of those are class invariants in an effort to simplify the method's invariants. It depends on heuristics to deal with the paths, as they are based on the execution of the code and because of that they can result in very complex symbolic execution. The authors compare it to *Daikon*, stating that the tool can provide most of the expected invariants, slightly behind *Daikon*, but their invariants are more relevant, as they achieve similar results using fewer sub-expressions.

2.6.4 The Aligator Package

Mathematica is a software that can be used for many types of mathematical computations and data manipulation [91]. *Aligator* is a package for it that detects polynomial loop invariants [49] using static techniques.

This package can detect invariants for *P-solvable* loops, which is a property of most programs using numbers [50]. They are a subset of all the possible imperative loops, which follow the definition: "the value of each program variable can be expressed as a linear combination of algebraic exponential terms in the loop counter with polynomial coefficients in the loop counter, with the property that the exponentials are polynomially related." (Kovács, 2007) [48].

Aligator shows a different side of the invariant detection world when compared to *Daikon*, by showing the usefulness of mathematical concepts with computational logic to program verification [91].

2.6.5 Other Examples

Another example is *DIDUCE* [42], which dynamically extracts invariants from program executions, starting with the strictest invariants and gradually relaxing them when it detects a violation from continually checking the program's behaviour against them. Also, in the context of hardware design, *IODINE* [41] uses dynamic analysis to infer likely invariants based on design simulations.

Chapter 3

The Dafny Language

Dafny [55] is a high-level imperative programming language created for program verification. The language has found usage in several companies that want to generate reliable software. For example, *Amazon Web Services* uses it to write and prove a variety of security-critical libraries, like encryption ones[1]. At *Consensys*, the leading *Ethereum* software company, the language is used in several components, including the verification of smart contracts [15] and consensus protocols.

This chapter will not explain the complete grammar of the language, but it will try to explain its essential aspects for this project.

3.1 Implementations

Dafny is an imperative language with functional and object-oriented programming (OOP) features. Even though *Dafny* has non-functional features, the specifications must be defined in the functional subset of the language, thus not having side effects and not being able to include methods, instead allowing functions to be used. These specifications do not clutter the compiled code because they are used only at the verification stage but dropped when compiling the program.

The language has a static program verifier responsible for its *correct-by-construction* approach, i.e. the components and contracts are verified at compile time, and the program is only compiled if the verifier validates it. Hence, the language guarantees that the executable will always follow the defined specifications. The output can be code for different languages, like *C#*, *JavaScript*, *Java*, *C++*, *Go*, and *Python*.

3.1.1 Functions and Methods

Dafny makes a clear distinction between functional and imperative code, as syntactically, there are two different types of procedures: functions, written functionally without side effects, and methods, written imperatively with the possibility of altering external state and including loops.

Functions can be declared using the keyword `function`, their body should be a single pure mathematical expression that defines its return value, and they can read external state, even though

[1] https://github.com/aws/aws-encryption-sdk-dafny

they cannot modify it. Another keyword is `predicate`, which is a way to indicate a `function` that outputs a Boolean value. We can see examples in Listing 3.1, where we can also see that predicates behave as functions but use an alternative syntax where we do not need to indicate their return type.

```
function sign(x : int) : int
{
  if x > 0 then 1
  else if x < 0 then -1
  else 0
}

predicate greater(x : int, y : int)
{
  x > y
}
```

Listing 3.1: Example of a a function and a predicate in *Dafny*

Methods, on the other hand, are more similar to what we would expect coming from imperative languages like *C*, *C++* and *Java*. They are declared with the `method` keyword, and their body is composed of statements that can alter and read objects' fields and run computations using mutable data.

We can see a method in Listing 3.2, that shows the creation of a variable using the keyword `var`, update it in the method, and causing external side effects by printing something at the end. The example also includes a while loop, with a syntax similar to other imperative languages.

```
method biggestOddDivisor(x : int)
  requires x > 0
{
  var current := x;
  while current % 2 == 0
    invariant current > 0
    decreases current
  {
    current := current / 2;
  }
  print "The input is a multiple of ", current, "\n";
}
```

Listing 3.2: Example of a method in *Dafny*

Methods, functions, and fields can be declared ghosts with the keyword `ghost`. This property indicates they can only be used for specifications and will not appear in non-ghost implementations or the compiled program. They provide a way to make separated declarations for contracts without cluttering the compiled code or allowing them to be used in the executable. In Listing 3.3, we can see an example of that and a verification error detected by *Dafny* when a non-ghost implementation calls ghost components.

```
ghost function successor(x : int) : int
{
  x + 1
}

method printUntil(n : int)
{
  var i := 0;
  while i < n
  {
    print i;
    i := successor(i); // Error: a call to a ghost function is allowed only in
    specification contexts (consider declaring the function without the 'ghost'
    keyword)
  }
}
```

Listing 3.3: Example of a ghost function in *Dafny*, indicating the error detected by the language

3.1.2 Classes

Classes are a way to define types in *Dafny*, following the OOP paradigm. They are ways to structure code in components defined by their own attributes and actions, including concepts like inheritance, encapsulation, and composition, which are ubiquitous in OOP.

They can have fields, where each field `f` has a type `T`. Variable fields cannot be initialized during its declaration, which uses the `var` keyword, and need to be explicitly typed, as in `var f : T`, while constant fields can be initialized, being declared using the keyword `const` instead. They can also be declared `ghost` to only be used in specifications and be omitted from the compiled program.

Classes can also have functions and methods, with a special notation for constructors using the keyword `constructor` and being called using the keyword `new`. They may be declared as `static`, like constant fields, if they can be accessed by the class and not from its objects, also not having access to the instance state. We can see all those properties in Listing 3.4, where an object of class `Finite` represents an integer between 0 and 10.

```
class Finite {
  static ghost const max := 10
  var value : int

  constructor(n : int)
    requires 0 <= n <= max
  {
    value := n;
  }

  ghost predicate isMax()
    reads this
```

```
13   {
14     value == max
15   }
16
17   function successor() : int
18     reads this
19   {
20     value + 1
21   }
22
23   method increment()
24     requires !isMax()
25     modifies this
26   {
27     value := successor();
28   }
29
30   static method zero() returns (f : Finite)
31   {
32     f := new Finite(0);
33   }
34 }
```

Listing 3.4: Example of a class in *Dafny*

3.2 Specifications

Dafny proves the program's correctness by proving that small parts of a program are correct, even in contexts with data abstraction and external variables using *dynamic frames* [44]. One can achieve this by adding specifications for methods (using the keywords `modifies` and `reads`).

Those specifications are written in pre-conditions (using the keyword `requires`) and post-conditions (using the keyword `ensures`), among others. It uses, following *Eiffel* [56], the same syntax for expressions at statements and specifications, but only pure functional expressions can be used on the second one, with additional support for quantifiers, exemplified by Listing 3.5.

```
1 ensures a.Length == old(a.Length)
2      && forall i :: 0 <= i < a.Length
3         ==> a[i] == old(a[i])
```

Listing 3.5: Example of a post-condition with quantifiers in *Dafny*

This assertion tells us that the variable `a` before the function call, represented as `old(a)`, has the same length as the same variable at the end. Moreover, the quantifier allows us to test for all elements of the array (we need to check all values $\texttt{i} \in [0, \texttt{a.Length})$), that the objects inside `old(a)` and `a` are the same. This post-condition is valid only when we have an array `old(a)` at the start, and the same variable at the end contains an array `a` with the same elements as the starting one. It does not tell us that `old(a)` and `a` are the same object (which means they are the

same reference) or that the values inside the arrays do not mutate; it just tells us that both arrays have the same elements.

The functional subset of operators for *Dafny* includes:

- Logical operators: equivalence, implication, reverse implication, conjunction, disjunction, and negation;
- Equality and inequality;
- Comparison operators: less than, at most, more than, at least;
- Bitwise operators: OR, AND, XOR and complement;
- Left-shift and right-shift;
- Collection membership and non-membership;
- Collection operators: disjointness;
- Arithmetic operators: addition, subtraction, multiplication, division, modulus, and unary minus.

Nevertheless, beyond operators, we can also use any function in specifications, including user-defined ones, lambda expressions, literal expressions, and identifier expressions.

3.2.1 Example Programs

One can create an example by implementing a method `rem` that takes a numerator n and a denominator d and returns the least positive remainder r after their integer division q.

As we know that d cannot be 0 and r needs to satisfy, by the Euclidean definition [9], $n = q \cdot d + r$ and $0 \leq r < |d|$, it is possible to model the function specification. Such a function can be seen in Listing 3.6, using the already-implemented remainder operator `%` for the implementation.

```
method rem(n : int, d : int) returns (r : int)
  requires d != 0
  ensures n == (n / d) * d + r
  ensures 0 <= r < abs(d)
{
  r := n % d;
}
```

Listing 3.6: Implementation of `rem` in Dafny, declared in an *examples.dfy* file with `div`

Dafny validates this specification because its remainder operator follows the same Euclidean definition.

Another example could be an incorrect program presented in Listing 3.7. Let us make a similar method, but for integer division, `div`, that divides two non-negative integers n and d and rounds the result down, r. So, we can add the requirement that they are non-negative to the specification and add some expressions the function will satisfy, like $r \geq 0$ and $n - r \cdot d \in [0, d)$.

```
method div(n : int, d : int) returns (r : int)
  requires n >= 0
  requires d >= 0
  ensures r >= 0
  ensures 0 <= n - r * d < d
{
  r := n / d;
}
```

Listing 3.7: Implementation of `div` in *Dafny*, declared in an *examples.dfy* file with `rem`

However, there is a problem with this, which will also be informed to the user by *Dafny*, exemplified in Listing 3.8. We forgot that we need $d \neq 0$, as divisions by 0 are not allowed. This mistake means we must either fix the implementation, do another operation when $d = 0$, or change the contract to require $d > 0$.

```
/path/to/program.dfy(7,9): Error: possible division by zero
Dafny program verifier finished with 0 verified, 1 error
```

Listing 3.8: *Dafny*'s output when trying to compile the `div` method

3.2.2 Mutable Objects

A *Dafny* method cannot alter the exterior state by default but is forced to declare which memory it can modify, as verification is easier when it knows exactly what each method can alter in the exterior state. We can indicate variables that can be mutated inside the method using the `modifies` keyword, which receives an argument representing an object that can be updated in its implementation.

For example, imagine we have a class with an integer variable and a function that increments it, like in Listing 3.9.

```
1  class Counter {
2    var count : int;
3
4    constructor () {
5      count := 0;
6    }
7
8    method increment()
9      ensures count == old(count) + 1
10   {
11     count := count + 1;
12   }
13 }
```

Listing 3.9: Implementation of a counter `Counter` class that does not verify in *Dafny*

This code does not verify because `count` is an attribute of `this`, which cannot be updated by the method `increment`. We would have to add line 9 to increment's specification to fix this contract, like in Listing 3.10.

```
class Counter {
  var count : int;

  constructor () {
    count := 0;
  }

  method increment()
    modifies this
    ensures count == old(count) + 1
  {
    count := count + 1;
  }
}
```

Listing 3.10: Fixed implementation of a counter `Counter` class in *Dafny*

3.2.3 By Methods

Programmers can be less expressive in functions' post-conditions, unlike in methods, because *Dafny* analyses their body expression when trying to verify the program, being able to infer some assertions related to their output, and, usually, these assertions are enough for most purposes. An example where declaring post-conditions may be convenient is recursive functions that have assertions that need induction to be proved, like in Listing 3.11, where, to prove a factorial is always a positive number, *Dafny* has to use inductive proofs, done implicitly in this case. We can also see how to refer to the result of the function in a post-condition, which is designated by a function call. We can also not that *Dafny* provides more complex ways to do proofs manually, using lemmas and assertions.

```
function fact(n : int) : int
  requires n >= 0
  ensures fact(n) > 0
{
  if n <= 1 then 1
  else n * fact(n - 1)
}
```

Listing 3.11: Implementation of a factorial function in Dafny

Using that same factorial function, we can show how methods and functions can work synergetically. For that, we provide a method in Listing 3.12 that should provide the same results are the implemented function.

```
method fact(n : int) returns (f : int)
  requires n >= 0
  ensures f == factf(n)
{
  f := 1;
  var i := 0;
```

```
  while i < n
    invariant i <= n
    invariant f == factf(i)
  {
    i := i + 1;
    f := f * i;
  }
}
```

Listing 3.12: Implementation of a factorial method in Dafny

We can see in the contracts of the method, specifically on the `ensures` clause, that we are comparing the result of the method to the function we defined before, thus, taking advantage of all the assertions inferred by *Dafny* for the function's post-condition.

This is, indeed, a common strategy used by programmers. We can write a function and a method and let *Dafny* verify their equivalence. The goal is to focus on the performance of methods and write functions that are easier to design, only focusing on them returning the correct values. It is also advantageous when we want to write an optimized method for a function with a straightforward definition but worse performance.

It is so common that *Dafny* also provides another syntax to do this, exemplified by Listing 3.13. In this example, we can see that the method inherits the assertions from the function, and *Dafny* automatically adds another assertion that indicates they must be equivalent. It is also important to note that the `by method` cannot have side effects; we have to declare and explicitly return the output variable `f`; and the function is automatically omitted during compilation, as it is only used for the verification of the method, which will be compiled.

```
function fact(n : int) : int
  requires n >= 0
  ensures factf(n) > 0
{
  ...
} by method {
  var f := 1;
  ...
  return f;
}
```

Listing 3.13: Example of a `by method` in *Dafny*

3.2.4 Class Invariants

Dafny allows programmers to indicate invariants that must be valid for all class objects at all times. As only methods can change the object's state, the compiler has to verify if each method guarantees a valid state for the object at its post-condition. On the other hand, we can assume the methods are only going to operate on valid states so that the invariant can be assumed in the pre-condition. In practice, this makes the class invariants be enforced for all pre- and post-conditions

of all the class methods, and forces us to pay attention to all the objects that could be modified. We can see it, for example, at Listing 3.14.

```
class {:autocontracts} CounterMod {
  const mod : int;
  var counter : int;

  constructor (mod : int)
    requires mod > 0
  {
    this.counter := 0;
    this.mod := mod;
    this.Repr := {};
  }

  predicate Valid {
    0 <= counter < mod
  }

  method increment()
    ensures counter == (old(counter) + 1) % mod
  {
    counter := counter + 1;
    if (counter == mod) {
      counter := 0;
    }
  }
}
```

Listing 3.14: Implementation of a counter `CounterMod` class in Dafny

Class invariants are generally paired with the `autocontracts` attribute to omit some boilerplate. This annotation adds the class invariant assertion (`Valid`) to each method's pre- and post-condition. In this example, we can see that `increment` assumes the object satisfies the invariant before the call and verifies that it preserves it at the end. Using `modifies` clauses is also unnecessary for changes in the class fields because they are handled when the class is annotated with `autocontracts`.

3.2.5 Recursive Definitions

Dafny's verifier tries to prove the termination of all functions and methods. When we deal with recursion, we can help it by providing an expression that decreases for each subsequent call using the keyword `decreases`. The verifier can use these expressions to prove the function always terminates.

A `decreases` specification indicates some strictly smaller expression for each recursive call to a method. This order depends on the type of argument, e.g. for integers, it is the expected operator; for sets, $x < y$ if x is a proper subset of y; `false` is smaller than `true`.

Dafny, by default, tries to identify decreasing expressions for methods and functions automatically and, lots of times, succeeds in it, but sometimes we need to provide it explicitly. Listing 3.15 is an example where *Dafny* can automatically detect that `decreases n` is a valid contract.

```
function fib(n : int) : int
  requires n >= 0
{
  if n <= 1
  then 1
  else fib(n - 1) + fib(n - 2)
}
```

Listing 3.15: Recursive implementation of the Fibonacci function in *Dafny*

The code includes a valid `decreases` expression because each subsequent call to `fib` uses a smaller argument than the previous. Nevertheless, on more complex programs, *Dafny* may not be able to automatically detect a valid decreasing expression, like in the program provided in Listing 3.16, with two mutually recursive methods.

```
method A(x : nat)
{
  B(x);
}

method B(x : nat)
{
  if x != 0 {
    A(x - 1);
  }
}
```

Listing 3.16: Mutually recursive methods in *Dafny* without contracts

And, this is not as simple as putting `decreases x` on both methods, because the call inside the method `A` uses the same argument, which is not smaller than itself. One possible solution for this case (exemplified in Listing 3.17) is to add another element to the `decreases` expression, making it a tuple.

```
method A(x : nat)
  decreases x, 1
...

method B(x : nat)
  decreases x, 0
...
```

Listing 3.17: Fixed mutually recursive methods in Dafny

This makes it so that a call to `A` with a value x evaluates the decreasing expression to $(x,1)$, then it calls `B`, where it is going to evaluate to $(x,0)$, and, if a recursive call to A happens, it will be evaluated to $(x-1,1)$. This sequence of evaluations is strictly decreasing, allowing *Dafny* to prove the termination of this code.

3.3 Development Support

Using verifiers for IDEs like *Visual Studio Code* and *Emacs* [55], programmers have real-time feedback. The author will see an error in the text editor whenever *Dafny* cannot prove something needed to verify the program. This automatic feedback can help minimize the effort and the programmer's expertise needed to use the verifier.

3.4 Dafny Architecture

Dafny has many modules. There is very limited documentation about the internal aspects of its compiler and a limited amount of comments in some parts of the code, so most of our understanding of its design comes from reading the source code itself, with some help from articles and forums. When we compile a program, the following happens:

- A thread is created to process all the compilation process on the `DafnyDriver` module;
- The program arguments are pre-processed, returning early if any problem happens;
- All source-code files are checked to verify if they can be parsed;
- The parsed program is translated to *Boogie*, an intermediate verification language [54] made for object-oriented languages that use the *.NET* framework [7];
- The program is resolved, type checked, and has its variants inferred;
- The program is finally verified;
- The compiled code for the verified program is generated. By default the code is in *C#*;
- The language server is initiated, if applicable;
- The results are presented in the configured channel, e.g. *stdout*.

The program can also be called with other purposes that parse the code and perform an action that is not the compilation, namely:

- Perform a dead code analysis, giving warnings when it is impossible for the program execution to arrive at a specific part of the code;
- Generate tests for the program (see Subsection 3.4.2);
- Format the code, currently only fixes indentation to better comply with what is considered idiomatic for *Dafny*;

3.4.1 Internal Dafny AST

Dafny's abstract syntax tree (AST) is generated at the parsing stage and changed during resolution. It contains information about the tokens that generated it and the symbols included in the program, following their hierarchical structure.

The class that represents the AST is named `Program`. A program contains information about its name and modules, a method to be used as an entry-point if the code is compiled, and the *Dafny* built-ins that can be used.

Each module contains top-level declarations, which include classes, data types, and traces. Each class, our main concern, contains a list of members, taking place as fields, functions, methods, constructors, among others. Following this structure, we can find a fully qualified member by finding its module, class, and so on.

To standardize the program structure, *Dafny* automatically adds a module to every source file, named `module_`, and a default class for every module, named `_default`. That way, for example, we do not need any exceptional cases for members declared outside of classes. Because of this, if we are searching for a component and cannot find it in the current context, we can try to find it in one of those default elements if they exist.

In *Dafny*, each `Method` contains zero or more pre- and post-conditions. These are represented by two lists of `AttributedExpression` in the AST generated by the verifier. We can then try to run this expression with concrete values to verify if they are followed. However, we need to use our own evaluator, as *Dafny* is a compiled language and does not include an interpreter.

The design for the `Method` class includes two lists, as presented in Listing 3.18.

```
public class Method {
    public readonly List<AttributedExpression> Req;
    public readonly List<AttributedExpression> Ens;
}
```

Listing 3.18: Selected attributes of the `Method` class

Declarations are accessible simply by reading the attributes of the program. For example, to read the top-level declarations, we can just write:
`input.Program.Modules().TopLevelDecls`.

3.4.2 Test Generation

Dafny has a module that deals exclusively with test generation[2]. However, it has some limitations: it works with all basic types except data types, arrays, and multisets. If the method has violations, it finds inputs that materialize them; otherwise, it just instantiates arguments, trying to consider every path or block. To do that, the tool generates one input for each path that can be taken by the function or method, achieving coverage of all the component's blocks.

[2]More information on the module's README at Source/DafnyTestGeneration

We can exemplify this by using the `rem` and `div` (Listings 3.6 and 3.7) methods, defined in an *examples.dfy* file. Using the file to run the command `dafny /generateTestMode:Block examples.dfy` results in the generation of the test file presented in Listing 3.19

```
include "examples.dfy"

module examplesUnitTests {
    method {:test} test0() {
        var r0 := rem(7719, 1);
    }

    method {:test} test1() {
        var r0 := div(8855, 0);
    }
}
```

Listing 3.19: Example test file generated by *Dafny*

As we can see, we only get one test for `rem`, as it only has one block. Also, one test for `div`, with arguments selected to invalidate our post-conditions, precisely as promised.

This functionality allows us to generate tests for a specific method using block or path coverage. It is valuable for the generation of fixes that depend on the generation of tests.

Chapter 4

Solution

In this chapter, we explain our tool, which, given a *Dafny* program that fails to verify, suggests modifications to its contract with the goal of making it compatible with the implementation and repair the fault, i.e. making the program pass *Dafny*'s verification. This is achieved by using tests to perform a dynamic analysis of the program behaviour, trying to find the mismatch between that behaviour and the specification.

In the subsequent sections, we formalize our problem, taking much inspiration from the formalization used by Pei et al. [75], which defines the algorithm used by us, and explain how their techniques to generate fixes work, as we use them as the basis for our tool (see Section 4.3).

4.1 Overview

The tool applies dynamic analysis by using tests to determine which states are problematic, and those findings will guide the fix generation stage, as the presence of failing tests is the problem we need to solve. After that, the tool has two main phases: weakening, which tries to solve the issue by allowing the problematic arguments to pass the pre-condition of the faulty routine, in case the assertion is too restrictive; and strengthening, which, as a last resource, disallows the problematic arguments to pass the pre-conditions of all the functions they pass through before the faulty one. Figure 4.1 and Figure 4.2 give an overview of the weakening and the strengthening phases, respectively. Both phases depend on a trace that has the needed information about the execution of the tests (See Subsection 4.3.1), so the invariants can be detected using the *Daikon* tool (See Subsection 4.3.3).

Our program takes as input:

- A *Dafny* program D;
- A function or method that has its contract broken r_n;
- An indication of the place *pos* where the fault it located, either a pre- or post-condition;

Figure 4.1: Overview of the weakening phase, receiving the *Dafny* program and the tests as input. This process returns valid fixes and invalid candidates that will be strengthened in the next phase, if possible.

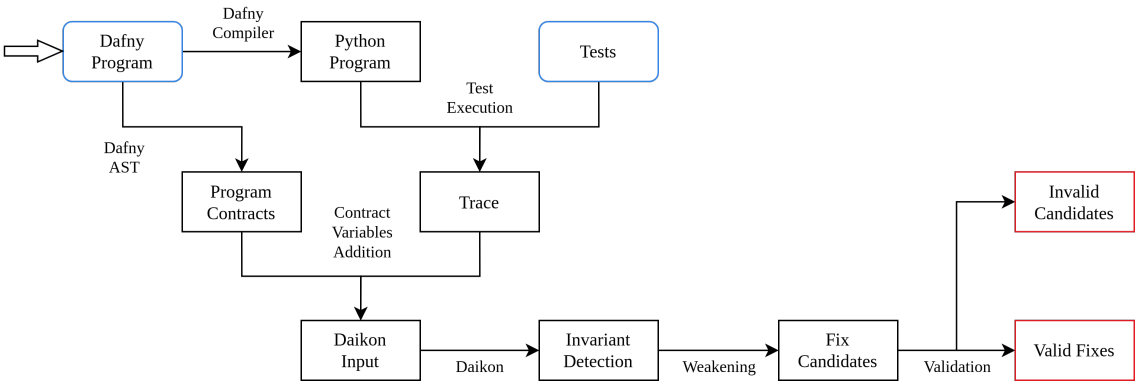
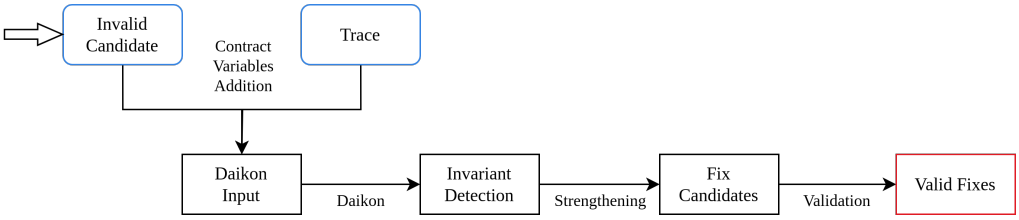


Figure 4.2: Overview of the strengthening phase. This is executed for each invalid candidate detected on the weakening phase, reusing the trace generated in that phase. This phase results in valid fixes, to be aggregated to the ones found in the weakening phase.



- The name of a function or method r_0 that is used as the outermost routine, i.e. all the tests are defined as a call to this routine. This is different from r_n because we would not get failing tests otherwise, and the strengthening techniques need previous functions in the trace;
- A list of tuples representing the concrete arguments of r_0 for each test.

Using the last two arguments we can also define A_n, the specific assertion that is faulty, by taking either the pre- or post-condition of r_n according to the value of *pos*.

To better explain and exemplify the stages of the tool, we provide the code in Listing 4.1 that will be used as an example program argument, i.e. the algorithm will try to repair it. For this program, we have $r_0 = rem$, $r_n = divRem$ and $pos = PRE$:

```
function abs(n : int) : int
{
  if n < 0 then -n else n
}

method divRem(d : int, n : int)
  returns (q : int, r : int)
  requires d >= 0
  requires n > 0 // stronger than rem's pre-condition
  ensures r + q * n == d
```

```
11 {
12   r := d;
13   var m := abs(n);
14   q := 0;
15   while r >= m
16     invariant r + q * m == d
17     invariant q >= 0
18     invariant m == abs(n)
19   {
20     q := q + 1;
21     r := r - m;
22   }
23   if n < 0 {
24     q := -q;
25   }
26 }
27
28 method rem(d : int, n : int)
29   returns (r : int)
30   requires d >= 0
31   requires n != 0 // weaker than divRem's pre-condition
32 {
33   var s_;
34   s_, r := divRem(d, n);
35 }
```

Listing 4.1: Implementation of `rem` amd `divRem` in *Dafny*

This program has a method `divRem` that calculates the quotient and the rest of the division from its two arguments, `d` and `n`, by applying an integer division algorithm. The method `rem` calls `divRem`, but only returns the rest, ignoring the quotient. The contract problems arise from the fact that the inner method has $n > 0$ as a pre-condition, while the outer method has a weaker pre-condition of $n \neq 0$, which means it allows negative numbers, while `divRem` does not. This results in the Dafny verifier not being able to prove the contracts for the method call on line 34, because some arguments can break it.

4.2 Test Definition

A test can be formally defined as a concrete function call for an object $o : T$, which has the form:

$$\tau : o.f(a_0, a_1, a_2, \ldots, a_n)$$

where a virtual or static method f of the class T has one implementation with n arguments, and, more specifically for our problem, we want $f = r_0$, because all tests should start as calls to the outermost routine. Unlike standard tests, we do not have an expected output, as we are interested in the contracts being followed, which already determines if the resulting output is expected behaviour.

It must be noted that the current implementation of the tool does not have support for objects, so o needs to be a default class for a Dafny module. We can call static methods if o itself is a class.

The execution of a test builds a trace, a sequence of function calls. Imagine we have a state $s_0 : \mathcal{S}$ that is used by our initial call. Each subsequent call could have a new state, so a trace is defined as a list of pairs (r_n, s_n), where $s_n : \mathcal{S}$ is the state when the call to r_n occurs, or the state at the output of that call, so each call appears twice in the trace. We can also define functions $pre\_state(r_n) = s_n$ and $post\_state(r_n) = s_{n+1}$ to refer to the pre- and post-states of the nth routine call.

A function or method f has a pre-condition P_f and a post-condition Q_f, and we denote a state following an assertion as $s \models A$, i.e. the assertion A evaluates to true for the state s. This is used to classify a test t, with a trace ρ_t, into: valid if the initial state satisfies the pre-condition of the outermost routine, or $s_0 \models P_{r_0}$, and invalid otherwise. Valid tests can be subdivided even further, into: passing if every call has its pre- and post-condition satisfied, which means no contract was broken, or, for each call i from ρ_t: $pre\_state(r_i) \models P_{r_i} \wedge post\_state(r_i) \models Q_{r_i}$, and failing otherwise, which means it broke a contract at some point in its execution.

In our example, we would find: a test $rem(1,1)$ to be passing, as no contract will be broken during its execution; a test $rem(1,0)$ to be invalid, as the call to the outermost routine does not satisfy its pre-condition; and a test $rem(1,-1)$ to be failing, as the call to `divRem` breaks its pre-condition, which does not allow `n` to be negative. Invalid tests are ignored by the algorithm, because they do not provide any useful information about the correctness of r_0.

4.3 Design of the Repair Generation Tool

In this section, we explain the used algorithm to generate fixes, which was an adaptation to *Dafny* of the approach used in *SpeciFix* [75] implemented for the *Eiffel* programming language. Using dynamic analysis, it takes a program with faulty contracts and suggests fixes by altering them, and it works for any language with contracts, making it an excellent reference.

This approach is the only one made for this purpose, even though alternative ones that could provide ways to work around our problem exist, like contract inference. Creating a new technique was not prioritized, but this could be a possibility in the future.

Standard software repair for contract programs assumes that the specifications are always correct, which allows the use of these as oracles or necessary verifiers of program behaviour. This assumption is not valid for us, so those tools are not helpful to our problem, as trust in the contracts is fundamental for them to work.

We will explore the different modules or stages that were developed for our tool, they include the following modules:

1. Test generation;
2. Trace generation;
3. Expression evaluation;

4. Invariant detection;
5. Weakening fixes generation;
6. Strengthening fixes generation;
7. Fix ranking;

The two different fix generation stages represent the two ways to try to fix the program:

- **Strengthening** makes the call invalid by disallowing the problematic arguments to pass the contracts, restricting the pre-conditions of routines;
- **Weakening** tries to fix it by allowing the arguments to pass through functions; this is not always possible, but it is preferred when compared to strengthening, as it makes the target routine allow more arguments instead of less, which could break compatibility if any other part of the program depends on the usage of the routine with any of the now-disallowed arguments.

These techniques are combined by first applying weakening and then trying to apply strengthening on top of the weakening candidates that failed validation, i.e. still result in failing tests when applied.

4.3.1 Tests and Trace Generation

Before the algorithm execution, we need to generate a trace of the program. To make the program execution provide this information in the first place, we made an update to the Dafny compilation, so that when we run the program we can also get its trace.

Whenever a *Dafny* program compiled with our changes is executed, a trace can be retrieved from the interactive environment with information of function call entrance and exit points and their respective inputs, outputs, and ID.

So, to generate the trace for a program, we can call each test and retrieve the trace from the program. The trace includes entries with the following information:

- Name of function called;
- Unique identifier for the call, where each identifier appears twice in the trace, corresponding to the entrance and exit points of a specific function call;
- List of arguments passed to the function;
- In case it is an exit point, the result of the execution.

Currently, the test generation is done manually, tests are provided by the programmer. This is a point of future improvement, as the addition of an automated way to generate tests is appealing, because it makes the solution more automated. As of now, *Dafny*'s test generation module is a possible option, but more analysis need to be done to verify if we can get more tests to a method or function, with the goal of creating a test suite.

4.3.2 Expression Evaluation

Dafny is a compiled language, so there was no provided way to evaluate its expressions dynamically. For that reason, we created a module to evaluate a subset of *Dafny*'s expressions with contexts. This is used extensively in the following modules.

An evaluator should receive an expression e and a context Γ that maps variables and their values, and, in our case, there are no side effects, because we are only interested in expressions that can appear in a specification in Dafny. The evaluation of an expression needs to follow rules like:

$$\frac{\Gamma \vdash a_1 \longrightarrow a_1^t, a_2 \longrightarrow a_2^t, ..., a_n \longrightarrow a_n^t}{\Gamma \vdash f(a_1, a_2, ..., a_n) \longrightarrow f(a_1^t, a_2^t, ..., a_n^t)}$$

Where f is a function or method that receives n arguments and $\Gamma \vdash x \longrightarrow x^t$ means that x evaluates to x^t in the context Γ. A way to think about this formula is that if we are dealing with functions without side effects, changing an expression e for an evaluation e' of it should give the same result, so we can evaluate each argument of a call to terminal expressions and use those values as to evaluate the call to function f itself.

Similar rules apply to other types of expressions, like operator, quantifier and identifier expressions, among others, which can be compared to a function in that they receive zero or more inputs and return zero or more values. The difference lies in the grammar of the language, as those types of expressions are treated differently from functions and methods by the compiler, because they have their own notations.

These rules are characteristic of the eager evaluation strategy, which was chosen for practical reasons, as both *Dafny* and *C#* also use this strategy, so it better represents the strategy of the compiled code, and it matches the programming language we are using.

To implement an evaluator for a subset of the *Dafny* language, the representation of built-in types and the evaluation of their operations need to be implemented, providing a module to evaluate expressions that we may encounter in real code.

4.3.3 Invariant Detection

This module is responsible for detecting the invariants that are true for the passing tests and false for the failing ones using the trace information and the contracts' information.

First it generates $\mathscr{I}$ which is a list of all the invariants that can be detected in the program. We use the filters $passing(\mathscr{I})$, $failing(\mathscr{I})$ and $invalid(\mathscr{I})$, as a notation for only keeping the respective passing, failing and invalid invariants.

To split the tests into those three categories, we use the information from the contracts. However, these may not necessarily be the same contracts present in the original program D, as they may have changes that were added in different stages of the algorithm, namely during the strengthening phase, where we build upon weakening candidates. That is the reason we receive the contracts' information as an argument.

The output of this module is a set of expressions:

$$\Omega = \{\omega | \omega \in passing(\mathscr{I}) \wedge \neg\omega \in failing(\mathscr{I})\}$$

This set aims to define what makes tests pass or fail, by including invariants that are true for passing tests and false for failing ones, ignoring the invalid ones.

In the `rem` example, we have that, using the original contracts, tests pass if $n > 0$ and fail if $n \leq 0$, and the same happens with $n \neq 0$, which makes the tests pass. So, we should expect $\{n \neq 0, n > 0\} \subseteq \Omega$.

4.3.4 Weakening Fixes Generation

The first way to fix the problem is to try to *weaken* A_n, by making it less restrictive, thus allowing the problematic calls to continue.

This module starts by calling the detection of the invariants, producing the set Ω, and adding to this set a dummy value *False*, which will be used later to create purely strengthening fixes, as those are built upon weakening candidates. As invariants are detected for every single routine, the invariants related to any routines but r_n are removed, as we want to change A_n with relevant invariants.

Each element ω of Ω creates a candidate weakening fix w, which is made by changing the faulty assertion A_n for $A_n \vee \omega$, which weakens the assertion by allowing contexts c for which $c \not\models A_n \wedge c \models \omega$. This results in a list W of all the candidate weakening fixes for the program. In our example, we should expect fixes that weaken A_n using one of the expressions $\{n \neq 0, n > 0, False\}$, as the invariants $n > 0$ and $n \neq 0$ were detected for r_n, and the dummy value *False* is always present.

The output of this module are two sets, $passing(W)$ and $failing(W)$, where a candidate is declared as passing if it passes all tests defined in the previous module (Subsection 4.3.1), i.e. the program breaks no contract.

In our example, one of the candidates f_0 is a member of $passing(W)$, because, by changing A_n to $A_n \vee n \neq 0$, we fix the fault, as `divRem` does indeed work with $n < 0$, even though A_n does not allow it to pass, so all previously failing tests turn into passing. But the other fixes do not solve the problem, and, in fact, $A_n \vee n > 0$, $A_n \vee False$ and A_n are all logically equivalent, so these changes do not change the sets of passing and failing tests, and we expect to find their candidates in $failing(W)$.

4.3.5 Strengthening Fixes Generation

The second generation phase tries to fix the failing weakening candidates by restricting some contracts. To do that, the module receives the set of failing weakening fixes ($failing(W)$) as input, and for each element w of this set, it does the following process.

First, it detects the pre-condition invariants $pre(\mathscr{I}_i)$ that hold for each function r_i that appears in the trace before a failure, so i ranges over $0...n-1$, and adds them to a set $\Sigma_i = \{\sigma | \sigma \in$

$passing(pre(\mathscr{I}_i)) \wedge \neg\sigma \in failing(pre(\mathscr{I}_i))\}$. This is done so we can restrict the problematic arguments from passing in each routine's pre-condition until r_n, so we do not get the failure if we start in any of the routines r_i.

Then, for each combination $\langle \sigma_0, ..., \sigma_{n-1} \rangle \subseteq \Sigma_0 \times ... \times \Sigma_{n-1}$, we can build a candidate strengthening fix s which corresponds to changing the pre-condition P_{r_n} of each routine r_n with $P_{r_n} \wedge \sigma_n$, creating a list S of all the candidate strengthening fixes for the program.

The output of this module is the set $passing(S)$ that represents the fixes that pass all tests defined in the previous module that have not become invalid.

In our example, we would expect a strengthening fix that changes the pre-condition of `rem` to $d \geq 0 \wedge n \neq 0 \wedge n > 0$, which, if simplified, gives us $d \geq 0 \wedge n > 0$. This solves our problem, because it disallows the negative values for n to appear in valid tests, making all previously failing tests invalid.

4.3.6 Fix Ranking

This module ranks all the passing fixes, from both fix generation phases, represented by the set $\Phi = passing(W) \cup passing(S)$. Currently, the fixes are ordered, with weakening fixes first and strengthening fixes later. Furthermore, there are no established ranking criteria among the fixes within each of these categories. A possible improvement is to rank the fixes from the same category, by comparing how much they weaken or strengthen the contracts, prioritizing fixes that invalidate the fewer amount of tests. Ideally, user studies should be conducted to better understand how to rank the fixes according to their practical relevance, but this is out of scope for our work.

This preference for the lesser restrictive fixes can be explained by them being more prone to breaking compatibly with other calls that may be implemented in other parts of the program and because of overfitting, as strengthening could, in the limit, just invalidate each single test until there are no failing tests, by disallowing each one of their respective arguments to pass the pre-condition of the routines.

The output of this module is an ordered list of all the valid fixes encountered, which is also the output of the entire program. These can then be presented to the end-user.

Chapter 5

Implementation

In this chapter, we discuss the implementation of the tool, which is added to the Dafny official release source-code as a module, it gets information from the parsed and resolved data from the AST of a *Dafny* program to apply the fix generation algorithm on. The tool, following *Dafny*'s implementation, is developed in *C#*.

5.1 Trace Generation

To make it possible to get a trace of the program, we changed the *Dafny* compilation that generates *Python* code. This target language was used so we could interact better with the runtime environment, being able to easily retrieve information from a trace variable, proceeding with the use of this information in our *C# code*. Adding an annotation to all declared methods, that adds entries to a list whenever a function is called or returns. The entries have the needed information and the return value of the call on exit points, the wrapper uses integers sequentially as unique call identifiers.

This is achieved by implementing the wrapper similarly to this:

```
@wraps(fn)
def wrapper(*args, **kwargs):
    ...
    trace.append((fn.__qualname__, unique_id, args))
    try:
        out = fn(*args, **kwargs)
    except:
    out = None
    trace.append((fn.__qualname__, unique_id, args + (out,)))
    return out
```

The exception verification is helpful when the code breaks, e.g. division by zero and infinite recursion. This can happen, as we force the code to compile even if *Dafny*'s compiler verification process did not validate the program, because we will use it to run the tests either way.

Also, to get *Dafny* to always produce the needed *Python* program, we had to change some configurations, namely: make it compile the code even if there was an error detected during the

verification step; and configuring it to compile to the *Python* language. Both can be achieved by using command line arguments.

And the generation of the trace is done by running *Python* code from *C#* using a library made for this purpose: Python.NET. The tests are sequentially run, and the trace can be retrieved at the end from a *Python* variable that accumulates all logs from the execution. Each function entrance and exit generates a log entry even if the code breaks, as we saw in the previous section.

5.2 Expression Evaluation

An evaluator was implemented using *Dafny*'s AST class `Expression` paired with a context, which has a mapping of symbols to their corresponding values. It's only focused on the functional subset of valid *Dafny* expressions, because its goal is to evaluate contract pre- and post-conditions, which are limited to that subset. The *Python* code does not include any type of mention to the *Dafny* contracts the functions are supposed to follow, so we have to evaluate them from within the *Dafny* compiler. The expressions come from the *Dafny* AST, available from our *C#* code. On the other hand, the context comes from the *Python* environment, being retrieved using a library, so it is accessible to our *C#* code.

Currently, the expression evaluation power is greater than what we can translate to *Daikon*. So not every type that can be processed on the contracts can be processed as a *Daikon* variable. Some expression types that can be handled by the evaluator are:

- Literal and variable expressions;
- Arithmetic expressions on numbers;
- Logical operations on Booleans;
- Collections (sequences, maps, and sets);
- Operations on collections;
- Collection update and selection;
- Function calls;
- If-then-else expressions.

First, the sub-expressions are evaluated recursively, and then those results are used to create an object that represents the result of the expression evaluation. Those objects need to be one of the classes that implement the interface `IResult`, which just contains the following abstract methods:

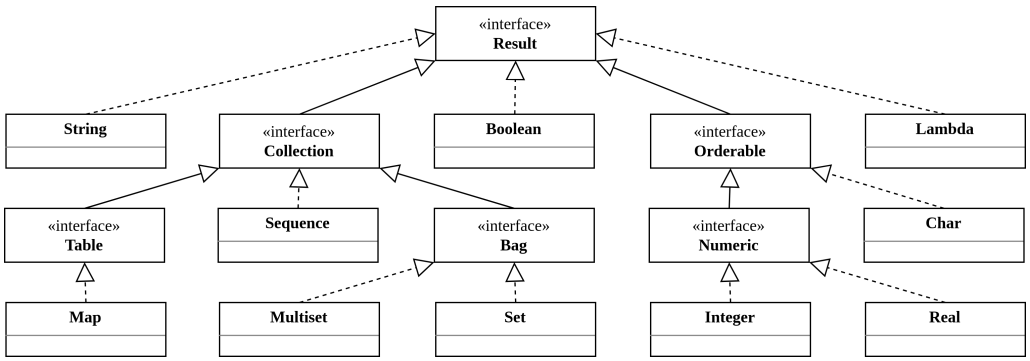
- Equality and inequality (by default it is derived from the first) for objects of the same type;
- Two functions that convert the result to a string to be used by *Daikon* or *Python*.

So any expression evaluation, when successful, will produce an object that represents a type that implements this interface. Even though we are only interested in Boolean results that come from contracts, the other classes are used for intermediate values that can be evaluated in sub-expressions. The valid types that have a class representing them are:

- Boolean;
- Char;
- Integers and reals;
- Strings;
- Maps, multisets, sequences, and sets;
- Lambda functions.

Figure 5.1 presents the way those classes are structured hierarchically, according to their valid operations.

Figure 5.1: Hierarchical structure of the different classes that implement the interface `IResult`.



5.3 Invariant Detection

We use *Daikon* [27] to detect invariants, the first step is to translate the trace generated by our program to a valid Daikon input, that can be written to a file.

The generated *Daikon* input adds extra variables that indicate if the pre- and post-conditions are followed at a function entry and exit points, they are respectively named *pre_condition* and *post_condition*. It also adds a variable to the exit point that indicates if any contract was broken by the trace generated by that call, named *inner_condition*. Using these Boolean variables, we can incentivize *Daikon* to make splits using them, helping us to identify what is valid or not valid in combinations of them.

We generate the *Daikon* input by processing each point reported to the *Python* logger, for each one of them we generate the respective *Daikon* point. The *Daikon* input file is composed of header

and trace, the header contains the definition of every point, and the trace contains the concrete values we get for those points during the execution of the code.

We keep track of every function called and reported, so we know which functions we need to add to the header of the *Daikon* file. For each method we add an enter point, with each parameter and a variable representing the evaluation of the pre-condition, and an exit one, with all the entrance variables plus the outputs, a variable representing the evaluation of the post-condition and a variable that reports if any of the calls generated by the method broke a contract (this may be thought as an inner-condition).

In the headers we have point definitions, for each one, we have the point's name, in the format `<class>.<method name>()` with indications if it is an enter or exit point for the method. And for each variable, information about its name, kind, type, representation in the *Daikon* input file, and the comparability key, which tells the tool which variables are comparable to it.

While, in the trace, we have the point name, a unique identifier for the call, followed by the variables names, values, and an indication if the variable was changed or not (this is always marked as zero in our implementation, because *Daikon* can verify if the variable was changed or not by itself).

The next example exemplifies this, and we can see the information needed for the headers and the trace generated when the method `rem(5, -3)` is called.

```
1  ppt module_.default__.rem():::ENTER
2  ppt-type enter
3  variable d
4    var-kind variable
5    dec-type int
6    rep-type int
7    comparability 2
8  variable n
9    var-kind variable
10   dec-type int
11   rep-type int
12   comparability 2
13 variable pre_condition
14   var-kind variable
15   dec-type boolean
16   rep-type boolean
17   comparability 1
18
19 ...
20
21 module_.default__.rem():::ENTER
22 this_invocation_nonce
23 192
24 d
25 5
26 0
27 n
```

```
-3
0
pre_condition
1
0
```

We can see that the variables for an enter point are its method's parameters and a contract variable representing its pre-condition. Exit points contain those same variables, plus each return variable and two more contract variables: `post_condition` and `inner_condition`. For the lack of a better descriptive kind, the contract variables are declared as `variables` in those files.

Then we also write files to tweak Daikon's configuration and splitting approach according to our needs. The configurations changed by us are:

- Lower the confidence level, so more invariants appear, even if *Daikon* is still not sure if they are valid or not;
- Add a dummy invariant level, so the invariants in the *spinfo* file are part of the output when no equivalent invariant can be found;
- Use pairwise implications, so more implications are verified by the tool;
- Enable the printing of all invariants detected, without filtering redundant ones;
- Use *Simplify* as the output language, this is so the parsing is easier, as the function names precede their arguments inside a parenthesis, like the *Lisp* language, so there is no ambiguity that needs to be solved by precedence;
- Make splits on the three artificially added contract variables: *pre_condition*, *post_condition* and *inner_condition*. This is a way to tell *Daikon* to consider splitting specific boolean variables depending on their value.

With these three files written, we can pass them as arguments to Daikon and get its output. The parsing is done for each line without any kind of filtering at first, it creates a list of tuples with the routine name, an indicator if the invariant happens at pre- or post-condition, and the invariant expression itself.

The expressions are translated from *Symplify* to the class used to represent expressions in the *Dafny* compiler, so they can be processed by our evaluator. The generated *Symplify* expressions use arithmetic, logical and comparison operators, which can be easily translated to valid Dafny's expressions using only built-in operators.

5.3.1 Filtering of Invariants

Not all invariants detected by Daikon are useful for us, so a process of filtering was implemented. The properties used for filtering are as follows:

1. We want invariants for passing and failing tests, so we filter invariants that do not refer to any contract variable (pre-condition, post-condition or inner-conditions);

2. We filter out the invariants that are not in the format

$$(contract\_variable = boolean\_value) \iff expression$$

because, according to the solution algorithm, we need invariants that satisfy

$$passing \rightarrow expression \wedge \neg passing \rightarrow \neg expression$$

3. To create a single list of invariants, we extract the expressions from the previous item, but negate the ones detected for failing tests. This may not be needed, as we could not find a situation where we gain information from the invariants detected for the failing tests when using a single *Daikon* run to detect all invariants;

4. We also remove invariants that relate only contract variables, because they cannot be used in the contracts;

5. We filter out invariants that are structurally equivalent;

6. In case we are trying to get invariants to weaken a pre-condition or to strengthen a contract, we filter the ones that have references to any output variables, as they cannot be used in those places;

7. In the weakening phase we also filter out any invariants detected for methods other than our faulty method.

5.4 Fix Generation

The fix generation implementation tries to be a translation of the process explained in the solution at Subsection 4.3.5 and Subsection 4.3.4.

To represent the two types of relevant contracts, pre- and post-condition, the `ContractType` enum was created, it has two values, one for each of those types. To represent fixes, a class `ContractManager` was created, it includes information about every contract that is modified by that specific fix, and their new values. To represent invariants, a pair of type `(ContractType, Expression)` is used.

The weakening stage receives the trace and starts by detecting the invariants for the received trace, as explained in Section 5.3. It then uses the list `o`, of candidate assertions, where each assertion is an invariant, to create a list of candidate weakening fixes `fs`, where each element represents a fix that uses one of the elements in `o` to weaken the faulty contract.

Now we validate those candidates, splitting them into two lists: `phi`, representing the valid weakening fixes and `w` representing the others, which will be passed as arguments to the strengthening stage together with the trace.

The strengthening stage can be divided in two substages: the generation of candidates and their validation. The first substage runs many times, one for each element in `w` plus a dummy fix to generate pure-strengthening fixes, while the second one is executed once, at the end.

To generate the candidates for an invalid weakening fix $a \in w$, we start by detecting the invariants assuming the contract has been weakened according to `a`, keeping track of the functions that were called before the faulty routine during the execution. Then, we create a map to relate each method with their detected invariants, create a list of every possible combination with exactly one invariant per method and a list of candidate fixes corresponding to each one of those combinations.

Now we can validate all the fixes, removing the ones that are invalid and returning the list of fixes that passed the tests `p`. At the end of the fix generation, we can join the lists of valid fixes $phi \cup p$ to create a list of all valid fixes.

The result of the program is printing of each one of those fixes, which are objects of the class `ContractManager`, to standard output, and indicate which methods to change in the following format: `Change <place> of method <method> to <new contract>`, where `place` is a `ContractType` object, `method` is a string representing the name of the method and `new contract` is an expression that represents the new pre- or post-condition of that method.

Ideally, this output could be presented to a user in a plugin so that it could select one of those fixes to be automatically applied to their source code. But this is left as an improvement to be implemented.

Chapter 6

Solution Evaluation

In this chapter, we explain our methodology to get results and discuss the resulting data after the experiments and evaluation. The goal is to analyse the effectiveness and efficiency of the tool, as well as a preliminary evaluation of the quality of the generated fixes.

6.1 Evaluation Setup

In this section, we compare the number of example programs that could be fixed by our tool, and the measurement of relevancy by programmers could also be considered. The gathered information for each example, besides the average runtime for different parts of the program, is presented in the next list. The results are split between weakening and strengthening, because the strengthening process is executed more than once in some cases, and analysing their values separately can be useful.

- **Tests #**: the number of tests that were generated and executed;
- **Invariants detected #** at the weakening process: the number of invariants present in the full *Daikon* output;
- **Relevant invariants #** at the weakening process: the total number of the previous detected invariants that passed the filtering process;
- **Unique relevant invariants #** at the weakening phase: the number of invariants if we count any group of logically equivalent invariants as one;
- **Valid fixes #** at the weakening phase: the number of valid pure-weakening fixes;
- **Unique valid fixes #** at the weakening phase: the number of unique fixes from those declared above;
- **Times run #**: the number of times the strengthening candidate generation algorithm was run, this is equal to the number of invalid weakening fixes plus one (a dummy one, used to

create pure-strengthening fixes), as the algorithm needs to be executed once for each one of those invalid fixes;

- **Invariants detected #** at strengthening phase: the number of invariants detected by *Daikon* for the strengthening process. In case the algorithm is run more than once, the average of the times is used;
- **Relevant invariants #** at the strengthening phase;
- **Unique relevant invariants #** at the strengthening phase;
- **Candidate combinations #**: the number of combination of candidates, with the average being used when running more than once;
- **Candidate fixes #**: the aggregated number of candidates from all runs of the strengthening generation phase;
- **Valid fixes #** at the strengthening phase: the number of valid fixes that use strengthening;
- **Unique valid fixes #** at the strengthening phase: the number of unique fixes from the ones above;
- **Valid fixes #** for all phases: sum of the number of pure-weakening fixes plus the ones that use strengthening;
- **Unique valid fixes #** for all phases: the number of unique fixes from the ones above.

Benchmarking against *SpeciFix* is unfortunately not possible, as we found no way to get its source code or access to the program, despite our efforts to contact the authors, who did not have it any more. The runtime of the different stages of the algorithm can be used to indicate that the solution is viable on its own. To proceed with the analysis of the runtime, we created six small faulty examples, namely:

- **DivRem**, composed by `divRem` running example that receives two integers and calculates their integer division and respective remainder, which is called by `rem` that discards the quotient and returns the remainder only;
- **Catalan numbers**, composed by a single recursive method that calculates the nth Catalan number. This code was adapted from exercises provided in classes at the University of Porto;
- **Harmonic Sum**, composed by `HarmonicSum`, a method that calculates the sum of the nth and $(n+1)^{\text{th}}$ harmonic terms, which are obtained by a call to `NthHarmonic`, a method that requires a positive number, wrongly disallowing its argument to be 0. This example was created by us;

- **Inverse Sign**, having a method `Enter` that receives an integer n and calls `div` which calculates $1/sgn(n)$ using a method `inverse` for the calculation of this division. This example was created by us;
- **Opaque Keyword**, having a single empty method with a post-condition that is only valid when its argument is in the interval $[0, 100)$. This example was adapted from *Dafny*'s repository tests;
- **Two Requires**, having a method `Enter` that calls an empty method with a strict precondition, allowing it to only be called with 0 as an argument. This example was adapted from *Dafny*'s repository tests.

Each example was executed 20 times on an *EndeavourOS* 2022.06.23 machine with 12 GiB of memory and a 2.1 GHz *AMD Ryzen* 5 4-core CPU. Each execution of the same example had the same arguments, which means there is no randomness, so all the runs generate the same values, but take different times to run.

6.2 Evaluation Results

All the information is displayed in Table 6.1, that includes the average runtime (in ms) for different stages of the program and some statistics related to the number of values generated by them. As indicated in the table, the initial stage of the strengthening phase is executed multiple times, once per invalid weakening candidate, but the final stage is executed once, using the aggregation and combination of the different strengthening assertions generated in the previous one. All the runtime data is averaged from the results of 20 runs, but the performance information about the part that is executed multiple times can be the average of more than 20 values, as the value represented

is the average of every single instance of the indicated step.

Table 6.1: Tool runtime and results statistics

	divRem	catalanNumber	harmonicSum	inverseSign	opaqueKeyword	twoRequires
Trace Generation						
Test execution	1407.30	1559.50	1088.00	1327.60	1453.00	1347.40
Tests #	225	15	15	15	300	15
Weakening						
Daikon input generation	22.10	64.25	5.95	4.60	16.45	3.45
Daikon execution	3988	3329	3034	3172	2925	2510
Daikon weakening %	98.2%	93.9%	99.4%	99.4%	98.8%	99.6%
Daikon output parsing	3.70	2.65	3.05	5.45	1.40	1.80
Invariants detected #	151	72	89	205	30	64
Invariant filtering	1.85	1.80	2.85	3.00	2.65	2.80
Relevant invariants #	3	3	3	3	1	1
Unique relevant invariants #	2	2	2	3	1	1
Candidate fix creation	0.50	0.40	0.45	0.45	0.40	0.45
Fix validation	43.80	147.55	6.50	6.80	15.45	1.90
Valid fixes #	3	0	3	0	1	1
Unique valid fixes #	2	0	2	0	1	1
Weakening generation	4059	3546	3053	3191	2960	2520
Strengthening (Multiple times per run, averaged)						
Times run #	1	4	1	4	1	1
Daikon input generation	23.55	73.83	2.85	1.88	13.65	2.00
Daikon execution	4033.85	3397.18	3027.95	3175.56	2935.45	2564.75
Daikon strengthening %	97.44%	96.56%	99.67%	99.60%	99.25%	99.82%
Daikon output parsing	4.05	1.84	2.05	3.94	0.65	1.00
Invariants detected #	163	72	94	175.75	30	57
Invariant filtering	0.05	0.01	0.05	0.01	0.05	0.20
Relevant invariants #	7	3	5	5.5	1	3
Unique relevant invariants #	4	2	3	3.75	1	2
Candidate fix creation	0.45	0.15	0.45	0.14	0.40	0.45
Candidate combinations #	7	3	5	7.5	1	3
Strengthening (Once per run)						
Candidate fixes #	7	12	5	30	1	3
Fix validation	77.90	180.75	4.70	26.60	7.30	1.05
Valid fixes #	2	8	2	18	0	2
Unique valid fixes #	1	1	1	3	0	1
Strengthening generation	4140	14073	3038	12753	2958	2569
All Phases						
Total runtime	9100	18681	6836	16980	7626	7431
Valid fixes #	5	8	5	18	1	3
Unique valid fixes #	3	1	3	3	1	2
Test Values	$[-5,10]^2$	[-5,10]	[-5,10]	[-5,10]	[-100,200)	[-5,10]

As noted in the table, the first stages of the strengthening algorithm can be executed more than once, which is indicated by the *Times Run #* row, as those stages need to be executed once for each invalid weakening candidate and for a dummy candidate. The times indicated under those

stages are averaged per execution, so to get the average time for each example we need to multiply the times by the number of *Times Run #*, which is taken into consideration when calculating the *Strengthening generation*.

All the calculations of the number of effectively unique invariants or fixes were done manually by comparing them, removing the repeated ones, i.e. the ones that had another invariant or fix as logically equivalent, and counting the remaining ones. These were added to the table to indicate the level of equivalence in those types of values.

The arguments that were used to generate the tests are indicated in the table, as *Test Values*. Ideally, they would be automatically generated, but this was not done, although it is something to be done in future work. We tried to get different sources, but the creation of a dataset of faulty programs is an important step, because it is critical for the evaluation of our tool and other tools that may need the same kind of programs.

The total runtimes to generate the fixes range from 6 to 19 seconds, which is an acceptable amount for this kind of tool, although the used examples are very simple and do not include some very common structures like classes. The majority of this time, perhaps as expected, is spent detecting invariants by running *Daikon*, which is followed by a wide margin by the time it takes to initially run the program to get the trace. So, it is reasonable to consider this a possible focus point in the future work, which can be improved by tweaking *Daikon* configurations or looking for alternative tools or ways to get the invariants.

6.3 Solutions Generated

In this section, we discuss the solutions that were generated by the tool, with the goal of having a first idea about their quality. But, to seriously evaluate the results, we would need a deeper analysis, taking place with, for example, more tests and, possibly, an investigation about how users would classify them.

The algorithm was able to generate a solution for each example, except the *Opaque Keyword* program, which depended on *Daikon* generating an invariant in the format $0 \leq x < 100$, but it only generated $0 \leq x$ despite our efforts in tweaking the configuration or changing test arguments. For the *divRem*, *Harmonic Sum* and *Two Requires* programs, we got the expected result of finding both weakening and strengthening results, and we also got the expected weakening results on the other examples.

A notable result can be found in the repair of the *divRem* program, where the found strengthening fix was stricter than what was expected and needed. *Daikon* was able to infer $n > d$ instead of $n > 0$, the expected one, for the pre-condition of the `rem` method, which paired with another inferred invariant, $d \geq 0$, is stronger than what was expected, and disallows valid calls for this method that do not result in any contract being broken, e.g. $d = 9 \wedge n = 3$.

The non-unique solutions are generated because of the equivalent expressions that are generated by the invariant detection tool. The existence of those expressions means extra time is spent

with fixes that are not needed in the first place, because an equivalent one has already been generated, and the algorithm runs some stages of the strengthening part once per invalid weakening solution, even if an equivalent one has already been processed. This also generates equivalent solutions, which are not good for the end user, as the selected ones to be shown can not be as representative of the possible solutions for a problem. So, another area that can be improved is the verification if we have equivalent invariants before these phases.

Chapter 7

Conclusion

This thesis presents a prototype tool that can generate contract fixes for faulty *Dafny* programs, assuming the correctness of the respective implementation.

We were inspired by a technique [75] previously proposed for *Eiffel*, another contract programming language, and currently rely on a third-party tool for dynamic invariant inference [27].

Our preliminary evaluation shows that the procedure is feasible in Dafny for our simple examples. It also shows that the steps performed by our tool take reasonable time to execute, but the overall process suffers from some efficacy and performance issues that stem from the invariant detection phase performed by *Daikon*. Although some fine-tuning of *Daikon*'s input parameters may improve its performance, scaling the procedure for more complex programs will likely require further research on this topic.

There is still substantial work to be done before the tool can be deployed in the development of Dafny programs. For example, language features such as mutable elements and classes have not yet been addressed. Also, the evaluation must be expanded to consider more complex and realistic programs. In addition, user-studies must be carried out to explore how helpful users find the generated fixes and if these fixes do contribute positively to produce a program that verifies.

We did not encounter any false positives, i.e. fixes that would pass our dynamic verification but be invalidated by *Dafny*'s validation. But techniques for preventing them may be considered when the tool becomes more complete, or may be desirable, as a configurable option, for users that can afford the computational cost of integrating our tool with *Dafny*'s verification, to statically check only the suggestions deemed more relevant before presenting them to the user.

Although previous work suggests that inferred contracts are important to complement contracts written by programmers [77], work on human factors that affect the use of inferred contracts is lacking. Further work in this area should be carried out, for example, to support programmers with determining the relevance and validity of the inferred invariants, as previous studies have shown that users have difficulty deciding if the invariants inferred by *Daikon* are true [83].

Moreover, techniques for automated test generation should be employed to create the test cases, and there is some work on this topic for Dafny [86][29]. To be useful for the community, such a repair technique should integrate the Dafny IDE, namely its plug-in for VS Code. Lastly,

as part of our future work, we aim to build a large dataset of faulty Dafny programs that will be made available to the community to foster further research in the area of Dafny program repair.

References

[1] Alexandre Abreu, Nuno Macedo, and Alexandra Mendes. Exploring automatic specification repair in Dafny programs. In *38th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, 2023.

[2] Rui Abreu, Alberto González, Peter Zoeteweij, and Arjan JC van Gemund. Automatic software fault localization using generic program invariants. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 712–717, 2008.

[3] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, et al. An orchestrated survey of methodologies for automated software test case generation. *Journal of systems and software*, 86(8):1978–2001, 2013.

[4] David Arthur. Evaluating Daikon and its applications. *URL: http://people. duke. edu/dga2/cps208/Daikon. pdf*, 2003.

[5] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[6] Thoms Ball. The concept of dynamic analysis. *ACM SIGSOFT Software Engineering Notes*, 24(6):216–234, 1999.

[7] 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 *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures 4*, pages 364–387. Springer, 2006.

[8] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525, 2014.

[9] Raymond T Boute. The Euclidean definition of the functions div and mod. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 14(2):127–144, 1992.

[10] Simón Gutiérrez Brida, Germán Regis, Guolong Zheng, Hamid Bagheri, ThanhVu Nguyen, Nazareno Aguirre, and Marcelo F. Frias. Bounded exhaustive search of Alloy specification repairs. In *43rd IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 1135–1147. IEEE, 2021.

[11] David Bingham Brown, Michael Vaughn, Ben Liblit, and Thomas Reps. The care and feeding of wild-caught mutants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 511–522, 2017.

[12] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[13] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. Exe: Automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):1–38, 2008.

[14] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NASA Formal Methods Symposium*, pages 3–11. Springer, 2015.

[15] Franck Cassez. Verification of the incremental Merkle tree algorithm with Dafny. In *24th International Symposium on Formal Methods (FM)*, volume 13047 of *LNCS*, pages 445–462. Springer, 2021.

[16] Jorge Cerqueira, Alcino Cunha, and Nuno Macedo. Timely specification repair for Alloy 6. In *20th International Conference on Software Engineering and Formal Methods (SEFM)*, volume 13550 of *LNCS*, pages 288–303. Springer, 2022.

[17] Liushan Chen, Yu Pei, and Carlo A Furia. Contract-based program repair without the contracts: An extended study. *IEEE Transactions on Software Engineering*, 47(12):2841–2857, 2020.

[18] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. *Acm Sigplan Notices*, 46(3):265–278, 2011.

[19] Ilinca Ciupa and Andreas Leitner. Automatic testing based on design by contract. In *Proceedings of Net. ObjectDays*, volume 2005, pages 545–557. Citeseer, 2005.

[20] Robert Clarisó and Jordi Cabot. Fixing defects in integrity constraints via constraint mutation. In *11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 74–82. IEEE Computer Society, 2018.

[21] Jake Cobb, James A Jones, Gregory M Kapfhammer, and Mary Jean Harrold. Dynamic invariant detection for relational databases. In *9th International Workshop on Dynamic Analysis (WODA@ISSTA)*, pages 12–17. ACM, 2011.

[22] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. Automatic inference of necessary preconditions. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 128–148. Springer, 2013.

[23] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. Dysy: Dynamic symbolic execution for invariant inference. In *Proceedings of the 30th international conference on Software engineering*, pages 281–290, 2008.

[24] Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electronic notes in theoretical computer science*, 217:5–21, 2008.

[25] Michael D Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, 2003.

[26] Michael D Ernst, Adam Czeisler, William G Griswold, and David Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd international conference on Software engineering*, pages 449–458, 2000.

[27] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69(1-3):35–45, 2007.

[28] Michael Dean Ernst. *Dynamically discovering likely program invariants*. University of Washington, 2000.

[29] Aleksandr Fedchin, Tyler Dean, Jeffrey S. Foster, Eric Mercer, Zvonimir Rakamaric, Giles Reger, Neha Rungta, Robin Salkeld, Lucas Wagner, and Cassidy Waldrip. A toolkit for automated testing of Dafny. In *15th International Symposium on NASA Formal Methods (NFM)*, volume 13903 of *LNCS*, pages 397–413. Springer, 2023.

[30] Viktoria Felmetsger, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Toward automated detection of logic vulnerabilities in web applications. In *19th USENIX Security Symposium (USENIX Security)*, pages 143–160. USENIX Association, 2010.

[31] George Fink and Matt Bishop. Property-based testing: a new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes*, 22(4):74–80, 1997.

[32] International Organization for Standardization. *Software and systems engineering — Software testing — Part 1: General concepts*. International Organization for Standardization, Vernier, Geneva, Switzerland, ISO/IEC/IEEE 29119-1:2022(en) edition, 2022.

[33] Carlo Alberto Furia and Bertrand Meyer. Inferring loop invariants using postconditions. *Fields of Logic and Computation: Essays Dedicated to Yuri Gurevich on the Occasion of His 70th Birthday*, pages 277–300, 2010.

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

[35] Patrice Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–54, 2007.

[36] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. Specification-based program repair using SAT. In *Tools and Algorithms for the Construction and Analysis of Systems: 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings 17*, pages 173–188. Springer, 2011.

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

[38] Stewart Grant, Hendrik Cech, and Ivan Beschastnikh. Inferring and asserting distributed system invariants. In *40th International Conference on Software Engineering (ICSE)*, pages 1149–1159. ACM, 2018.

[39] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the aaai conference on artificial intelligence*, volume 31, 2017.

[40] John Guttag. Abstract data types and the development of data structures. *Communications of the ACM*, 20(6):396–404, 1977.

[41] Sudheendra Hangal, Naveen Chandra, Sridhar Narayanan, and Sandeep Chakravorty. IODINE: A tool to automatically infer dynamic invariants for hardware designs. In *42nd Design Automation Conference (DAC)*, pages 775–778. ACM, 2005.

[42] Sudheendra Hangal and Monica S Lam. Tracking down software bugs using automatic anomaly detection. In *24th International Conference on Software Engineering (ICSE)*, pages 291–301. ACM, 2002.

[43] Michael Harder, Jeff Mellen, and Michael D Ernst. Improving test suites via operational abstraction. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 60–71. IEEE, 2003.

[44] Luke Herbert, K Rustan M Leino, and Jose Quaresma. Using Dafny, an automatic program verifier. In *LASER Summer School on Software Engineering*, pages 156–181. Springer, 2011.

[45] Hengle Jiang, Sebastian Elbaum, and Carrick Detweiler. Inferring and monitoring invariants in robotic systems. *Auton. Robots*, 41(4):1027–1046, 2017.

[46] Cem Kaner, Jack Falk, and Hung Q Nguyen. *Testing computer software*. John Wiley & Sons, 1999.

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

[48] Laura Kovács. Automated polynomial invariant generation by algebraic techniques for imperative program verification in Theorema. *Giese and Jebelan (2007) pp*, pages 56–69, 2007.

[49] Laura Kovács. Aligator: A Mathematica package for invariant generation (system description). In *International Joint Conference on Automated Reasoning*, pages 275–282. Springer, 2008.

[50] Laura Kovács. Reasoning algebraically about p-solvable loops. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 249–264. Springer, 2008.

[51] Laura Kovács and Andrei Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *International Conference on Fundamental Approaches to Software Engineering*, pages 470–485. Springer, 2009.

[52] Shriram Krishnamurthi and Tim Nelson. The human in formal methods. In *Formal Methods–The Next 30 Years: Third World Congress, FM 2019, Porto, Portugal, October 7–11, 2019, Proceedings 3*, pages 3–10. Springer, 2019.

[53] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992.

[54] K Rustan M Leino. This is Boogie 2. *manuscript KRML*, 178(131):9, 2008.

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

[56] K Rustan M Leino. Accessible software verification with dafny. *IEEE Software*, 34(6):94–97, 2017.

[57] Nancy Leveson. Are you sure your software will not kill anyone? *Communications of the ACM*, 63(2):25–28, 2020.

[58] Ye Liu and Yi Li. InvCon: A dynamic invariant detector for Ethereum smart contracts. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 160:1–160:4, 2022.

[59] Fan Long, Peter Amidon, and Martin Rinard. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 727–739, 2017.

[60] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 298–312, 2016.

[61] Jessica McBroom, Irena Koprinska, and Kalina Yacef. A survey of automated programming hint generation: The HINTS framework. *ACM Computing Surveys (CSUR)*, 54(8):1–27, 2021.

[62] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*, pages 691–701, 2016.

[63] B. Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992.

[64] Bertrand Meyer. Eiffel: programming for reusability and extendibility. *ACM Sigplan Notices*, 22(2):85–94, 1987.

[65] Bertrand Meyer. *Design by contract*. Prentice Hall Upper Saddle River, 2002.

[66] Markus Mock. Dynamic analysis from the bottom up. In *WODA 2003 ICSE Workshop on Dynamic Analysis*, page 13, 2003.

[67] Facundo Molina, Pablo Ponzio, Nazareno Aguirre, and Marcelo Frias. Evospex: An evolutionary algorithm for learning postconditions. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1223–1235. IEEE, 2021.

[68] Martin Monperrus. Automatic software repair: A bibliography. *ACM Comput. Surv.*, 51(1), jan 2018.

[69] Toh Ne Win and Michael D. Ernst. Verifying distributed algorithms via dynamic analysis and theorem proving. Technical Report 841, MIT Laboratory for Computer Science, Cambridge, MA, May 25, 2002.

[70] Charles Gregory Nelson. *Techniques for program verification*. Stanford University, 1980.

[71] Chris Newcombe. Why Amazon chose TLA+. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 25–39. Springer, 2014.

[72] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services uses formal methods. *Communications of the ACM*, 58(4):66–73, 2015.

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

[74] Jeremy W Nimmer and Michael D Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. *Electronic Notes in Theoretical Computer Science*, 55(2):255–276, 2001.

[75] Yu Pei, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. Automatic program repair by fixing contracts. In Stefania Gnesi and Arend Rensink, editors, *Fundamental Approaches to Software Engineering*, pages 246–260, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[76] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. *IEEE Transactions on Software Engineering*, 40(5):427–449, 2014.

[77] Nadia Polikarpova, Ilinca Ciupa, and Bertrand Meyer. A comparative study of programmer-written and automatically inferred contracts. In *18th International Symposium on Software Testing and Analysis (ISSTA)*, pages 93–104. ACM, 2009.

[78] Xiao Qu, Myra B Cohen, and Katherine M Woolf. Combinatorial interaction regression testing: A study of test case generation and prioritization. In *2007 IEEE International Conference on Software Maintenance*, pages 255–264. IEEE, 2007.

[79] Martin C Rinard. Survival techniques for computer programs. 2006.

[80] Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. Adapting proof automation to adapt proofs. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 115–129, 2018.

[81] Neha Rungta. A billion smt queries a day. In *International Conference on Computer Aided Verification*, pages 3–18. Springer, 2022.

[82] Todd W Schiller, Kellen Donohue, Forrest Coward, and Michael D Ernst. Case studies and tools for contract specifications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 596–607, 2014.

[83] Todd W Schiller and Michael D Ernst. Reducing the barriers to writing verified specifications. In *27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 95–112. ACM, 2012.

[84] Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools: (tool paper). In *Computer Aided Verification: 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006. Proceedings 18*, pages 419–423. Springer, 2006.

[85] Kavir Shrestha and Matthew J. Rutherford. An empirical evaluation of assertions as oracles. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 110–119, 2011.

[86] Patrick Spettel. Delfy: Dynamic test generation for Dafny. Master's thesis, Eidgenössische Technische Hochschule Zürich, 2013.

[87] Michael Stueben and Michael Stueben. Defensive programming. *Good Habits for Great Coding: Improving Programming Skills with Examples in Python*, pages 123–126, 2018.

[88] John Symons and Jack K. Horner. Why there is no general solution to the problem of software verification. *Foundations of Science*, 25(3):541–557, Sep 2020.

[89] Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. Automated model repair for Alloy. In *33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, pages 577–588. ACM, 2018.

[90] Yi Wei, Carlo A. Furia, Nikolay Kazmin, and Bertrand Meyer. Inferring better contracts. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 191–200, New York, NY, USA, 2011. Association for Computing Machinery.

[91] Stephen Wolfram. *The MATHEMATICA® book, version 4*. Cambridge university press, 1999.

[92] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[93] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM computing surveys (CSUR)*, 41(4):1–36, 2009.

[94] Lingming Zhang, Guowei Yang, Neha Rungta, Suzette Person, and Sarfraz Khurshid. Feedback-driven dynamic invariant discovery. In *2014 International Symposium on Software Testing and Analysis (ISSTA)*, pages 362–372. ACM, 2014.

[95] Guolong Zheng, ThanhVu Nguyen, Simón Gutiérrez Brida, Germán Regis, Nazareno Aguirre, Marcelo F. Frias, and Hamid Bagheri. ATR: Template-based repair for Alloy specifications. In *31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 666–677. ACM, 2022.

[96] Michael Zhivich and Robert K Cunningham. The real cost of software errors. *IEEE Security & Privacy*, 7(2):87–90, 2009.

Appendix A

Example Source-Codes

A.1 Programs Used for the Evaluation

The outermost routine is named `Enter`, unless expressly stated otherwise.

A.1.1 divRem

The outermost routine is `rem`.

```
function abs(n : int) : int {
 if n < 0 then -n else n
}
method divRem(d : int, n : int)
 returns (q : int, r : int)
 requires d >= 0
 requires n > 0
 ensures r + q * n == d
{
 if n == 0 {
  return 1 / 0, 0;
 }
 r := d;
 var m := abs(n);
 q := 0;
 while r >= m
 invariant r + q * m == d
 invariant q >= 0
 invariant m == abs(n)
 {
 q := q + 1;
 r := r - m;
 }
 if n < 0 {
 q := -q;
 }
}
```

```
method rem(d : int, n : int)
 returns (r : int)
 requires d >= 0
 requires n != 0
{
 var s_;
 s_, r := divRem(d, n);
}
```

And, just as an example, it is possible to see the output generated by the tool for this example at Listing A.1.

```
Weakening

Change PRE_CONDITION of divRem to (true && d >= 0 && n > 0) || true

Change PRE_CONDITION of divRem to (true && d >= 0 && n > 0) || n != 0

Change PRE_CONDITION of divRem to (true && d >= 0 && n > 0) || !(n == 0)

Strengthening

Change PRE_CONDITION of divRem to (true && d >= 0 && n > 0) || false
Change PRE_CONDITION of rem to true && d >= 0 && n != 0 && d < n

Change PRE_CONDITION of divRem to (true && d >= 0 && n > 0) || false
Change PRE_CONDITION of rem to true && d >= 0 && n != 0 && !(d >= n)
```

Listing A.1: Output generated by the tool when running the `divRem` example.

A.1.2 Catalan Number

Translated from an *Eiffel* implementation of the same algorithm [1]. The outermost routine is `NthCatalanNumber`.

```
method NthCatalanNumber(n : int) returns (r : real)
  // requires n >= 0
{
  if n == 0 {
    r := 1.0;
  } else {
    var t := (4 * n - 2) as real;
    var s := (n + 1) as real;
    var p := NthCatalanNumber(n - 1);
    r := t / s * p;
  }
}
```

[1] https://rosettacode.org/wiki/Catalan_numbers#Eiffel

A.1.3 Harmonic Sum

```
method NthHarmonicTerm(x : int) returns (c : int)
  // Calculates the nth harmonic term, equivalent
  // to the reciprocal of 1 / (x + 1)
  requires x >= 1
{
  if x < 0 {
    return 1 / 0;
  }
  return 1 / (x + 1);
}

method Enter(n : int) returns (sum : int)
  requires n >= 0
{
  var i := 0;
  sum := 0;
  while i < n {
    var ith := NthHarmonicTerm(i);
    sum := sum + ith;
    i := i + 1;
  }
}
```

A.1.4 Inverse Sign

```
method inverse(x : int) returns (i : int)
  requires x != 0
{
  i := 1 / x;
}

function sign(x : int) : int
{
  if x == 0
  then 0
  else if x > 0
  then 1
  else -1
}

method div(x : int, y : int) returns (d : int)
  // requires y != 0
{
  var s := sign(y);
  var inv := inverse(s);
  d := x * inv;
}
```

```

method Enter(x : int)
{
  var r := div(1, x);
}
```

A.1.5 Opaque Keyword

Adapted from the *Dafny* repository tests [2]

```
ghost predicate {:opaque} F(n: int)
{
  0 <= n < 100
}
method Enter(x : int)
  ensures F(x)
{
}
```

A.1.6 Two Requires

Adapted from the *Dafny* repository tests [3].

```
method A(x: int)
  requires x >= 0
  requires x <= 0
  ensures true
{}

method Enter(x: int) {
  A(x);
}
```

A.2 Other Examples

A.2.1 Circular List

```
type T = int

class {:autocontracts} Circular {
  var list: array<T>;
  var count: int;
  var cursor: int;

```

[2] https://github.com/dafny-lang/dafny/blob/edb901dbb6d984601ba6a3dc657e932995b33f48/Test/dafny0/OpaqueFunctions.dfy#L173

[3] https://github.com/dafny-lang/dafny/blob/edb901dbb6d984601ba6a3dc657e932995b33f48/Test/dafny0/one-message-per-failed-precondition.dfy#L7

```
  constructor (size: int)
    requires size >= 1
    ensures list.Length == size
  {
    list := new T[size];
    count := 0;
    cursor := 0;
  }

  predicate Valid {
    && 0 <= count <= list.Length
    && 0 <= cursor <= count
  }

  method Duplicate(n: int) returns (result: Circular)
  {
    var i := 0;
    var size := Math.Min(n, count);
    result := new Circular(count);
    while i < size
      modifies  result.list
    {
      var index := (cursor + i) % count;
      result.list[i] := list[index];
      i := i + 1;
    }
    result.count := size;
  }
}
```

A.2.2 Hanoi Tower

```
method move(n: int, from: int, dest: int, via: int)
  decreases n
  // requires n > 0
{
  if n != 1 {
    move(n - 1, from, via, dest);
    move(1, from, dest, via);
    move(n - 1, via, dest, from);
  }
}
```