# Automated Program Repair of Dafny Programs

Repairing Arithmetic Programs

**Hugo Rafael Fecha Martins**

Thesis to obtain the Master of Science Degree in

## Computer Science and Engineering

Supervisors: Prof. João Fernando Peixoto Ferreira
Prof. Alexandra Mendes

## Examination Committee

Chairperson: Prof. Pedro Miguel dos Santos Alves Madeira Adão
Supervisor: Prof. João Fernando Peixoto Ferreira
Member of the Committee: Prof. Rui Filipe Lima Maranhão de Abreu

**November 2022**

This work was created using LaTeX typesetting language
in the Overleaf environment (www.overleaf.com).

# Acknowledgments

Firstly, I would like to thank my supervisors Prof. João Ferreira and Prof. Alexandra Mendes for their support, dedication and accessibility. From the start, it has been a pleasure working with them, from always providing academic and technical support whenever needed to worries about personal state during the development of the projects, I could not have chosen better supervisors for this work.

Secondly, I would like to thank my parents for everything they've done throughout the years, for their love and care, and the tireless support from them. Without them, I would have never reached the state where I am able to do this kind of work, work that I love.

Thank you to my brother, for all the times he was waiting for me when I came back home from university, for all the moments we shared together, and for being the best brother I could ever have.

I would also thank my girlfriend for all the good and tough times, for all the love and care, for all the company while I was doing university work, for all the times you had to listen to me explain technical terms without understanding them but never actually complaining about that, and for being supportive and kind.

Last but not least, to all my friends and colleagues that shaped me into who I am today and for being there in good and bad times. Thank you.

To each and every one of you – Thank you.

# Abstract

Writing software is hard. It would be amazing to be able to write programs without bugs, or at least write them and easily finding those bugs. Beyond finding those bugs, being able to receive suggestions on how to fix those bugs in useful time would be a tool that would greatly boost the productivity of any programmer.

In this thesis we document the process of creating a solution to repair arithmetic Dafny programs with bugs, using formal verification and expression templates.

To repair programs, we used an existing synthesizer, as well as an existing verifier, and in the process created a framework that allows integration with any other program generator.

The proposed solution identifies suspicious statements in the input program with the help of the Dafny verifier, traces the program execution to determine the point where the program failed, translates to the language of the synthesizer (in case it doesn't support Dafny *out-of-the-box*, in our case, Suslik's SSL) and suggests a correction for the bug that was found.

# Keywords

Automated Program repair, Deductive Synthesis, Dafny, Formal Verification, Constraint Solving, Program Synthesis, Dynamic Framing

# Resumo

Escrever software é difícil. Seria incrível ser capaz de escrever programas sem bugs, ou pelo menos escrever software, e ter a possibilidade de encontrar os bugs sem perder horas em debugging. Para além de facilmente encontrar os bugs, receber sugestões de como os corrigir em tempo útil seria uma ferramenta excelente para propulsionar o desempenho de qualquer programador.

Nesta tese documentamos o processo que utilizá-mos para criar uma *framework* para reparar programas de aritmética com bugs em Dafny.

Para reparar programas, utilizámos um sintetizador já existente, assim como um verificador, para reparar programas Dafny com bugs. No processo, criámos uma framework extensível que permite adicionar integração com outros geradores.

A solução proposta identifica expressões de código problemáticas recorrendo ao verificador da linguagem Dafny, cria os *Expression Templates*, traduz o estado actual do programa para a linguagem do sintetizador (caso a sua linguagem nativa não seja Dafny), e sugere uma correção para um bug que foi encontrado.

# Palavras Chave

Reparação Automática de Programas, Síntese Dedutiva, Dafny, Verificação Formal, Resolução de Restrções, Síntese de Programas, Dynamic Framing

# Contents


**1 Introduction** **1**

1.1 Work Objectives . . . 4

1.1.1 Contributions . . . 4

1.1.2 Motivational Examples . . . 5

1.1.2.A Sum program and verification . . . 5

1.1.2.B Value program . . . 7

1.2 Document Structure . . . 8

**2 Background** **9**

2.1 Hoare Logic . . . 11

2.1.1 Specification Language . . . 12

2.2 Dafny . . . 13

2.3 The Frame Problem and Dynamic Frames . . . 15

2.4 Program Synthesis . . . 17

**3 Related Work** **21**

3.1 Suslik . . . 23

3.2 Jennisys . . . 24

3.3 Automatic Program Repair Using Formal Verification (MAPLE) . . . 27

3.3.1 Verifying Programs . . . 27

3.3.1.A Bug Localization . . . 27

3.3.2 Template Patch Creation and Analysis . . . 28

3.3.3 Solving Constraints to Discover Repaired Programs . . . 28

3.4 Systematic method to deduce and synthesize Dafny programs . . . 29

**4 Solution Development** **31**

4.1 Architecture Overview . . . 33

4.2 Dafny-Boogie Interaction . . . 33

4.3 Translation Process . . . 34

4.3.1 Module Architecture . . . 34

4.3.2 Errors trace structure . . . . . 36
4.3.3 Suslik SSL Translation . . . . . 38
4.3.3.A Base Implementation . . . . . 38
4.3.3.B Extending the base implementation to support branching . . . . . 40
4.3.4 Connection to the Synthesizer . . . . . 41

**5 Evaluation** **43**
5.1 Initial considerations . . . . . 45
5.2 Functional Example . . . . . 45
5.3 Functional Example of the Extension . . . . . 46
5.4 Program Benchmark . . . . . 47

**6 Conclusion** **49**
6.1 Conclusions . . . . . 51
6.2 Future work . . . . . 51

**Bibliography** **53**

# List of Figures


2.1 Specification Language . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
2.2 Hoare inference rules . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

3.1 swap method implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

4.1 Solution Architecture Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
4.2 TranslationFactory Pattern applied . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.3 CounterExample structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37

5.1 Solution Integration Script . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
5.2 Output of running test 9.dfy . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47

# List of Tables

5.1 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48

# Listings

1.1 Sum written in Dafny with a bug introduced . . . . . 5
1.2 Verification of the program . . . . . 6
1.3 Sum written in Dafny . . . . . 7
1.4 Value method implementation . . . . . 7
1.5 Value method verification . . . . . 8
2.1 Function defined in Dafny . . . . . 14
2.2 Loop invariants . . . . . 15
2.3 Definition of Node Class . . . . . 16
2.4 Framing Problem . . . . . 16
2.5 Framing Problem . . . . . 17
3.1 Dafny assume and false derivation example . . . . . 26
4.1 GetTranslation implementation . . . . . 35
4.2 Error generated by our example program . . . . . 37
4.3 Translate psudo-code . . . . . 38
4.4 Intermediary program in SSL logic . . . . . 40
4.5 Translate extension to support branching . . . . . 40
5.1 Value method . . . . . 45
5.2 Example method with branching represented in the benchmark by test 9.dfy . . . . . 46
5.3 Example of generated intermediary program for example 9.dfy . . . . . 47

# 1

# Introduction

## Contents

1.1 Work Objectives . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
1.2 Document Structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

Writing Software is hard. Writing software without bugs is even harder. It is estimated that around 15-50 bugs are introduced per 1000 lines of code [1]. The fault of error introduction in software artifacts can be mainly attributed to human causes.

The process of repairing human-introduced errors is not only very tedious for the programmers, but also very expensive. To reduce this debugging cost, several researchers have been making developments in the field of Automated Program Repair (APR): the repair of programs without user intervention.

Due to the complex nature of this relatively new procedure, the interest in this subject has been gradually increasing, with a lot of investigation being developed on this area, with several approaches being proposed, most of which depending on techniques such as mutation testing [2, 3] and deep learning [4–6].

Test-suite [3] approaches are currently the common ground used to localize bugs, and to generate and validate patches, but they have one major flaw: since the program uses test-cases to find the bugs, the patch found is limited by them, which may sometimes result in repaired programs that break in unexpected cases. Another important approach taken by many researchers is mutation-based patch generation [2, 3], which uses genetic programming operators such as *mutate*, *insert*, *delete*, to generate mutated programs, which are then validated by a verifier. These approaches work for simpler programs, but they ultimately fail at repairing some non-trivial bugs.

Since some of these processes have limitations, some researchers turned to formal specification to guide the repair process [7–9]. The correctness of a program is ensured by using several logical formulas like pre- and post-conditions, assertions and invariants. The approaches based on formal specification have some advantages over test-based approaches since they provide better case coverage, leading to programs that are less prone to fail in edge cases.

In this work, we aim to use a formal-specification-based approach to fix programs written in Dafny [10]. To the best of our knowledge, our solution is the first APR solution that can repair Dafny programs. We will focus initially on a simpler fragment of the Dafny language, with the goal of extending it later to support dynamic memory.

The language Dafny was chosen since it has significant adoption in industry, with several companies using it daily to develop highly-reliable software (e.g. Amazon Web Services), and it can be compiled to several other languages, broadening our field of impact even further.

The general solution we propose is a multi-step approach that can be summed up as:

1. The verifier is invoked to verify if the program corresponds to its specification. If the verification is successful, there is nothing to repair.

2. If the program does not match its specification, the output of the verifier is collected and analyzed. This context will then be used to locate the statements where the verification fails. For the scope of this project we will only focus on simple arithmetic programs, due to the complexity of the

implementation.

3. From the artifacts collected by analyzing the context provided by the verifier, we can find which statement is problematic. The purpose of this is to introduce a *Template Patch*, an artifact that will be used to generate functional code to replace the buggy statement, with its main characteristic being that it has no body, and its purpose being playing a key role in the generation of the new program that will fix the bug found.

4. The context of the program is then analyzed, with the template patch pre and post-conditions being generated according to a set of defined heuristics. This is the translation part, where we analyze the context of the error trace, and build the pre-conditions based on every defined statement previous to the bug, and the post-condition from the post-condition of the original method.

5. The translated Template Patch created in the previous step will be used by a program synthesizer to generate appropriate pieces of code, and thus by replacing the template patch with the generated code we will get the repaired program.

This approach is, although not entirely similar, based on the work developed by Nguyen et al. [9], with the key difference being the language and context in which it was implemented. Instead of attempting to repair faulty C programs, in this project the target is the automated repair of Dafny programs. We will also be using the Dafny Verifier instead of a verifier developed in the context of the project, as well as making use of an existing synthesizer, namely the Suslik Synthesizer by Polikarpova et al. [11]

## 1.1 Work Objectives

The main goal of this project is to explore techniques for automatic repair of programs using formal specification and expression templates. More specifically, the objectives are to:

1. Explore APR techniques to repair programs in Dafny.

2. Test the usage of existing synthesizers, like Jennysis [12] to support the synthesis of **Template patches**.

3. Repair examples of buggy programs in Dafny. We intend to use a small benchmark of synthetic buggy examples, that despite being artificial, will establish a basis to be able to create future tests.

### 1.1.1 Contributions

With this thesis we present an implementation of automated program repair that was developed as the first solution to automatically correct buggy programs in Dafny.

With this implementation we created a framework that can be extended to support other synthesizers. Along with that, a benchmark was established to compare implementations.

### 1.1.2 Motivational Examples

In this section, we present the motivating examples we will be using throughout the thesis. We will be considering two programs: the first one being `sum` based on a single method that computes the sum of all numbers from 0 up to a given input number $n$, and the second one being a simple program that returns the value of a variable. Without getting into much detail, since we will look further into the specification of Dafny programs in Section 2.2, we will now go over what the program does, and its implementation.

#### 1.1.2.A Sum program and verification

Firstly, let us look at the method `sum`'s header: it has as input an integer **n**, and the defined return is a variable of integer type called **totalSum**.

The function is defined with resort to recursion (line 7), with the base case being that when $n$ is equal to 0, the sum of that element is 0 (line 5). After that, when the variable **n** is bigger than 0 (line 6), it will calculate the sum of all natural numbers from 0 to `n-1` (lines 7 and 8). After calculating the sum of all the elements, the result gets assigned to the `totalSum` variable, and this is where in Listing 1.1 we have a bug introduced: instead of summing the result to `n`, we are summing and multiplying the value of `n` by 2.

**Listing 1.1:** Sum written in Dafny with a bug introduced

```
1  method sum(n: int) returns (totalSum: int)
2      requires n >= 0;
3      ensures totalSum == n * (n + 1) / 2;
4  {
5      if (n == 0){totalSum := 0;}
6      else{
7          var subsum: int := sum(n - 1);
8          totalSum := subsum + 2 * n;
9      }
10 }
```

The program's specification has a pre- and a post-condition, and although these terms will be defined in Section 2.1 in further detail, they essentially mean that when the method call begins it *requires* that n is bigger or equal to 0, and that when the program exits (and thus, returns) the variable to be returned (totalSum) will be equal to $n * (n + 1)/2$.

This program is easily verified by Dafny, which has no problem detecting the bug in line 8. This is detected because the post-condition cannot be met.

After the program is verified like in the Listing 1.2, and since in line 18 the post-condition can not be proven from the state presented, the bug is identified in line 17 (line 7 of the original program). The statement is then replaced by a template patch, that in the context of our project corresponds to the last statement. At this point, it is possible to know the final program state that needs to prove the postcondition, in the form $n >= 0 \wedge n! = 0 \wedge s = (n-1) * n/2 \wedge res = f(s, n) \vdash res = n * (n+1)/2$. It is also possible to know the pre-state, defined in line 19.

**Listing 1.2:** Verification of the program

```
1  method sum(n: int) returns (totalSum: int)
2      requires n >= 0;
3      ensures totalSum == n*(n+1)/2;
4  {
5      // n >= 0 (From the pre-condition)
6      if (n == 0){
7          // n>=0 && n = 0
8          totalSum := 0;
9          // n >= 0 && n = 0 && totalSum = 0 (Final program state)
10         // Needs to prove the postcondition
11     }
12     else{
13         // n >= 0 && n != 0
14         var subsum: int := sum (n - 1);
15         // n >= 0 && n != 0 && subsum = (n-1)*n/2
16         //
17         // (From the post-condition of the function call)
18         totalSum := subsum + n;
19         // n >= 0 && n != 0 && subsum = (n-1)*n/2 &&
20         // totalSum = 2 * n + subsum
21         //
22         // Needs to prove the post-condition
23     }
24 }
```

After the generation of the proof obligations, they are then translated to the language of the synthesizer, which then proceeds to generate a program that meets the specification. After this process is done, a suitable body for the template patch is generated, and the program proceeds to replace the line

8 in Listing 1.3 with totalSum := subsum + n, and by fixing the assignment in line 8, like shown in Listing 1.3, the Dafny verifier successfuly verifies the program with 0 errors.

**Listing 1.3:** Sum written in Dafny

```
1  method sum(n: int) returns (totalSum: int)
2      requires n > = 0;
3      ensures totalSum = = n * (n + 1) / 2;
4  {
5      if (n = = 0){totalSum := 0;}
6      else{
7          var subsum: int := sum (n - 1);
8          totalSum := subsum + n;
9      }
10 }
```

#### 1.1.2.B Value program

Now, let us look at the program value present in Listing 1.4. As we can see according to the method post-condition, the method simply returns the value of the variable i in variable j. From the body of the program we can see there is a bug: instead of doing an attribution with `i`, we are doing it with `2`.

**Listing 1.4:** Value method implementation

```
1  method value(i: int) returns (j: int)
2      requires i >= 0
3      ensures j == i
4  {
5      j := 2;
6  }
```

This is verified by Dafny as having one error, and the verification can be done like in Listing 1.5, where we can see that the state of the program in line 7 does not imply the post-condition of the method.

Listing 1.5: Value method verification

```
method value(i: int) returns (j: int)
    requires i >= 0
    ensures j == i
{
    // i >= 0 (From the pre-condition)
    j := 2;
    // i >= 0 && j := 2
    //
    // Needs to prove the post-condition
}
```

This example will be used in cases where the complexity of the example Sum in Section 1.1.2.A is too complex to describe the entire process.

## 1.2 Document Structure

This thesis is divided as follows: firstly, we go over the background of the project in Chapter 2, where we review the documents and information on which this thesis stands. In this section we make a small introduction to Hoare logic in Section 2.1, to the Dafny language and verifier in Section 2.2, to the Frame Problem and to how Dafny manages to solve it in Section 2.3, and finally we go over program synthesis and the dimensions of it in Section2.4.

Secondly, after the background is reviewed, we introduce several contributions made by some researchers in the related areas in Chapter 3, such as Suslik 3.1, Jennisys 3.2, and an approach to Automated Program Repair by using template patches called MAPLE 3.3.

Then we get to the solution in Section 4, where we introduce our method of detecting bugs, tracing the program state to the bug found, creating the constraints for the template patch and using a synthesizer to generate the code to place in the template patch's place.

After we describe the solution, we present the results we obtained by testing our approach in Chapter 5, we create the benchmark with repair times and a small test description, and we interpret the results we obtained.

Lastly, we take some conclusions on Chapter 6 and present future work that can be developed to rectify shortcomings of our project.

# 2

# Background

Contents

2.1 Hoare Logic . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
2.2 Dafny . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
2.3 The Frame Problem and Dynamic Frames . . . . . . . . . . . . . . . . . . . . . . . . . 15
2.4 Program Synthesis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

In this section, we will go over the background of the project. Initially we will go over the basics of Hoare Logic which the whole process stands on. Secondly, we will define a Specification Language that will be used in examples, which is in fact a subset of the language Dafny, which will be introduced in third place.

After introducing Dafny, we will go over the Frame Problem and Dafny's approach to solving it (Dynamic Frames).

Lastly, we will introduce the basics on Program synthesis, and look at an example of a proposed solution for generation of Dafny programs that is bleeding edge in the field.

By the end of this section we should have all the background information we need to fully grasp the problem, and the foundations of the solution.

## 2.1 Hoare Logic

In this section, we will go over some of the basics in Hoare Logic.

Hoare logic is a system to provide a formal way to reason about program correctness. The core concept behind Hoare's logic is specification as a contract. The specification is provided by the programmer while the body of the program is irrelevant to the client: provided that the client meets the requirements, the output will meet the guarantees.

These requirements and guarantees are precisely pre- and post-conditions, where pre-conditions are predicates that define conditions that the input must fulfil for the program to provide correct function, provided that post-conditions describe the conditions in which the output will be provided, if the pre-conditions are met. The objective of the pre- and post-condition is that the client can trust the results obtained from the program call.

In Hoare Logic a program is *partially correct* with respect to its specification if before executing the program the pre-condition is met, if the program terminates the post-condition is true. A program is *correct* with respect to its specification if besides meeting the pre-condition before executing, the program terminates and the post-condition is correct.

Hoare Logic uses Hoare triples to reason about program correctness, taking the form $\{P\}\ C\ \{Q\}$, where $P$ represents the pre-conditions, $Q$ represents the post-conditions, and $C$ represents the statements that implement the function. The meaning of a triple $\{P\}\ C\ \{Q\}$ is that if we start the program $C$ with $P$ being true, the program will terminate in a state where $Q$ is true.

Consider the following Hoare triple $\{x = 0\}\ x := x + 5\ \{x > 0\}$. In the specification it is stated that x must be equal to 0, for the program to give us a bigger value than 0 on x. This triple is clearly correct since $0 + 5$ will be assigned to x, which in turn will be bigger than 0, and thus fulfilling the post-condition.

$$e ::= c \mid x \mid res \mid -e \mid e_1 + e_2 \mid e_2 - e_2 \mid e_1 \cdot e_2 \mid e_1 / e_2$$

$$P ::= true \mid false \mid e_1 = e_2 \mid e_1 \neq e_2 \mid e_1 > e_2 \mid e_2 \geq e_2 \mid e_1 < e_2 \mid e_2 \leq e_2$$
$$\mid \neg P \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid P_1 \longrightarrow P_2 \mid \forall x.P \mid \exists x.P$$
$$S ::= requires\ P_1\ ensures\ P_2 \mid invariant\ P$$

**Figure 2.1:** Specification Language

$$\frac{\{P\}\mathrm{C}_1\{Q\} \quad \{Q\}\mathrm{C}_2\{R\}}{\{P\}\mathrm{C}_1;\mathrm{C}_2\{R\}}\ \text{seq} \quad \frac{\{I \wedge B\}C\{I\}}{\{I\}\ \text{while}\ B\ C\{I \wedge \neg B\}}\ \text{while}$$

**Figure 2.2:** Hoare inference rules

### 2.1.1 Specification Language

In this section we present the specification language used in the definition of Hoare's Logic. It is presented in Fig. 2.1

In our specification language *c*, *x* and *res* denote respectively an integer constant, a variable, and res is the special variable used to define the output of the program. An expression *e* can be defined either by one of the previously mentioned operators, or a arithmetic operation combining two of them: subtraction, addition, multiplication and division. In this language, *P* denotes a first-order logical formula, which is composed of equality and arithmetic constraints, combined with logical connectives and quantifiers. And lastly *S* denotes a specification that can be either a pair of pre- and post-conditions of a program (*requires* and *ensures* keywords) or an invariant that is used in a loop statement.

We use Hoare Logic [13] to verify a program against its specification. The Hoare triple $\{P\}\ C\ \{Q\}$ is composed of two assertions, $P$ and $Q$ that represent the pre- and post-conditions of the program $C$, which in turn states that for a given program state that satisfies $P$ if program $C$ is executed and has termination, the new program state will satisfy $Q$. In Hoare logic, this means that the program is correct - the pre-condition is satisfied, the program terminates and it will match the post-condition.

To reason about program correctness, Hoare logic defines inference rules and axioms for almost all of the constructs of a simple imperative programming language. In addition to this, many rules have been extended to other fields of application, and even to support multiple other contexts, like concurrency, pointers and jumps. In this paper inference rules include rules handling conditional branching, assignment, function calls, etc.

In Fig. 2.2 there are two inference rules represented: the first one being the sequential rule or composition, while the second one refers to the while (loop) rule. These are only two examples, and since these rules are widely used in the field of program verification, many works by other researchers often make use of them [14] .

## 2.2 Dafny

While traditionally the full verification of a program's functional correctness has been done by hand, with pen and paper, more recently the focus of some researchers has been to fully automate the process of verification. The main goal is to achieve the verification of a program without having interaction of the programmer, meaning that all the information that the program needs to be verified is already within the specification of the program itself.

Currently there are two techniques in which researchers have mainly been focusing on: *Verification Condition Generation* [15] and *Symbolic Execution* [16]. While the first one essentially transforms the program and its specification into a big formula, and afterwards using it to feed an *automated theorem prover*, which is responsible for doing the proofs. The second approach is an SE-based technique that executes all possible program paths of the program using *symbolic* values instead of actual values, and collecting logical information along the way, while calling the theorem prover every time by passing it the logical information it collected for each state.

While the technique plays a big part, the theorem prover is essentially the key component of the verifier. Currently the most popular automated theorem prover is *Satisfiability Modulo Theories*, such as Z3 [17] which is used by Dafny.

Dafny is a language and a verifier itself. The language Dafny is an imperative language, following a sequential flow and it supports generic classes and dynamic allocation, while allowing programmers to specify its programs with specifications that the client must use. The specifications include pre-, post- and framing conditions, as well as termination metrics. To better specify the programs, the language includes user-defined mathematical functions and ghost variables, which in turn allow for modular verification: the separate verification of all parts of a program implies the correctness of the whole system.

Dafny's program verifier works by translating the given Dafny program into the intermediate verification laguage *Boogie 2* [18] in a way such that the correctness of the *Boogie* program implies the correctness of the Dafny program. The semantics of Dafny are close to the ones of Boogie to ease the translation. The Boogie tool is then used to generate first-order verification condition, which are then passed to the theorem prover *Z3*.

As an imperative language, it offers several constructs any programmer is used to: branching statements (*if*, *else*), loops, functions, classes, etc. Although functions are offered by Dafny, there is one key difference from other languages: they define mathematical functions, which have no side-effects. Notice they do not have a contract (a set of pre-conditions and post-conditions) like in Listing 2.1

**Listing 2.1:** Function defined in Dafny

```
1 function abs(x: int): int
2 {
3     if x < 0 then -x else x
4 }
```

Apart from functions, Dafny also provides *methods*, which in contrast to functions do define contracts. Methods compute one or more values and they may change the program state. Differently from other imperative languages they can be defined **outside** classes. Since they define a contract, they have the *requires/ensures* syntax, having the *requires* clause to declare the pre-conditions and the *ensures* defining the post-conditions. Another key difference from functions is that the inputs are **immutable**, and the outputs are **named**. The methods follow the design-by-contract methodology, meaning that Dafny only has to reason about the body of the method it is verifying, instead of verifying every method the programmer is using. To achieve this, Dafny relies on the contract (specification) of a method, and thus relying on *opacity*. It is relevant to point out that Dafny can use functions in the specification of methods.

Dafny also lets us define local variables with the keyword *var*, which may or may not have type declarations, since it can infer their type in almost all situations.

Another key feature of Dafny is the usage of loop invariants to specify the behavior of loops. These loop invariants have the syntax represented in Section 2.2 in line 5, and they are used to ensure the condition represented holds upon entering the loop, and during its execution. Another feature we can see in the listing are *assertions* that are represented in line 9. Assertions can be introduced in any point, and the usage of an assertion means that the condition should always hold whenever it reaches that part of the code. The last feature we are able to see in Listing 2.2 is related with the usage of the *decreases* keyword. This keyword is used to help Dafny reason about termination, and although it is not strictly necessary every time, its function is to prove that the program does not run forever. There are two places where Dafny needs to prove termination: Loops and recursion, which both require either correct guess by the verifier, or explicit declaration.

Dafny also has support for Arrays and Quantifiers. The behaviour of Arrays closely follows the implementation in other mainstream languages, with the key difference being that accesses to the array must be proven to be within bounds. These are proved in verification time, while in runtime no checks happen. Quantifiers in Dafny most often take the form of a *forall* expressions or *exists*, which respectively represent a universal and an existential quantifier, and they are often used to reason about elements of an Array or a data structure.

**Listing 2.2:** Loop invariants

```
method m4(n: nat)
{
    var i := 0;
    while i < n
        invariant 0 <= i
        decreases i
    {
        i := i+1;
    }
    assert i==n;
}
```

And lastly, Dafny follows the paradigm of Object Oriented Programming (*OOP*), however some aspects of *OOP* are not supported yet: it does not provide definition of subclasses and their verification for example. Classes can have Class Invariants, which define properties that must hold at the entry and exit point of every method, for every instance of the class. They are often used to express properties about the consistency of the internal representation of an object, and they are typically transparent to the clients.

With the definition of classes, it is important to take a look at Ghost fields. They are defined with the modifier *ghost* and they are used only in verification time, so they are not present in compiled versions of the code. Ghost variables are helpful to model Dynamic Frames [19], Dafny's approach to solving the *Frame Problem* [20], which will be described in the next section.

## 2.3 The Frame Problem and Dynamic Frames

In this section we try to grasp the complexity of the *frame problem*, and take a look at Dafny's approach to solving it.

By looking at Listing 2.3 we can see the implementation of class Node. The Method setX will set the value of X to an arbitrary value v, and at the end of the execution the specification ensures that the value is correctly set.

**Listing 2.3:** Definition of Node Class

```
1  class Node {
2      var x: int;
3
4      constructor ()
5          ensures x == 0
6      {
7          var x := 0;
8          this.x := 0;
9      }
10
11     method setX(v: int)
12         ensures this.x == v
13     { this.x := v; }
14 }
```

By looking at the Node class definition in Listing 2.4 we can see it clearly fails the assertion, because the given contracts are too weak to prove the assertion. In fact, the verifier will not allow the program to compile with the following error: *Error: assignment may update an object not in the enclosing context's modifies clause*. If not for the verifier, there would still be a problem happening after executing the statement $c2.setX(10);$ in line 5, since the contract of *setX* allows it to change the state of the project arbitrarily, as long as it keeps the contract.

This is what is commonly referred to as the *Frame Problem*: *when formally reasoning about a change in a system, how can one specify that only certain parts of the state can be modified?*

**Listing 2.4:** Framing Problem

```
1  method Main()
2  {
3      var n1 := new Node();
4      n1.setX(5);
5
6      var n2 := new Node();
7      n2.setX(10);
8
9      assert n1.x == 5;
10 }
```

To prove the assertion, one needs to strengthen the Node method's contracts. To do this, Dafny (and also some other languages) introduce the concept of *Dynamic Frames*: introduction of footprints of methods and functions.

The footprint of a method is the set of all fields it is able to modify, while the footprint of a function is the set of all fields that the function is permitted to read. By expressing footprints in the abstract vocabulary we are able to express the absence of interference by proving that the footprint of a method is disjoint from the footprint of a pure function.

The footprint of a method is expressed with the usage of modifies: the set of fields that are included in the pre-state that the method is able to modify, or newly allocated objects. In opposition, the footprint of a function is expressed with the usage of reads: the function can only read fields present in the "reads" clause.

Going back to the contract of the method setX, if we analyze the output of the verifier we are able to realize what is missing: the declaration of what the method is able to modify. To do this, we need to change the contract to follow the So, with dynamic frames, the difference should be the one present in line 3 in the following listing:

**Listing 2.5:** Framing Problem

```
1  ...
2  method setX(v: int)
3      modifies this // KEY DIFFERENCE HERE
4      ensures this.x == v
5  { this.x := v; }
6  ...
```

By introducing the "modifies this" clause we are defining that the method can only change the current object, and nothing else in the environment can be changed.

## 2.4 Program Synthesis

Program synthesis is the task of automatically generating a program that satisfies a wanted behavior, usually expressed in a high-level specification, and it is commonly regarded as a *search problem* [21] for programs, since the desired behavior is to find a program that specifies a certain set of rules, in a finite number of possible programs.

The process of program synthesis can be divided into a three step process: firstly, the program specification is written, secondly the synthesizer tries to generate a fitting program with the specification by using deductive synthesis and constraint-solving algorithms, and lastly a program is generated (if this is possible).

The specifications which a program synthesizer takes as arguments can be of several types, from a full formal definition such as a first-order formula [22] to a natural language sentence [23].

According to *Gulwani et al.* [24], program synthesis can have three key dimensions: the technique used for expressing the user intent, the space of search over which the desired program will be searched, and the search technique used.

**First dimension** - *Describing the User intent*: This is according to the original author the most important dimension from the point-of-view of the user. It is responsible for establishing the desire of the user and passing it over to the synthesizer.

Like previously mentioned, the intent of the user can be expressed in different encodings, but for the sake of this project we will only be referring to one of the encodings: Logical Specifications. The restriction of encodings is due to the scope of the project, and the synthesizers used being based on this approach.

**Second dimension** - *Search Space*: This dimension concerns itself with the search space where the synthesizer will search for the desired program. The choice of the search space is usually done by the implementer of the synthesizer, but it can also be restricted by the user, to better meet the user requirements.

Although the search space is not the main concern of this project, we can just say that the synthesizer's we will be using use the space of Logical Representations. Please refer to [24] for more information.

**Third dimension** - *Search Technique*: Search techniques is concerned with the technique used by the synthesizer to find the desired program. Although these relate closely to the choice of program space and the form of input constraints, there is a vast field of possible techniques: from Genetic Programming [25] to Machine Learning Based Techniques [4].

In our case, the synthesizer's we will be using use Logical Reasoning Based Techniques, a class of techniques that reduces the synthesis problem to a SAT/SMT formula, and uses an already existing SAT/SMT solver to explore the search space.

These techniques usually involve two steps: *Constraint Generation* and *Constraint Solving*.

*Constraint Generation* is the process of generating a logical constraint to restrict the program upon its generation. In the case of our synthesizer, we are concerned with invariant-based generation, where the generation essentially bases itself on the generation of constraints that assert that the desired program should behave correctly on all inputs.

*Constraint Solving* on the other hand is responsible for solving the formulas generated in the step of *Constraint Generation*. The synthesis constraints resulting from the generation process are second-order logic formulas with universal quantification, so these cannot be solved with standard constraint

solvers. Instead, there are some techniques that can be applied to allow the usage of off-the-shelf constraint solvers:

1. Reducing Second-Order to First order on the unkowns: techniques such as boolean combination of arithmetic constraints [26] for example.
2. Universal Quantifier Elimination: *Farkas Lemma* [26] or cover algorithms [27] are common approaches to removing universal quantifiers.

# 3

# Related Work

## Contents

3.1 Suslik . . . . . . . . . . 23
3.2 Jennisys . . . . . . . . . . 24
3.3 Automatic Program Repair Using Formal Verification (MAPLE) . . . . . . . . . . 27
3.4 Systematic method to deduce and synthesize Dafny programs . . . . . . . . . . 29

In this section we will be visiting the related work related to the project. Firstly, we will go over Suslik [11], a deductive approach to program synthesis of imperative heap-manipulating programs. This paper introduces us to a tool for used for generation of heap-manipulating programs with pointer usage and introduces a new set of rules for reasoning about programs that access dynamic memory called *SSL*.

Secondly, we will shift our attention to Jennisys [12], a language that follows the object-oriented paradigm that by modelling a concrete data representation and introducing an abstract model can use automatic synthesis to produce executable code.

After that, we will look at the contribution by Wang et al. [28], a systematic method to deduce and synthesize the Dafny programs.

After presenting the generators, we will review the technique used by Nguyen et al. [9], which closely relates to ours, since it essentially uses the same template-patch-based technique.

## 3.1 Suslik

Suslik is a deductive program synthesizer that generates imperative programs with pointers, by using declarative specifications written in its own form of Separation Logic.

This work was developed by Polikarpova et al. [29], and the authors's approach is based on their own framework used to reason about separation logic called Synthetic Separation Logic (SSL).

Consider the implementation present in Fig. 3.1 of the procedure $swap(x, y)$, a program used for swapping the values stored in two distinct heap locations x and y. The pre- and post-conditions of the method can be expressed in Separation Logic (SL) - a Hoare-style program logic used to verify programs with pointers.

$$\{x \mapsto a * y \mapsto b\} \text{ void swap (loc x, loc y) } \{x \mapsto b * y \mapsto a\}$$

**Figure 3.1:** swap method implementation

This is a declarative specification, it describes what should be in the heap before and after the code is executed. It states that the program takes as two inputs *x* and *y* in form of pointers, where each of them points to separate locations, with the values a and b stored in the location, respectively.

The previous example illustrates Separation Logic perfectly: assertions that capture the program state, represented by a symbolic heap. Separation Logic is different from Dynamic Frames, because while Dynamic Frames tries to restrict what the method can access to only what is being used in its body, Separation Logic does not have the need to get the frame assertions specifically written, with the access being inferred from the assertions in the pre- and post-conditions.

The authors then proceed to introducing the rules that were implemented in the context of the ap-

proach that compose Synthetic Separation Logic, a framework used to reason about dynamic memory access in heap-manipulating programs. Some examples of rules include READ, WRITE, FRAME, EMP to handle reasoning about ghost variables and termination, ALLOC and FREE to handle allocation and freeing of dynamic memory, etc.

After defining the rules for SSL, the authors proceed to turning it from a declaratively defined inference system to an algorithm used for deriving provably correct imperative programs. The algorithm follows the idea of a standard goal-directed backtracking proof-search, and it works in a *depth-first* manner, starting from the initial synthesis goal and always extends the left-most *open* leaf (that is not a terminal application). The algorithm has multiple steps: initially, the algorithm starts by trying to apply all the rules to the top-level goal. If it succeeds, it collects the set of sub-derivations, and tried to process the set of sub-derivations, at which point the algorithm will either try to apply another SSL rule, or try to solve the sub-goals and apply a continuation. If it succeeds, a program will result from here. If in any step of the algorithm it fails apply every single rule possible, the synthesis for the current goal fails, and the algorithm backtracks. If by the end the algorithm could not find code that is suitable for the specification, the synthesis fails and the algorithm ends.

After introducing the main algorithm the authors also develop several optimizations and extensions to give the algorithm better coverage and efficiency such as *invertible rules*, *branch abduction* and defining *early failure rules*, which greatly improve the performance of the algorithm.

Suslik has proven to be efficient, with all of the 22 benchmarks defined by the authors being synthesized within 40 seconds.

From this work we took a lot of information: from the synthesis algorithm the authors used, to the syntax used to do the program generation. Although it is important to understand the synthesis algorithm, for our implementation we used Suslik as the main synthesizer, translating from Boogie to SSL specification, having the synthesis done by Suslik.

## 3.2 Jennisys

Jennisys [12] is based on the affirmation that the desired behavior of a program can be described using an abstract model, which can then be compiled into executable code by using synthesis.

According to the authors, most programming languages provide a mechanism to delineate between the public specification of a method, type or a module, and the private specification thereof. The public specification can consist of a behavioral contract, or just a simple type signature, letting users know the types of the parameters to be used. Jennisys takes this delineation further: dividing a program into three parts: *public interface*, *data structures* and *executable code*.

**Public interface** is the first component, and it is used to define an abstract model of the component,

defined with resort to mathematical structures like sets and sequences. It is also used to define the components operations and their behavioral effects. These are not compiled, the are only used during compilation time.

**Data structures** is the second component, it is concerned with describing the data structures that are used to represent a component in run-time. It declares fields that are part of each instance of the component, which other component instances are part of the representation (referred to by the authors as the *frame*, and also declares an invariant that constrains the concrete variables and the frame and couples them with the model variables in the public interface.

**Executable code** is the final part of Jennisys, it is responsible for the executable code that will implement the component operations. This is the most revolutionary feature about Jennisys: there are multiple ways that the programmer can utilize to produce the code: code synthesis, code-generation hints, program sketches and manual coding.

In the paper the authors propose a way to generate code from abstract variables, abstract code, concrete variables, and a coupling invariant (from the **public interface** and the **datamodel** of a component), with the process being able to generate loop-less programs, with branching, assignment and method calls.

The technique that the authors use to do synthesis is different from the ones mentioned previously: it uses the program verifier to obtain sample inputs/outputs that satisfy the given specifications, which are then extrapoled into code for all the input variables. The name given to the algorithm is *dynamic synthesis*, because it combines two approaches to the same problem: symbolic execution and concrete execution.

The authors define an algorithm that is used for systematic state exploration since the verifier - Dafny - only outputs a single valid input/output (pre- and post-state) pair, which is not enough for generating code. It uses systematic state exploration and program extrapolation from concrete instances to fix the previously mentioned issue. Due to the nature of Synthesis being undecidable, the algorithm does not always succeed, but when it does the program generated is provably correct.

By using the **assume** keyword, the Dafny verifier will assume both the pre-condition and the post-condition, and asking it to derive false like in Fig. 3.1, we know if there is or not a possible example where the constraints hold, and thus if it is possible to derive a program from there. If one counterexample is generated, the values for a pre- and a post-state can be directly extracted from it.

**Listing 3.1:** Dafny assume and false derivation example

```
1   ...
2  method Dupleton() modifies this; {
3      var x, y: int;
4      assume a != b && elems == {a, b} && Invariant();
5      assert false;}}
```

The example present in Listing 3.1 is part of the running example of the paper, and it shows us what was just mentioned: by assuming the pre- and post-state (line 4) and making a false assertion, we can ask Dafny to derive from there. If Dafny is able to succeed, the pre- and post-conditions are mutually inconsistent, so any attempt at generating suitable code for the specification will fail. Otherwise, a counterexample is returned, and it has the particularity of the pre- and post-conditions holding, so the values of the pre- and post-state can be directly extracted.

The problem with this approach is that using concrete values such as the ones we just mentioned are not likely to result in a correct program. So the authors try to generalize from a concrete instance and find symbolic assignments instead. While these are obviously more general than constants, such symbolic assignments can still be correct only for certain program scenarios. When that happens, a logical condition (guard) must be generated, to characterize these particular scenarios.

After generating the guard for the conditional statement, a new specification is created, by adding a negation of the inferred guard as a fresh pre-condition, and calling the synthesis continues recursively.

If a solution is found for which a guard was not needed, the solution is proven unconditionally correct for the portion of the program space. This solution can be attributed to the last branch (*else*) of an *if-then-elif-...-else* structure that the author's approach synthesizes.

The authors go into a lot of detail on the definition of their algorithm, and for the sake of this paper we will not be going over all the specification. Interested readers can refer to [12] to further inspect the algorithm used by the authors.

However, this approach has some limitations. By the way that Jennisys works, only requesting the declaration of the specifications and not having user input, it has some limitations such as only being able to synthesize a limited number of programs, namely constructors and certain read-only recursive methods.

In our approach we will try using Jennisys as a synthesizer, but we envision that it will be a bottleneck to our approach, and we are hoping to move beyond it, by using some key ideas os Jennisys and improving upon it.

## 3.3 Automatic Program Repair Using Formal Verification (MAPLE)

In this section we will go over the work developed by Nguyen et al. [9] in the field of Automated Program Repair. The authors proposed a novel automated approach to repairing programs using formal verification and expression templates.

The framework developed by the authors can be divided into 3 main steps: Localization, Patching and Verification. We will now generally describe these steps, and after we will get into more detail on these processes:

In the first step the input program is verified symbolically, using Hoare Logic, against its specification. If the verification step fails, the program passes on to the bug localization step.

In the second step, the bug localization step, the possible buggy statements are collected, and possible template patches are generated. Template Patches are, in the context of this work, linear expressions of the program's variables with unknown coefficients. Each of the template-patched programs are then verified to collect a set of proof obligations that will constrain the template patch.

In the third step the constraints obtained in the second step will be solved with resort to Farka's Lemma to discover the unknown coefficients of the template patch. If a solution is found, a candidate patch to repair the program is obtained. Then, the program with the candidate patch is validated against its original specification, to determine if the patch is correct. If the buggy statement cannot be repaired, the program will continue with another suspicious statement.

We will now describe the main components of the framework to verify and localize the bugs, create template patches, collecting constraints and solving them.

### 3.3.1 Verifying Programs

The authors use Hoare Logic to verify programs and localize possible buggy statements. This process has been exemplified in Section 2.1.

When a program is being analyzed symbolically against its specification, its pre-condition is assigned to the initial program state. This means that the state of the program obeys the pre-condition, and the pre-condition captures its specification accordingly. Then, the program state after each statement is executed is computed using Hoare rules. In the end, the post-state of the program has to imply the post-condition. If it doesn't, we know the program has bugs, and we proceed to the Bug localization step.

#### 3.3.1.A Bug Localization

After the program verification is finished, invalid proof obligations are identified to discover the path of the buggy execution. In the author's implementation, the correspondence of the constraints in each

proof obligation is recorded with the program specification and code. Using this record, the antecedent of the invalid proof obligation can be simplified by removing all constraints that belong to the program specification, and the remaining are the ones that correspond to the code causing the bug.

After all the reasoning is done, and to make the localization more efficient, the remaining constraints are ranked by how suspicious they are: how likely they are to cause the bug. The heuristics are:

1. If a constraint has its corresponding program code in the correct execution path, the constraint is less likely to casue the bug in other execution paths.

2. On the other hand, if it is on a buggy execution path, the constraint is more likely to cause the bug.

### 3.3.2 Template Patch Creation and Analysis

In this step, each suspicious statement discovered in the previous step is substituted with a linear template patch to create a template-patched program. Then, each program that has temporarily been patched will be verified against its own specification, to obtain a set of proof obligations (or entailments) related to the expression template.

### 3.3.3 Solving Constraints to Discover Repaired Programs

After collecting the entailments for the template patches from the template-patched programs, it is needed to solve a set of entailments (proof obligations). To do this, the authors use Farka's Lemma [30].

Theorem 1 (Farkas' Lemma). Given a system $S$ of linear constraints over real-valued variables $x_1, \ldots, x_n$ :

$$S \triangleq \bigwedge_{j=1}^{m} \sum_{i=1}^{n} a_{ij} \cdot x_i + b_j \geq 0.$$

When $S$ is satisfiable, it entails the following linear constraint $\psi$ :

$$\psi \triangleq \sum_{i=1}^{n} c_i \cdot x_i + \gamma \geq 0$$

if and only if there exists non-negative numbers $\lambda_1, \ldots, \lambda_m$ such that

$$\bigwedge_{i=1}^{n} c_i = \sum_{j=1}^{m} \lambda_j \cdot a_{ij} \quad \text{and} \quad \sum_{j=1}^{m} \lambda_j \cdot b_j \leq \gamma$$

Given the set of entailments, which contain unknown coefficients of the template patch, it is solved by the authors by transforming the entailments and applying Farka's lemma to eliminate universal quantification, and using an off-the-shelf prover like Z3 to find the concrete values for the coefficients of the patch.

Although the described project only supports repairing linear arithmetic expressions, it has since been extended to support repairing more types of expressions such as arrays, strings and even dynamically allocated data structures. It has also been extended to support fixing more than one expression at a time.

In our work we try to apply the same methodology as the authors, with some key differences: instead of doing the verification and synthesis, we will try to use already available solutions for both steps of the process - in case, we will use the Dafny Verifier to do the program verification, and as synthesizer we will use Suslik, but we will apply some architectural decisions that allow us to easily extend the implementation, leveraging comparison between multiple synthesizers.

## 3.4 Systematic method to deduce and synthesize Dafny programs

In their work, the authors of the article introduce a systematic method that is used to deduce and synthesize Dafny Programs [28].

Initially, the authors begin by describing the problem using strict mathematical language. Then, the problem goes through the derivation process, in which the program is derived using a set of mathematical rules that allow the authors to transform the program in a way that allows the Dafny program to be defined by recursive relationships and loop invariants.

In their work, the authors provide us with two examples: firstly, the authors apply the method defined to derive the minimum contiguous sum in an array. This is done by applying a divide-and-conquer approach, subdidiving the problem in smaller sub-problems with the same structure, and then constructing a resursive relationship used to solve the problem.

From this article we can denote an approach being done by other authors that can be used in a future iteration of our project, in which the development of a native Dafny synthesizer can be compared against the translation method.

# 4

# Solution Development

## Contents


4.1 Architecture Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

4.2 Dafny-Boogie Interaction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

4.3 Translation Process . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

In this chapter, a description of the approach taken to achieve the solutions previously described is presented. In the previous section, we described our choice of technology, with Dafny being chosen as language and verifier. With the choice of technology, Boogie was also introduced, an intermediary language on which Dafny programs are translated to, and that we will closely follow in our solution.

This chapter is divided into multiple sections: firstly, the overall architecture of the solution is presented, where we delve deeper into the input and output of each section. Then, we present the architectural decisions and patterns that were applied, and lastly we finish the chapter with a small demo of our implementation.

## 4.1 Architecture Overview

As presented in Figure 4.1, our solution is made up from different components, each with their own purpose. In the overview figure we can see that the program the user designed is input to the Dafny verifier, which in turn translates it to Boogie and passes it to the Boogie verifier. This is where our main area of focus lies. After the verification process is done, a list of errors is passed to the synthesis/translation module we implemented, where it gets translated by a translation module to the target synthesis language. This is then passed to the synthesizer, that is responsible for providing the user with the generated code.

In the next sections we analyze the solution development by going over the architecture and the design decisions that were taken. Firstly, we will shortly glance over the Dafny-Boogie interaction, that although not developed by us needs to be addressed for the purposes of understanding the underlying mechanics of the soluton. Then, we will look over the translation process, in the first step to the architectural pattern applied to reach the solution, and after that to the translation algorithm. We then follow up by presenting another script, the one responsible for making the entire solution work.

## 4.2 Dafny-Boogie Interaction

As we mentioned in Section 2.2, Dafny is not only a language, but also a capable verifier. This verifier is developed in a .NET solution, with multiple components. For the sake of this project, we only used one of these components, `DafnyDriver`. This component is the main component responsible for taking a Dafny program, verifying it, and outputing to the user the result of the verification. This is the component responsible for the communication with Boogie, as well as providing the compilation of Dafny verified programs to languages like `C#` or `Go`. We discovered that this component is only responsible for translating the program to Boogie Syntax, calling the Boogie verification tool with the translated input, and, after the verification is done, receiving the output from it.

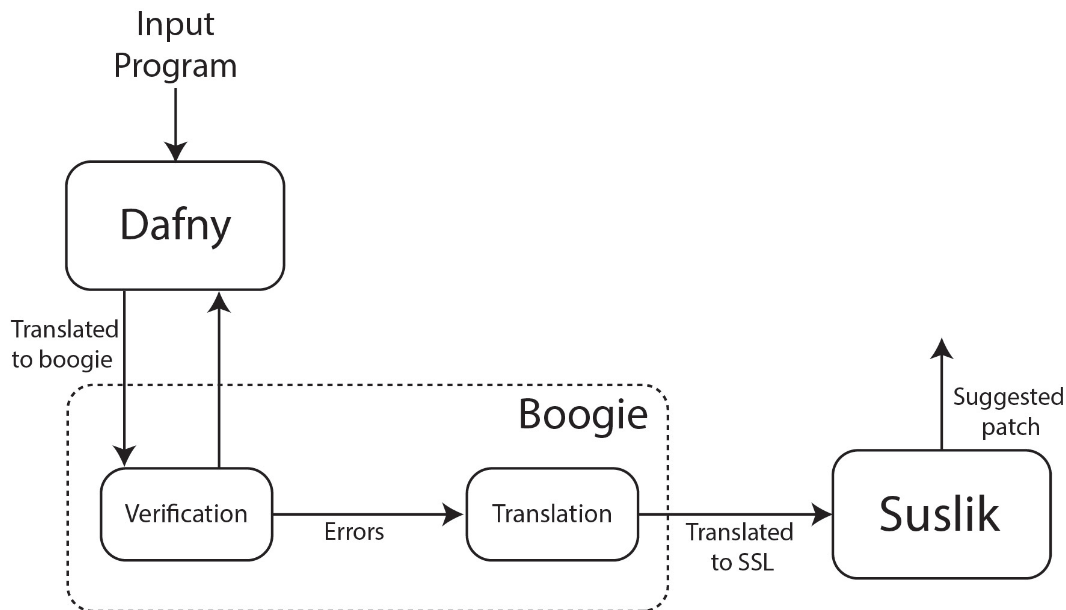


**Figure 4.1:** Solution Architecture Overview

Boogie is also a .NET solution, and to run the Dafny verifier it gets bundled in the solution as a dependency managed by *NuGet*. After providing Dafny with a custom implementation of Boogie, which we cloned from the official *Github* repository, we discovered that Dafny didn't actually get the output of the verification process, but rather only a summary of what's happened during the verification process. The Boogie tool is also responsible for compilling the Boogie language to verification conditions which get passed to Z3.

## 4.3 Translation Process

After the verification is done by the Boogie verififer, a list of counterexamples/errors is generated. The translation module is responsible for analyzing the output of this step, and from it generate the template patch we are going to pass to the synthesizer.

### 4.3.1 Module Architecture

As we previously described, in the process of designing the translation module we intended from the start to leverage comparison between different synthesizers. This, along with the output of the verifier,

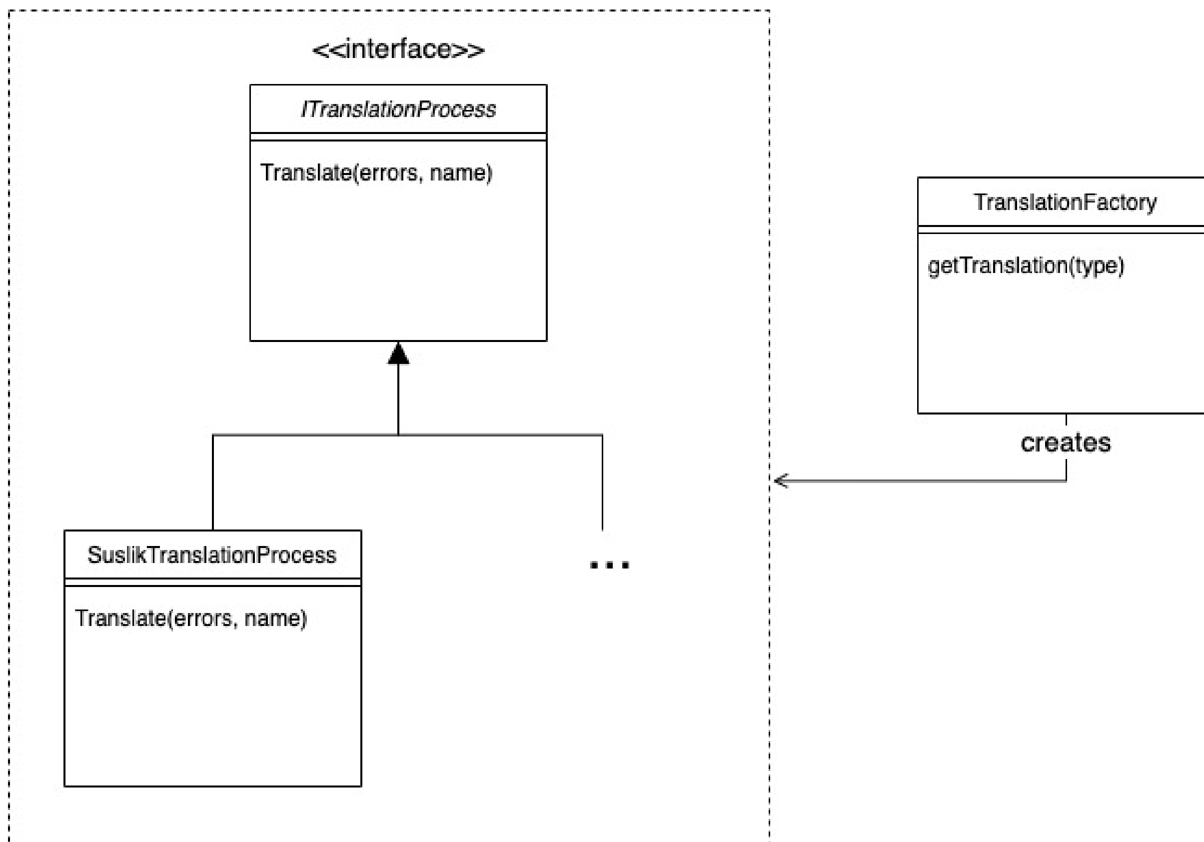


**Figure 4.2:** TranslationFactory Pattern applied

are the main driving reasons behind some of the design decisions we took.

In Fig. 4.2 we can see the structure of the Translation method. The `TranslationFactory` class has a single public method available: `getTranslation`. In it's implementation, the `TranslationFactory` class is responsible for calling the correct implementation of the interface `ITranslationProcess`. In our case, we have one implementation: `SuslikTranslationProcess`.

**Listing 4.1:** GetTranslation implementation

```
1 public ITranslationProcess GetTranslation(string type) {
2     switch (type) {
3         case "suslik": return new SuslikTranslationProcess();
4         default: throw new ArgumentException("Invalid type", type);
5     }
6 }
```

This design pattern is very important in our solution since it allows for easily extending the existing solution with new implementations, for different synthesizers/different strategies. This is present in Listing 4.1, where implementations are given a key. The key for an implementation is useful when

`GetTranslation` is used by another method, to identify the type of translation we want to provide

### 4.3.2 Errors trace structure

As mentioned in previous sections, the output of the verification process is a List of objects of the class CounterExample. In this section, we will look at the structure of this class and what can be used to develop implementations. In Figure 4.3 we can see the structure: a `CounterExample` class that has several relations to other objects that were here displayed as attributes for the simplicity of the diagram, and along with them has a list of `Blocks`, that represent blocks in the code of the input program. In each block there contains a list of `Cmd` that contain the information about the expression of that command.

Using the example presented in Listing 1.1.2.B, after the verification is done, like presented in Listing 1.5, we can see there is an error in the program. After verification, the errors list will have 1 error, since it has only a single method like previously mentioned, and the error is similar to the structure presented in Listing 4.2, with some parts omitted for clarity. As we can see, in line 3 we have the failing ensures, the post-condition that failed, and in lines 6 to 24 we have the trace of the error, the structure that will allow us to generate the state of the program.

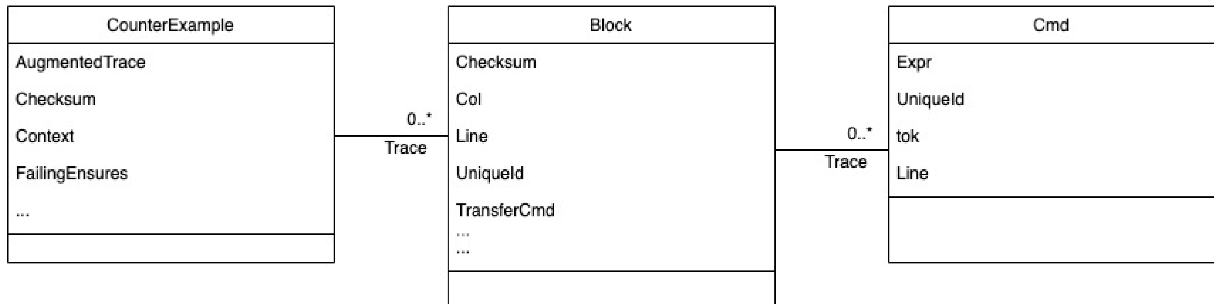


**Figure 4.3:** CounterExample structure

**Listing 4.2:** Error generated by our example program

```
{
    FailingEnsures: {
        Condition: {NaryExpr} j#0 == i#0
    }
    Trace: [
        {
            Label: PreconditionGeneratedEntry
            Cmds: [
                {AssumeCmd} assume $IsGoodHeap($Heap) && $IsHeapAnchor($Heap),
                {AssumeCmd} assume 0 == $FunctionContextHeight,
                {AssumeCmd} assume i#0 >= LitInt(0)
            ]
        },
        {
            Label: anon0,
            Cmds: [
                {AssumeCmd} assume $_Frame#AT#0 == lambda#0(null, $Heap, alloc, false);
                {AssumeCmd} assume true,
                {AssumeCmd} assume true,
                {AssumeCmd} assume j#0#AT#0 == LitInt(2),
                {AssertEnsuresCmd} assert j#0#AT#0 == i#0,
            ]
        }
    ]
}
```

### 4.3.3 Suslik SSL Translation

Like mentioned in the previous sections, in our solution we have one class implementing `ITranslationProcess`. This class is responsible for encapsulating the logic we developed to translate the programs from Boogie Syntax to Suslik's own SSL-compliant syntax. We describe this translation in this Section.

#### 4.3.3.A Base Implementation

For this next section we will be focusing on algorithm. The implementation has several steps, as it's represented in Listing 4.3, which we will get into more detail later in this section.

**Listing 4.3:** Translate psudo-code

```
1  string translate(List<CounterExample> errors, string programName) {
2      var (headerVars, postCondVars) = findAllVars(errors);
3
4      var preCalculatedState = calculateState();
5
6      var failingEnsures = errors[0].FailingEnsures;
7
8      var generatedPreCond = genPreCond(preCalculatedState);
9      var generatedPostCond = genPostCond(failingEnsures, preCalculatedState);
10
11     assemblePatch(programName, headerVars, generatedPreCond, generatedPostCond)
12 }
```

As shown by the pseudo-code, the main source of information to run our algorithm is the errors list provided by the Boogie verifier, like presented in Section 4.3.2. Since we will only be considering one method defined in each file, and each method only having only one error, we can safely assume the errors trace will always have a single element.

The errors list consists of a collection of objects with information about the proof that was done by Z3. While this does not contain information about the program state at the point where it failed, it does contain the trace of the expressions that came before the corresponding error in the list.

By following the Trace of expressions of the error, we can roughly estimate the state of the program at the point where the erroneous state was found. This is done by iterating over the trace and generating the information we need: what the state of the program was before the error, and what it should be like after the error. After collecting this information, we can generate the syntax of the synthesizer and pass the request to it.

This is where one of the big decisions we took deeply impacted our solution: to generate the pre-

and post-state of the program, we follow the trace and save the state of each variable in a Map, replacing the previous declaration with the new unresolved value, instead of evaluating it and replacing the mapping with the new value. We decided against implementing an evaluator with framing since the implementation of such evaluator is out of scope for this project, and we left it in the backlog for further implementations.

The algorithm works by firstly finding all the variables the program has: the header variables, meaning variables that are present in the header of the original method and the post-condition. These variables are used since Suslik has a particularity: all the variables in the pre- and post-conditions need to be in the header, and vice-versa. Secondly, we calculate the state up to the point where the error occurred, and finally the failing assertion. Lastly, we will generate the pre- and post-condition of the method, by combining the pre-calculated state with the variables that exist in the headers but are never referenced in the body of the functions to generate the pre-conditions, and the pre-calculated state along with the failing assertion. This ensures that nothing in the program will be changed except for the variables included in the desired assertion.

Following our algorithm with our motivational example, firstly we explore all the trace to find the variables that need to be present in the header, the variables that belong to the Post-Condition and the ones present in the Pre-Condition. This is done by iterating over the trace, and filtering out the `IdentifierExpr` that are present in in `AssumeCmd`. Some of these Identifiers need to be ignored, such as the ones present in lines 9, 10 and 17, with those being statements to assume the state of the heap and framing conditions. Since we are not using those structures, we can safely ignore them. To collect the header variables we need to iterate through all the trace, and create a list containing them. Then, to find the variables in the postCondition, we need to look at the `FailingEnsures`. This will guarantee that in the end, the post-condition of the TP contains the variable of the original method's post-condition, even if it does not appear in the body.

After collecting all the variables, we need to calculate the pre-state before the bug. This is done once again by iterating over the trace, collecting all the variables and the assignments made to them, ignoring the framing and heap assumptions. In the example this generates a list with two expressions: `[i > 0, j == 2]`. Finally, after generating the `preCalculatedState`, we will collect the failingEnsures and initiate the translation: We are going to generate the pre-Condition based on the pre-calculated state, and the post-condition based on the pre-calculated state, with the mapping of the variable `j` replaced by the failed `Ensure` statement.

This will generate the output presented in Listing 4.4: the post condition contains an empty mapping for the variable i, and the variable j points to an address with 2 in it. In the post-condition, the variable i does not change, and the variable j points to an address with the value of i in it.

Listing 4.4: Intermediary program in SSL logic

```
Problematic program
###
{ i :-> 0 ** j :-> 2 }
void value (loc i, loc j)
{i :-> 0 ** j :-> i }

###
```

#### 4.3.3.B Extending the base implementation to support branching

After our base case was working, we extended the algorithm to support branching. In the Boogie language branching is handled in a very specific way: when there is an `if-else` expression, the structure of the errors trace has some changes, essentially having a `GeneratedUnifiedExitCondition` in the trace is created. This command used to define a common exit-point for both branches, without splitting the flow of the program. So, to support branching, we had to take that into account.

Listing 4.5: Translate extension to support branching

```
string translate(List<CounterExample> errors, string programName) {

    var (headerVars, postCondVars, preCondVars) = findAllVars(errors);
    var (preConditionGeneratedEntry, preCalculatedState) = calculateState();

    var failingEnsures = errors[0].FailingEnsures;

    if (branchingInProgram()) {
        var generatedPreCond = genPreCondWithBranching(preCalculatedState, preCondVars);
    } else {
        var generatedPreCond = genPreCond(preCalculatedState, preCondVars);
    }

    var generatedPostCond = genPostCond(failingEnsures, preCalculatedState);
    assemblePatch(programName, generatedPreCond, generatedPostCond)
}
```

The main changes we had to add to the algorithm were: firstly, the `GeneratedUnifiedExitCondition` had to be handled differently by the algorithm: if the flow of the program ends inside of a branch,

the `GeneratedUnifiedExitCondition` will not add any information to the state of the program, instead the condition generated is an assertion of the post-condition. This can be seen in Listing 4.5, in line 9, where we call a different implementation. We also had to treat the last assignment of the branch differently - if the flow of the program ends here (with a return or last assignment to the end variable) the last assignment is generated as an `assumption` that does not add any information to the state of the program.

### 4.3.4 Connection to the Synthesizer

After developing the implementation of the translation module, the program had to be connected to the synthesizer. During this step, we tried different alternatives, including an attempt to generate a `DLL` to introduce the functionality of Suslik into our program, but ultimately the attempt failed since the framework responsible for generating the `DLL` didn't work as intended, and we weren't able to reliably call the needed methods.

For this reason, we decided to use IO to develop the integration script, and develop the IO through files. This was partly decided since Suslik expects a file to work. To develop the integration script, we used `bash`, to create a script that would call the Dafny verification, translation, and later forward the output of that step to the synthesizer. This, of course, is not the case if the program had no errors. If it didn't have any errors, the program will stop right after the verification phase.

# 5

# Evaluation

## Contents

5.1 Initial considerations . . . . . . . . . . 45
5.2 Functional Example . . . . . . . . . . 45
5.3 Functional Example of the Extension . . . . . . . . . . 46
5.4 Program Benchmark . . . . . . . . . . 47

In this chapter, we detail the evaluation methods of our solution, how the benchmark of programs was developed and the results we obtained with the benchmarks.

## 5.1 Initial considerations

During the development of our project, we created several inputs to test our implementation. Although the coverage of the solutions is not very diversified and the examples are in a limited number, they are enough so we can take some conclusions from this state-of-the-art project. In the following sections we will present some examples, look at the generated output, rationalize about its correctness and benchmark our solution.

## 5.2 Functional Example

As mentioned before, the coverage of our project is limited only to simple programs with one or multiple assignments, and simple branching. For the first example, let's consider the code present in Listing 5.1, a simple program that given an input variable i, returns the value of that variable. From the specification we can see there is an error in line 5. Using our solution, we are successfully able to generate a patch to replace the expression.

**Listing 5.1:** Value method

```
1 method value(i: int) returns (j: int)
2     requires i >= 0
3     ensures j == i
4 {
5     j := 2;
6 }
```

The generation process initially verifies the program, generates the intermediary program in Suslik's syntax and calls the synthesizer to get the result. The process can be seen in Fig. 5.1.

The intermediary program generated is present in Listing 4.4. The variable i is generated with an empty mapping, thus pointing to zero and the variable j is generated pointing to two, referencing the last assignment to the variable. This is because from the errors trace the only way of knowing what is the last statement is by discarding the last expression of one of the blocks. This would be problematic in case of branching, so we decided to keep the last statement, and assume the patch would be generated and replaced after it. In this case, the generated patch can be seen in Figure 5.1, and it can be used by replacing the original statement or appending it to the body of the program `j := i`.

```
❯ ./run-test.sh 0.dfy
                                                                    by hmartins

Running example: 0.dfy

-- First Phase: dafny verification --

        Generated the output file in Dafny-Examples/interm/ called interm.interm
-- Second Phase: Suslik Generation --

{i :-> 0 ** j :-> 2}
{i :-> 0 ** j :-> i}
void sum (loc i, loc j) {
  *j = i;
}
```

**Figure 5.1:** Solution Integration Script

## 5.3 Functional Example of the Extension

By taking a look at Listing 5.2 we can see the method whose signature is: the name of the method is 9, and it receives an integer. Its behavior guarantees the same as the previous example, but with branching added. If `i`'s value is 0, it will return the value of i. Of course, this program has a bug added, in line 9. `j` will take the value of `k`, which in turn was defined in line 8 to have value 1.

**Listing 5.2:** Example method with branching represented in the benchmark by test 9.dfy

```
1  method 9(i: int) returns (j: int)
2      requires i >= 0
3      ensures j == i;
4  {
5      if i == 0 {
6          j := i;
7      } else {
8          var k := 1;
9          j := k;
10     }
11 }
```

After running our solution, the intermediary program generated by our solution will have three locations defined: `i, k, j`, along with a pre-condition containing an empty mapping for variable i, k pointing to 1 and j having the value of variable k. The post-condition will keep variable i and k with the same values, but it will change the variable j to point to the value of variable i. This can be seen in Listing 5.3.

```
❯ ./run-test.sh 9.dfy
 ___        __           ___                      _   _
|   \ __ _ / _|_ _ _  _  / __|___ _ _  ___ _ _ __ _| |_(_)___ _ _
| |) / _` |  _| ' \ || | | (_ / -_) ' \/ -_) '_/ _` |  _| / _ \ ' \
|___/\__,_|_| |_||_\_, |  \___\___|_||_\___|_| \__,_|\__|_\___/_||_|
                   |__/                              by hmartins

Running example: 9.dfy

-- First Phase: dafny verification --

        Generated the output file in Dafny-Examples/interm/ called interm.interm
-- Second Phase: Suslik Generation --

{i :-> 0 ** j :-> k ** k :-> 1}
{i :-> 0 ** j :-> i ** k :-> 1}
void TP (loc i, loc k, loc j) {
  *j = i;
}
```

**Figure 5.2:** Output of running test 9.dfy

The final output of the generation is shown in the output of the integration script present in Figure 5.2, with the final fix for the problem being assigning to the variable j the value of the variable i (`j := i`).

**Listing 5.3:** Example of generated intermediary program for example 9.dfy

```
1
2 Problematic program
3 ###
4 { i :-> 0 ** k :-> 1 ** j :-> k }
5 void TP (loc i, loc k, loc j)
6 {i :-> 0 ** k :-> 1 ** j :-> i }
7
8 ###
```

## 5.4 Program Benchmark

As demonstrated by Table 5.1 present in this section, we created a small benchmark of simple programs our implementation mostly supports. From what we can see, our solution is somewhat effective to repair simple programs in an acceptable amount of time.

Another thing we should take into consideration is that the standard verification process, without intervention from our solution, takes around 2.06 seconds. Our most complex example takes only 3.18 seconds, and although it may seem like a big a difference, with an increase of 65% of the time, we were

| Program name | Small Description | Generated Solution (Y/N) | Time (s) |
|---|---|---|---|
| 0.dfy | Correct program | - | 2.06 |
| 1.dfy | Simple variable attribution | Y | 2.44 |
| 2.dfy | Attribution of a variable with a simple expression | Y | 3.08 |
| 4.dfy | Attribution of a variable with a complex expression | Y | 3.16 |
| 5.dfy | Multi-line attribution of simple variables | Y | 3.10 |
| 6.dfy | Multi-line attribution with expressions | Y | 3.23 |
| 7.dfy | Branching program with wrong return statement in IF branch | Y | 3.03 |
| 8.dfy | Branching program with wrong return statement in ELSE branch | Y | 3.02 |
| 9.dfy | Branching program with multiple attributions | Y | 3.10 |
| 10.dfy | Branching program with body after the branching statement | Y | 3.18 |
| max.dfy | Maximum of two variables with conditional return | N | |
| sum.dfy | Sum program presented in the motivational example | N | |

**Table 5.1:** Results

able to generate a solution to a bug that would take significantly more time to find.

# 6

# Conclusion

## Contents

6.1 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
6.2 Future work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51

## 6.1 Conclusions

As we described in the last sections, the results of our solution are rather interesting: with only a slight increase from the verification time, we were able to create corrections for multiple programs.

Despite the range of programs supported being very limited and very synthetic, we can take this implementation as an example of automated program repair in Dafny, and a proof that it can be done.

Our implementation not only proves that APR can be done in Dafny, but it also establishes a baseline for future comparisons, with improvements that can be done to increase the scope of supported programming constructs, some of which we describe in the next section

## 6.2 Future work

To actually have a tool to automatically suggest fixes/repair programs some major improvements need to be done, we will go over them in this section.

Firstly, since our approach only statically evaluates a program, assignments against same variables (e.g. `sum := sum + 1`) would be a problem, since the translation wouldn't account for the value present in the variable and it would never be possible to synthesize a patch for that statement.

Secondly, as we mentioned in the previous chapter, we assumed that the last statement of the program would be used in the synthesis since ignoring it would be a problem in several cases. This, despite it being a relatively simple change to our code, goes to show that our implementation has some deep flaws: since the trace accessible from Z3 is limited, it includes limited information, and it would be impossible to achieve program repair without creating a "second" prover in the program.

And lastly, since Dafny uses Dynamic Frames and Suslik has a different implementation named SSL, the range of programs supported would be limited from the start. To have an approach that works with the full range of Dafny's programming constructs, a synthesizer would need to be developed specifically for Dafny.

# Bibliography

[1] S. McConnell, *Code Complete, Second Edition*. USA: Microsoft Press, 2004.

[2] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 802–811.

[3] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, pp. 54–72, 2012.

[4] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, 2017, p. 1345–1351.

[5] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, p. 1943–1959, Sep 2021.

[6] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: combining context-aware neural translation models using ensemble for program repair," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. Association for Computing Machinery, Jul 2020, p. 101–114. [Online]. Available: https://doi.org/10.1145/3395363.3397369

[7] E. Kneuss, M. Koukoutos, and V. Kuncak, "Deductive program repair," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, D. Kroening and C. S. Păsăreanu, Eds. Springer International Publishing, 2015, p. 217–233.

[8] R. Könighofer and R. Bloem, "Automated error localization and correction for imperative programs," in *In FMCAD*, 2011.

[9] T.-T. Nguyen, Q.-T. Ta, and W.-N. Chin, "Automatic program repair using formal verification and expression templates," in *Verification, Model Checking, and Abstract Interpretation*. Springer

International Publishing, 2019, pp. 70–91. [Online]. Available: http://link.springer.com/10.1007/978-3-030-11245-5_4

[10] K. R. M. Leino, *Dafny: An Automatic Program Verifier for Functional Correctness*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, p. 348–370.

[11] N. Polikarpova and I. Sergey, "Structuring the synthesis of heap-manipulating programs," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, p. 1–30, Jan 2019.

[12] K. R. M. Leino and A. Milicevic, "Program extrapolation with jennisys," *SIGPLAN Not.*, p. 411–430, 2012.

[13] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, p. 576–580, 1969.

[14] M. Huth and M. Ryan, *Logic in computer science - modelling and reasoning about systems.* Cambridge University Press, 01 2000.

[15] C. Belo Lourenco, M. J. Frade, S. Nakajima, and J. Sousa Pinto, "A generalized approach to verification condition generation," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, 2018, p. 194–203. [Online]. Available: https://ieeexplore.ieee.org/document/8377656/

[16] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, p. 385–394, jul 1976.

[17] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer-Verlag, 2008, p. 337–340.

[18] M. Barnett, B.-Y. Chang, R. Deline, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *Formal Methods for Components and Objects*, 09 2006, pp. 364–387.

[19] I. Kassios, "Dynamic frames: Support for framing, dependencies and sharing without restrictions," in *Proceedings of the 14th International Conference on Formal Methods*, 08 2006, pp. 268–283.

[20] B. Meyer, "Framing the frame problem," p. 11, 2015.

[21] S. Srivastava, S. Gulwani, and J. S. Foster, "Template-based program verification and program synthesis," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 5–6, p. 497–518, Oct 2013.

[22] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, “Synthesis of loop-free programs,” *SIGPLAN Not.*, p. 12, 2011.

[23] A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, S. R, and S. Roy, “Program synthesis using natural language,” in *Proceedings of the 38th International Conference on Software Engineering*. ACM, May 2016, p. 345–356.

[24] S. Gulwani, “Dimensions in program synthesis,” in *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming - PPDP ’10*. ACM Press, 2010, p. 13–24. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1836089.1836091

[25] G. Katz and D. Peled, “Synthesis of parametric programs using genetic programming and model checking,” *Electronic Proceedings in Theoretical Computer Science*, vol. 140, p. 70–84, Feb 2014, arXiv: 1402.6785.

[26] S. Gulwani, S. Srivastava, and R. Venkatesan, “Program analysis as constraint solving,” *SIGPLAN Not.*, p. 12, 2008.

[27] E. W. Dijkstra, “Solution of a problem in concurrent programming control,” *Communications of the ACM*, vol. 26, no. 1, p. 21–22, Jan 1983.

[28] C. Wang, X. Ding, J. He, X. Chen, Q. Huang, H. Luo, and Z. Zuo, “A method to deduce and synthesize the dafny programs,” vol. 26, no. 6, pp. 481–488. [Online]. Available: https://wujns.edpsciences.org/10.1051/wujns/2021266481

[29] N. Polikarpova and I. Sergey, “Structuring the synthesis of heap-manipulating programs,” *Proc. ACM Program. Lang.*, vol. 3, pp. 1–30, 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3290385

[30] M. Colon, S. Sankaranarayanan, and H. Sipma, “Linear invariant generation using non-linear constraint solving,” *Lecture Notes in Computer Science*, vol. 2725, p. 420–432, Jan 2003.