Automated Program Repair of Arithmetic Programs in Dafny

Hugo Martins
hugo.r.f.martins@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

October 2022

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

1. Introduction

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 [17]. 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 [9, 13] and deep learning [5, 3, 16].

Test-suite [13] 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 [13, 9], 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 [11, 12, 19]. 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 [14]. 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 the scope in the future.

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. [19], 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. [20]

2. 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* [15] 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.

2.1. 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.

2.2. Motivational Example

In this section, we present the motivating example we will be using throughout the thesis. We will be presenting a simple program that returns the value of a variable. This program is part of our benchmark presented in Section 7. Without getting into much detail, since we will look further into the specification of Dafny programs in Section 3.2, we will now go over what the program does, and its implementation.

The program's specification has a pre- and a post-condition, and although these terms will be defined in Section 3.1 in further detail, they essentially mean that when the method call begins it *requires* that i is bigger or equal to 0, and that when the program exits (and thus, returns) the variable to be returned (j) will be equal to i.

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 5 in Figure 1 with `j := i`, and by fixing it the Dafny verifier successfuly verifies the program with 0 errors.

This is verified by Dafny as having one error, and the verification can be done like in Figure 2, where we can see that the state of the program in line 163 does not imply the post-condition of the method.

```
method value(i: int) returns (j: int)
    requires i >= 0
    ensures j == i
{
    j := 2;
}
```

Figure 1: Value method implementation

```
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
}
```

Figure 2: Value method verification

3. Background

3.1. 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.

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

Figure 3: Hoare inference rules

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.

We use Hoare Logic [6] 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. 3 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 [7] .

3.2. 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* [2] and *Symbolic Execution* [10]. 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 tech-

nique 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 [4] 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* [1] 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.

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.

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

Figure 4: Loop invariants

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 4 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 4 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.

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 [8], Dafny's approach to solving the *Frame Problem* [18].

4. Related Work

4.1. 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. [21], 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. 5 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 5: 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 approach 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.

5. Jennisys

Jennisys [15] 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.

6. Implementation

6.1. Architecture Overview

Our solution is made up from different components, each with their own purpose. 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.

6.2. Dafny-Boogie Interaction

As we mentioned in Section 3.2, 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.

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.

6.3. Module Architecture

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.

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

To facilitate the integration of different implementations, we applied a Factory Pattern. 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`.

6.4. Errors trace structure

As mentioned in previous sections, the output of the verification process is a List of objects of errors of the class CounterExample. This CounterExample class has relations with many objects, and among them 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 that will be used to generate the pre-state of the program.

6.5. 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.

6.5.1 Base Implementation

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 6.4. 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.

The algorithm works by firstly finding all the

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

    var (headerVars, postCondVars) =
        findAllVars(errors);

    var preCalculatedState =
        calculateState();

    var failingEnsures =
        errors[0].FailingEnsures;

    var generatedPreCond =
        genPreCond(preCalculatedState);
    var generatedPostCond =
        genPostCond(failingEnsures,
            preCalculatedState);

    assemblePatch(programName,
        headerVars, generatedPreCond,
            generatedPostCond)
}
```

Figure 6: Translate psudo-code

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.

6.5.2 Extending base implementation

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.

```
method 9(i: int) returns (j: int)
    requires i >= 0
    ensures j == i;
{
    if i == 0 {
        j := i;
    } else {
        var k := 1;
        j := k;
    }
}
```

Figure 7: Example method with branching represented in the benchmark by test 9.dfy

6.6. Connection to 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.

7. Results and discussion

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.

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 Figure 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.

Taking into account the program in Figure 7, we can see that the generation process initially verifies the program, generates the intermediary program

```
Problematic program
###
{ i :-> 0 ** k :-> 1 ** j :-> k }
void TP (loc i, loc k, loc j)
{i :-> 0 ** k :-> 1 ** j :-> i }

###
```

Figure 8: Intermediary Suslik-syntax file generated with 9.dfy

in Suslik's syntax and calls the synthesizer to get the result. An intermediary program is created in the translation phase, where 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 can be seen in Figure 8.

The reason for the last assignment to be present is that 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.

7.1. Benchmark

As demonstrated by Table 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 able to generate a solution to a bug that would take significantly more time to find.

8. 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

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	

Table 1: Results

8.1. 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.

References

[1] 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*, pages 364–387, 09 2006.

[2] 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)*, page 194–203, 2018.

[3] 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*, 47(9):1943–1959, Sep 2021.

[4] 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*, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[5] 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*, page 1345–1351, 2017.

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

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

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

[9] 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)*, pages 802–811, 2013.

[10] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, jul 1976.

[11] E. Kneuss, M. Koukoutos, and V. Kuncak. Deductive program repair. In D. Kroening and C. S. Păsăreanu, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, page 217–233. Springer International Publishing, 2015.

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

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

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

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

[16] 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*, ISSTA 2020, page 101–114. Association for Computing Machinery, Jul 2020.

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

[18] B. Meyer. Framing the frame problem. page 11, 2015.

[19] 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*, pages 70–91. Springer International Publishing, 2019.

[20] N. Polikarpova and I. Sergey. Structuring the synthesis of heap-manipulating programs. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–30, Jan 2019.

[21] N. Polikarpova and I. Sergey. Structuring the synthesis of heap-manipulating programs. *Proc. ACM Program. Lang.*, 3:1–30, 2019.