Exploring Automatic Specification Repair in Dafny Programs

Alexandre Abreu
Department of Informatics Engineering
Faculty of Engineering,
University of Porto & INESC TEC
Porto, Portugal
up201800168@up.pt

Nuno Macedo
Department of Informatics Engineering
Faculty of Engineering,
University of Porto & INESC TEC
Porto, Portugal
nmacedo@fe.up.pt

Alexandra Mendes
Department of Informatics Engineering
Faculty of Engineering,
University of Porto & INESC TEC
Porto, Portugal
alexandra@archimendes.com

***Abstract*—Formal verification has become increasingly crucial in ensuring the accurate and secure functioning of modern software systems. Given a specification of the desired behaviour, i.e. a contract, a program is considered to be correct when all possible executions guarantee the specification. Should the software fail to behave as expected, then a *bug* is present. Most existing research assumes that the bug is present in the implementation, but it is also often the case that the specified expectations are incorrect, meaning that it is the specification that must be repaired. Research and tools for providing alternative specifications that fix details missing during contract definition, considering that the implementation is correct, are scarce.**

This paper presents a preliminary tool, focused on Dafny programs, for automatic specification repair in contract programming. Given a Dafny program that fails to verify, the tool suggests corrections that repair the specification. Our approach is inspired by a technique previously proposed for another contract programming language and relies on Daikon for dynamic invariant inference. Although the tool is focused on Dafny, it makes use of specification repair techniques that are generally applicable to programming languages that support contracts. Such a tool can be valuable in various scenarios, such as when programmers have a reference implementation and need to analyse their contract options, or in educational contexts, where it can provide students with hints to correct their contracts.

The results of the evaluation show that the approach is feasible in Dafny and that the overall process has reasonable performance but that there are stages of the process that need further improvements.

***Index Terms*—contract programming, automatic program repair, contract repair, Dafny**

I. Introduction

As software becomes increasingly pervasive in our daily lives — from the ubiquitous presence of mobile phones to the vital role of medical equipment in diagnosing and treating medical problems — the dependency on its correct functioning grows. It is more important than ever to design software with its correctness in mind, i.e. following approaches that guarantee that the software is free from bugs.

Contract programming, also known as *design by contract* [1], [2] or *DbC*, is a way of programming based on the idea of precisely and formally specifying the expected behaviour of software components by providing assertions (also known as *contracts*). Created with reliability in mind, contract programming aids programmers in achieving correct and robust software.

Contracts serve as a means to specify what obligations and guarantees are provided by a software component. Programming languages that support contracts provide verifiers for checking if the contracts are followed by the implementation, forcing the programmer to implement a program that satisfies the specified contract. A program that is verified can be trusted to follow its specification. However, when a program fails to verify, it can mean one of two things: either the specification is correct and the implementation is not, or vice versa — the implementation is correct, but the specification is not. Most current research, however, assumes that the specification is correct, focusing on repairing the implementation. Yet, it is known that many issues with software stem from incorrect specifications that provide a false sense of security [3]. In this work, we aim to improve the research in the other direction by shifting the focus towards fixing the specification.

Research on repairing specifications is valuable in scenarios where verification fails but we can assume the implementation to be correct. A good example is when programmers have a reference implementation of an algorithm and are still analysing their contract options. In such cases, when the programmers trust the implementation to be correct, suggestions to change the contracts based on it would be helpful. Once the contract is sound according to the reference implementation, it can then be used in the verification of more advanced and optimized implementations. This type of work can also be useful in precisely documenting (in the form of specifications) the assumptions and guarantees of the environment in which the software component will be deployed, by suggesting appropriate specifications for the software. A last scenario where contract repair may prove useful is in the educational context, where exercises may expect students to write contracts for implementations provided by the instructors. Students are known to struggle when writing formal specifications [4], and automated repair techniques can be used to provide hints to help students autonomously correct their contracts.

In this paper we present a tool, focused on Dafny pro-

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project EXPL/CCI-COM/1637/2021.

grams [5], for automatically fixing specifications in contract programming. Given a Dafny program that fails to verify, the tool suggests and applies relevant contract fixes that repair the bug — in the sense that the program is verified. Our approach is inspired by a technique previously proposed for contracts in Eiffel programs [6]. Although our tool is focused on Dafny, it makes use of specification repair techniques that are generally applicable to programming languages that support contracts.

We also present the results of our experimental evaluation, which shows that the approach is feasible in Dafny but also that the overall process can benefit from improvements to its efficacy and performance, in particular in what concerns the invariant detection phase.

The rest of this paper is structured as follows. Section II presents a motivating example and an overview of the approach. Section III explores related work relevant for this work. Section IV presents the proposed approach and its implementation, whose experimental evaluation is shown in Section V. Lastly, Section VI wraps up the paper and points directions for future work.

II. Motivation and Overview

Dafny is a high-level verification-aware programming language that is widely used in teaching and has significant adoption in industry, with several companies using it to develop highly-reliable software. For example, at Amazon Web Services, Dafny is used to write and prove a variety of security-critical libraries such as encryption libraries[1]; at Consensys, the leading Ethereum software company, Dafny is used, amongst several other Dafny projects, to verify smart contracts [7] and consensus protocols.

Dafny programs can be written in a combination of functional and imperative paradigms[2] and includes built-in constructs for writing specifications and reasoning about programs. *Functions* (and *predicates*) are written in a functional style and must be free of side effects. *Methods* are written in an imperative style, and can contain loops and side effects. Following design by contract principles, functions, and methods can be assigned contracts, i.e. formal specifications of an agreement between a client and a supplier of a component (in this context, a function/method), where the supplier expects that certain conditions are met by the client before using the component (*pre-conditions*, keyword **`requires`**), maintains certain properties from entry to the component to exit (*invariants*, keyword **`invariant`**), and guarantees certain properties on exit (*post-conditions*, keyword **`ensures`**). Methods are used to provide the implementation of the program and are usually assigned contracts that can call functions that specify the expected behaviour. During compilation, proof obligations are generated that test whether the implementations conform to the contracts. Dafny tries to automatically discharge such proof obligations by relying on Satisfiability Modulo Theories

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

method divRem(d : int, n : int)
  returns (q : int, r : int)
  requires d ≥ 0
  requires n > 0
  ensures r + q ⋆ n = d
{
  r := d;
  var m := abs(n);
  q := 0;
  while r ≥ m
    invariant r + q ⋆ m = d
    invariant q ≥ 0
    invariant m = abs(n)
  {
    q := q + 1;
    r := r - m;
  }
  if n < 0 {
    q := -q;
  }
}

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

Fig. 1. An example Dafny program with a bug, with possible contract fix locations highlighted in red.

(SMT) solvers, guaranteeing the executables obey the contracts (which are discarded after the verification process).

Figure 1 presents a simple example of a Dafny program that happens to fail the verification process. The method `divRem` simultaneously calculates the quotient `q` and the remainder `r` of two integers `d` and `n` using a simple algorithm based on repeated subtractions. The contract states that given a non-negative dividend and a positive divisor, `divRem` returns the correct quotient and remainder. During compilation, Dafny is able to verify that `divRem` obeys that contract. A second method `rem` is also defined, which calls `divRem` and retains only the remainder of the division. However, either by distraction or by being unaware of the `divRem` contract, the contract assigned to `rem` allows the divisor to be negative. At this point, Dafny finds an error during compilation, stating that it cannot guarantee that the pre-condition of `divRem` holds

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

[2]Dafny also supports object-oriented features, such as classes and class invariants, but these are still not fully supported by our prototype and will not be explored in this paper.

when called in `rem`. This means that either the pre-condition of `rem` is too weak, or that the one of `divRem` is too strong. Inspecting `divRem` more closely, it becomes clear that its implementation actually also supports negative divisors, so its pre-condition can be relaxed.

The tool we propose in this paper automatically explores possible fixes to the contracts that result in a correct program, assuming that the implementation is correct. In this particular case, the tool proposes either strengthening the pre-condition of `rem` to forbid negative numbers or weakening the pre-condition of `divRem` to also allow negative divisors.

Figure 2 shows an overview of the proposed approach, inspired by previous work on contract repair for Eiffel [6]. A set of test cases is executed in the incorrect program, so that, from the resulting traces, invariants can be dynamically inferred to characterize the states before and after the execution of each method. These invariants are then used to strengthen or weaken contracts that result in a correct program. For the context of this work, we are mostly focused on the repair generation stage. We leave the automatic generation of test cases and the integration in an IDE to support the application of repairs as future work. For dynamic invariant inference, we rely on *Daikon* [8].

III. Related Work

A. Automated Software Repair

There is a large body of knowledge addressing the automatic repair of programs (as opposed to the repair of their contracts or formal specifications) [9], [10]. Such automatic repair techniques can be roughly divided into two classes: *generate-and-validate* (or *heuristic*) — which explore a search-space for repair candidates which are subsequently tested for correctness — and *constraint-based* (or *semantics-driven*) — where constraints representing a correct repair are derived, and solvers deployed to generate correct-by-construction repairs. Automated repair techniques require that an oracle is provided, specifying the correct behaviour of a program. For most of existing work, this oracle takes the shape of a test suite, but a few techniques consider program contracts or formal specifications as oracles, such as the work by Gopinath et al. for repairing Java programs with associated Alloy specifications [11], or *AutoFix* to repair Eiffel programs with contracts [12]. In the educational context, there are also works that use a reference implementation of a program as an oracle to fix students' submissions [13].

To the best of our knowledge, only one technique has been proposed to automatically repair program contracts, *SpeciFix*, for fixing contracts in Eiffel programs [6]. *SpeciFix* assumes that the program implementation is correct and uses it as the oracle. Other work has focused on repairing formal specifications but without considering implementations as oracles, including for repairing OCL constraints given inconsistent information bases [14] and for repairing faulty Alloy specifications using test cases [15] and reference specifications [16]–[18]. As far as we are aware, no work has focused on the repair of Dafny, either program implementations or their contracts.

B. Invariant Inference

An invariant is an expression that is always true at particular program points during execution. Invariants are useful for, e.g., software verification, repair, and fault localization [19]. They are also useful in software evolution, as, with explicit invariants in the code, programmers can be alerted to any changes that violate assumptions necessary for ensuring the correctness of a program. They act as constraints that must be preserved during the program's execution, regardless of any modifications or changes made to the code. However, despite their advantages and uses, most programmers do not annotate their code with invariants.

A way to increase the existence of invariants in code, is to automatically infer them. There are several existing approaches for invariant inference, which are divided mainly into two main classes: *static* and *dynamic*. Static approaches involve analysing the source code of a program and deriving information without executing it. Dynamic approaches, on the other hand, involve observing and evaluating the behaviour of software while it is executing [20]. The accuracy of dynamic detection is dependent on the coverage and quality of the tests [21] and it can be a computationally expensive process. Combining dynamic invariant detection with static verification of those detected expressions may be desirable [22].

A well-known example of an invariant detection program, and the pioneer in dynamic invariant inference, is *Daikon* [8]. With the help of a test suite that exercises the functionality of a program, *Daikon* receives as input an execution trace and generates a set of invariants that are statistically justified by the trace. There is no formal assurance that these invariants are correct, but they match the observed program executions and there is statistical evidence that their occurrence is meaningful and relevant to the program's behaviour. Also, as with any dynamic analysis, there is the possibility of reporting redundant invariants and properties that are true for certain execution traces but not in general. Although *Daikon* will still report invariants that are not useful, efforts have been made to make the inferred invariants more relevant [8], [23].

Several tools make use of, and improve on, *Daikon*. For example, *iDiscovery* [24] uses symbolic execution to improve the quality of invariants computed by *Daikon* by following a feedback loop of instrumenting the generated invariants into the code, symbolically executing the code to generate new tests, and feeding the new tests into *Daikon* to refine the results. *Daikon* has been used and extended in many other contexts which include the detection of logic vulnerabilities in web applications [25], invariants for relational databases [26], distributed systems [27], robotic systems [28], and Ethereum smart contracts [29].

Although *Daikon* appears to be the most widely adopted, there are other works available on dynamic invariant inference. For example, *DySy* [30], a tool created with the goal of increasing the relevance of inferred invariants when compared with *Daikon*, which combines the concrete execution of actual test cases with a simultaneous symbolic execution of the same

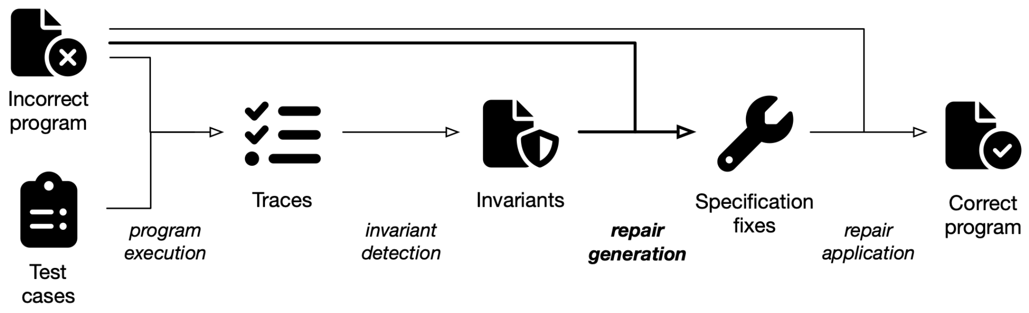

Fig. 2. Overview of the proposed approach, highlighting the main step addressed in this work.

tests. Another example is *DIDUCE* [31], which dynamically extracts invariants from program executions, starting with the strictest invariants and gradually relaxing them when it detects a violation from continually checking the program's behaviour against them. Also, in the context of hardware design, *IODINE* [32] uses dynamic analysis to infer likely invariants based on design simulations.

IV. Contract Repair for Dafny

A. Automated Contract Repair

As shown in Section III, *SpeciFix* [6] is the only technique that has thus far been proposed for repairing contracts of programs, namely of Eiffel programs. As a first step towards a technique to repair Dafny contracts, this work mainly explores whether that approach can be applied to the Dafny context. Here we describe how our procedure works, inspired by the *SpeciFix* approach. Roughly, given a program with contracts, test cases are automatically generated, executed, and tested against the contracts. From the set of passing and failing test cases, invariants are inferred to both strengthen and weaken the contracts. The candidates are then validated against the tests, those that pass being ranked and presented to the user.

More formally, let a *test case* (or simply a *test*) $t = r(a_1, \ldots, a_m)$ be defined by a method r being applied to a sequence of m arguments $a_1, \ldots, a_m$. From the outermost method r call, other methods may be called, and so on recursively. We denote the *call sequence* of a test t by $\kappa_t = r_1 r_2 \ldots r_n$, with $r_1 = r$ if there are n nested method calls. Each method is called from a particular pre-state and produces a post-state. We denote the *trace* of a test t as $\rho_t = s_1 r_1 s_2 r_2 s_3 \ldots s_n r_n$, where s_i denotes a program state.

Each method r_i may be annotated with a pre- and post-condition, which are denoted by P_{r_i} and Q_{r_i}, respectively. Given one such assertion A, we denote the fact that it holds in a state s_i as $s_i \models A$. A test is said to be *valid* if the pre-condition of its outermost method P_r holds in its initial state s_1; invalid tests do not represent acceptable executions and are not even considered during repair. A valid test t is said to be *passing* if all states s_i in its trace ρ_t pass the pre-condition of the succeeding method call P_{r_i} and the post-condition of the preceding method call $Q_{r_{i-1}}$ (for $i \neq 1$); it is said to be failing otherwise. We assume that the execution of a failing test terminates as soon as an inconsistent state is found, and thus, for failing tests, r_n denotes the routine whose contract has been broken (either its pre- or post-condition). For our example, a test $\texttt{rem}(20, 0)$ is invalid, since it breaks the pre-condition of `rem` that `n` $\neq$ `0`, $\texttt{rem}(20, -10)$ is a failing test since it fails the pre-condition of `divRem` that `n > 0` when called from `rem`, and $\texttt{rem}(20, 10)$ is a passing test.

Whenever a failing test t is found, contracts can be fixed in two ways: either strengthening the pre-condition of the outermost method r, making the test invalid; or weakening the contract (pre- or post-condition) of the failing method r_n. Strengthening is always a valid fix, but may break other calls to r. In our example, the pre-condition of `rem` could simply be strengthened to `n > 0`, but this would disallow calculating the remainder of a negative divisor, which is actually possible in the current implementation. Weakening may not always work, as it may result in incorrect executions. In our example, the pre-condition of `divRem` can be relaxed to `n` $\neq$ `0` because the reference implementation actually works correctly for negative divisors; if that was not the case, it would not be possible to fix the contracts by weakening. In [6] the authors argue that to generate fixes that are consistent with the way the API is used, strengthening fixes should actually be applied along the complete trace, and not just to the outermost method. Moreover, they also use the strength of proposed fixes when ranking them to be presented to the user; here we leave fix ranking for future work.

Given this background, the generation of candidate fixes works as follows, generating weakening and strengthening candidate fixes (and their combination).

Trace generation: Execute all test cases, and identify a fault to be fixed, i.e., a set of test cases with the same call sequence that break assertion A_n (either a pre- or post-condition) of method r_n. Let $\mathcal{P}_r$ be the set of passing test cases with the outermost method r, and $\mathcal{F}_r$ the set of failing tests for that fault. In our example, A_n is a pre-condition of `divRem`, `d > 0`.

Generating weakening fixes: The result of this stage is a set of weakening fixes Φ_W and a set of candidate weakening fixes Ψ that will be combined with strengthening candidates in the next stage.

- Let $\tilde{r_n}$ be a version of method r_n with the broken assertion A_n relaxed to **`true`**, and create new sets of passing and failing tests, $\tilde{\mathcal{P}_r}$ and $\tilde{\mathcal{F}_r}$. From those sets, infer the

invariants $\mathcal{I}^{\tilde{\mathcal{P}}}$ and $\mathcal{I}^{\tilde{\mathcal{F}}}$. Let $W = \{p \mid p \in \mathcal{I}^{\tilde{\mathcal{P}}} \land p \notin \mathcal{I}^{\tilde{\mathcal{F}}}\}$, the set of minimal weakening assertions required for $\tilde{r_n}$ to pass. In our example, `n` $\neq$ `0` is a possible weakening assertion for `divRem`.
- For each weakening assertion $p \in W \cup \{$**`false`**$\}$ create a candidate fix that replaces A_n with $A_n \lor p$ and add it to a set of weakening fix candidates Ψ. Dummy assertion **`false`** allows the generation of purely strengthening candidates in the next stage. In our example, we would end up with `n > 0` $\lor$ `n` $\neq$ `0` and `n > 0` $\lor$ **`false`**, which simplify to `n` $\neq$ `0` and `n > 0` respectively.
- Add every weakening candidate fix $f \in \Psi$ that now passes tests $\mathcal{P}_r \cup \mathcal{F}_r$ to Φ_W and let $\Psi' = \Psi \backslash \Phi_W$. In our example, candidate `n` $\neq$ `0` would make all tests pass.

Generating strengthening fixes: Since it is not always possible to fix a contract by simply weakening, this stage combines invalid weakening fixes with strengthening fixes. To keep the program API consistent, it tries to strengthen all pre-conditions in the trace, rather than just the outermost method.

- For every invalid weakening candidate $f \in \Psi'$, determine $\mathcal{I}_i^{\mathcal{P}}$ and $\mathcal{I}_i^{\mathcal{F}}$ as the invariants for the tests currently passing and failing the pre-condition of each method r_i in the trace, with $i < n$. For each r_i, let $S_i = \{p_i \mid p_i \in \mathcal{I}_i^{\mathcal{P}} \land p_i \notin \mathcal{I}_i^{\mathcal{F}}\}$, the set of minimal strengthening assertions that make r_i pass. In our simple example, there is a single method call before the failing assertion, `rem`, and a strengthening assertion such as `n > 0` could be generated associated with the remaining weakening candidate in Ψ (which was actually **`false`**, leaving the pre-condition of `divRem` unchanged).
- For all combinations of strengthening assertions $p \in S_1 \times \ldots \times S_{n-1}$ for a weakening candidate fix $f \in \Psi$, create a fix that replaces P_{r_i} by $P_{r_i} \land p_i$ and add it to the set of strengthening candidates, Σ, along with weakening fix f. For our example, this would create the strengthening candidate for `rem` and leave `divRem` unchanged.
- Add every strengthening candidate fix $f \in \Sigma$ that now passes tests $\mathcal{P}_r \cup \mathcal{F}_r$ to Φ_S.

The fixes presented to the user are those in the set $\Phi_W \cup \Phi_S$.

B. Implementation

The implementation of the contract repair technique is built on top of the Dafny official release[3] which is developed in C#. In particular, our tool relies on the Dafny library to parse Dafny programs and manipulate their AST.

To extract the execution trace of each test case, we rely on a compilation of Dafny into Python. This translation was adapted to wrap the Python-version of the Dafny methods with a decorator that registers the state of the program in a trace logger before and after each function call, including cases when a Python exception occurs (e.g., division by zero and infinite recursion) and this is marked in the trace. *Python.NET* is then used to retrieve the resulting trace information back into the C# module. Note that execution traces are obtained without testing for the contracts, so they always completely run regardless of failing the assertions. This allows us to reuse the same execution traces in different stages of the process.

Once the execution trace of a program is retrieved, it is then enhanced with information indicating whether the contracts are holding at each state. To that purpose, the main C# module evaluates the contracts (i.e., pre- and post-conditions) at each state of the traces. This evaluation is very effective since it is only evaluating an assertion over a specific program state.

Execution traces, along with that contract passing information, are then passed to *Daikon* to infer the dynamic invariants. By having information regarding passing contracts as a trace variables, *Daikon* is able to determine sufficient and necessary conditions for a contract to hold. The header of the *Daikon* trace format, *dtrace*, contains the trace point definitions. Figure 3 shows an example of a declaration for the entering state of `rem` from our running example. It contains the name of the point (with the method's qualified name and argument types, along with information about the point type), its type (either enter or subexit), and for each variable, its name, kind (either field, function, array, variable or return value), how its value is represented in the trace, and its comparability value (*Daikon* only creates comparison invariants for variables with the same comparability value). Here, there are two variables for the `rem` arguments `d` and `n`, and a Boolean `pre_condition`, registering whether the pre-condition passed. An identical variable `post_condition` would occur in exit trace points, together with a variable that indicates if, during its execution, any other contract was broken.

To detect conditional invariants, *Daikon* requires a splitter, which can be manually provided by the user. This is done by indicating which Boolean variables the invariants should be split on depending on their value. For our context, we define the invariant splitter on the variables that identify whether contracts pass or fail, namely on the values of variables `pre_condition` and `post_condition`.

The actual body of a *dtrace* file contains the trace itself, conforming to the point declarations. Each trace point contains concrete values for the variables declared for that point, representing a concrete state of the trace. Currently, our tool supports only variables of primitive types in the states of the traces. Figure 4 shows a particular trace point from the execution of test case for `rem`, identifying the point name, a unique identifier (repeated only in the exit point for the same call), and a list of the point's variable state, with their names, concrete values, and whether they were changed or not. In the example, we have `d = 5`, `n = -3` and `pre_condition = 1` since the pre-condition of `rem` holds for those arguments.

After *Daikon* executes, the provided invariants are processed back into the repair module and converted into Dafny expressions. *Daikon* generates invariants for all entry points and variables, many of which are not relevant to the repair process, so they are filtered before being considered as fix candidates. In particular, an invariant A is considered relevant if:

[3]https://github.com/dafny-lang/dafny

```
ppt module_.default__.rem():::ENTER
ppt-type enter
variable d
        var-kind variable
        dec-type int
        rep-type int
        comparability 2
variable n
        var-kind variable
        dec-type int
        rep-type int
        comparability 2
variable pre_condition
        var-kind variable
        dec-type boolean
        rep-type boolean
        comparability 1
```

Fig. 3. Entry point declaration for `rem` in the *dtrace* format.

```
module_.default__.rem():::ENTER
this_invocation_nonce
192
d
5
0
n
-3
0
pre_condition
1
0
```

Fig. 4. Entry point example for test case `rem(5,-3)` in the *dtrace* format.

- A is in the format `(<contract variable> == <boolean value>) <==> <expression>`;
- the right-hand `<expression>` in A refers to variables other than contract variables;
- A is not structurally equivalent to others;
- when calculating fixes for pre-conditions, A cannot refer to return values;
- in the weakening phase, A cannot refer to methods other than the faulty call r_n.

V. Experimental Evaluation

For the evaluation of the developed prototype, we developed a few simple Dafny programs with faulty contracts:

- **Catalan numbers**, composed by a single recursive method that calculates the n^{th} Catalan number;
- **DivRem**, composed by `divRem` running example that receives two integers and calculates their integer division and respective remainder, which is called by `rem` that discards the quotient and returns the remainder only;
- **Harmonic Sum**, composed by `HarmonicSum`, a method that calculates the sum of the n^{th} and $(n+1)^{\text{th}}$ harmonic terms, which are obtained by a call to `NthHarmonic`, a method that requires a positive number, wrongly disallowing its argument to be 0;
- **Inverse Sign**, having a method `Enter` that receives an integer n and calls `div` which calculates $1/sgn(n)$ using a method `inverse` for the calculation of this division;
- **Opaque Keyword**, having a single empty method with a post-condition that is only valid when its argument is in the interval $[0, 100)$;
- **Two Requires**, having a method `Enter` that calls an empty method with a strict pre-condition, allowing it to only be called with 0 as an argument.

The repair generation process requires a set of test cases from which to retrieve the execution traces. These tests ideally should be generated automatically for the program to be fixed, but that has been left as future work. To have access to a large set of test cases to perform the evaluation, we have manually implemented a script to generate test cases within a certain range for each example program defined above.

The repair procedure was run 20 times for each example and execution times averaged. All tests were run on an EndeavourOS 2022.06.23 machine with 12 GiB of memory and a 2.1 GHz AMD Ryzen 5 4-core CPU. When running *Daikon*, the confidence level was set to 0 in order to obtain a maximum number of invariants, and the splitter was set on the contract passing variables as explained in Section IV-B. Results are presented in Table I, with detailed information about the various stages of the process. Note that the strengthening stage must be run for each invalid weakening candidate in Ψ'; the presented values refer to the sum of all those iterations. After execution, we manually inspected the produced fixes and realized many were actually logically equivalent. To discuss that issue, we also added to the table the number of fixes we found to be effectively *unique* (i.e., not logically equivalent).

Our prototype was able to generate fixes for all the faulty examples except for `opaqueKeyword`. This particular example required a strengthening assertion of the shape `0` $\leq$ `x < 100` but *Daikon* was only able to detect invariant `0` $\leq$ `x` despite our configuration efforts. For `divRem`, `harmonicSum`, and `twoRequires` our tool found both weakening and strengthening fixes, and for the remainder examples only strengthening fixes, which was expected. Interestingly, we also detected a case where the returned strengthening fix was stronger than required, namely for the `divRem` program. Rather than just forcing `n > 0` in the pre-condition of `rem`, the weakest strengthening fix, *Daikon* actually inferred that `n > d`, which associated with the other pre-condition `d` $\geq$ `0` is stronger than `n > 0` (e.g., `d = 9` and `n = 3` is not accepted by the stronger pre-condition).

Regarding execution times, evaluation shows that our tool takes from 6s to 12s to generate the fixes for our examples. While we feel such times are acceptable to a tool of this nature, it must be noted that our benchmark consists of very simple programs. Looking at the different stages of the

TABLE I
EXPERIMENTAL EVALUATION RESULTS.

		divRem	catalanNumber	harmonicSum	inverseSign	opaqueKeyword	twoRequires
Trace	Number of test cases	225	15	15	15	170	15
	Test execution (ms)	1114	1216	1107	1062	1083	1347
Weakening	Weakening fixes generation (ms)	2631	2807	2543	2717	2424	2520
	Invariant detection (ms)	2571	2631	2527	2699	2405	2510
	Invariant detection relative weight	97.7%	93.7%	99.3%	99.3%	99.2%	99.6%
	Weakening candidate fixes ($\#\Psi$)	3	3	3	3	1	2
	Valid pure-weakening fixes ($\#\Phi_W$)	2	0	2	0	0	1
	Unique pure-weakening fixes	1	0	1	0	0	1
Strengthening	Iterations ($\#(\Psi')$)	1	3	1	3	1	1
	Strengthening fixes generation (ms)	2723	8294	2523	8174	2433	2569
	Invariant detection (ms)	2639	8072	2514	8131	2420	2565
	Invariant detection relative weight	96.9%	97.3%	99.6%	99.5%	99.5%	99.8%
	Strengthening candidate fixes ($\#\Sigma$)	6	6	4	12	1	3
	Valid strengthening fixes ($\#\Phi_S$)	2	6	2	12	0	2
	Unique strengthening fixes	1	1	1	2	0	1
Total	Runtime (ms)	5913	12154	6469	15557	6127	7431
	Invariant detection relative weight	88.1%	88.1%	77.9%	69.6%	78.8%	68.3%
	Fixes ($\#(\Phi_W \cup \Phi_S)$)	4	6	4	12	0	3
	Unique fixes	2	1	2	2	0	2

procedure, the evaluation shows that, perhaps as expected, most of the time is spent during invariant detection using *Daikon*, followed by the time spent initially executing the test cases to retrieve the traces. This indicates that future effort should focus on this stage of the process, either fine-tuning the *Daikon* configuration or exploring alternative approaches.

It is also worth noting that the tool is generating equivalent fixes due to equivalent assertions being provided by the invariant detection stage, meaning additional costly, but irrelevant, iterations of the strengthening stage (besides possibly encumbering the user with spurious repair candidates). For instance, for the `divRem` program, *Daikon* returns `n ≠ 0` and `¬(n = 0)` as weakening assertions, and `d < n` and `¬(d ≥ n)` as strengthening assertions. It may also be worth to employ techniques for the detection of equivalent assertions to reduce the number of candidate assertions.

To further illustrate the fixes suggested by the tool, we present in Figure 5 a simple example, the *Harmonic Sum*, used in our evaluation. In this example, the pre-condition of the method `NthHarmonic` incorrectly requires its argument to be a positive number, but a value of zero should also be accepted. The tool proposes four fixes, two for each method. For the `NthHarmonic`, it suggests the weakening assertions `x ≥ 0` and `¬(x ≤ -1)`, both equivalent; for the `HarmonicSum`, it suggests strengthening fixes `n ≥ 1` and `¬(n ≤ 0)`, both also equivalent. Although fixing the `NthHarmonic` by weakening its pre-conditions would likely be the most desirable fix, the suggestions for the `HarmonicSum` also results in a program that verifies and, as such, corresponds to a valid repair.

```
method NthHarmonic(x : int)
  returns (c : int)
  requires x ≥ 1
{
  if x < 0 {
    return 1 / 0;
  }
  return 1 / (x + 1);
}

method HarmonicSum(n : int)
  returns (r : int)
  requires n ≥ 0
{
  var n0 := NthHarmonic(n);
  var n1 := NthHarmonic(n + 1);
  r := n0 + n1;
}
```

Fig. 5. The faulty *Harmonic Sum* example used in the evaluation, with possible contract fix locations highlighted in red.

VI. Conclusions and Future Work

This paper presents a prototype tool for the repair of faulty contracts in Dafny programs, assuming the correctness of a reference implementation. We were inspired by a technique previously proposed for another contract programming language [6], and currently rely on a third-party tool for dynamic invariant inference [8].

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

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

Although previous work suggests that inferred contracts are important to complement contracts written by programmers [33], work on human factors that affect the use of inferred contracts is lacking. Further work in this area should be carried out, for example, to support programmers with determining the relevance and validity of the inferred invariants, as previous studies have shown that users have difficulty deciding if the invariants inferred by *Daikon* are true [34].

Moreover, techniques for automated test generation should be employed to create the test cases, and there is some work on this topic for Dafny [35], [36]. To be useful for the community, such a repair technique should integrate the Dafny IDE, namely its plug-in for VS Code. Lastly, as part of our future work, we aim to build a large dataset of faulty Dafny programs that will be made available to the community to foster further research in the area of Dafny program repair.

References

[1] B. Meyer, *Design by contract*. Prentice Hall Upper Saddle River, 2002.

[2] B. Meyer, "Applying 'design by contract'," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[3] N. Leveson, "Are you sure your software will not kill anyone?" *Commun. ACM*, vol. 63, no. 2, pp. 25–28, 2020.

[4] S. Krishnamurthi and T. Nelson, "The human in formal methods," in *3rd World Congress on Formal Methods - The Next 30 Years (FM)*, ser. LNCS, vol. 11800. Springer, 2019, pp. 3–10.

[5] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, ser. LNCS, vol. 6355. Springer, 2010, pp. 348–370.

[6] Y. Pei, C. A. Furia, M. Nordio, and B. Meyer, "Automatic program repair by fixing contracts," in *17th International Conference on Fundamental Approaches to Software Engineering (FASE)*. Berlin, Heidelberg: Springer, 2014, pp. 246–260.

[7] F. Cassez, "Verification of the incremental Merkle tree algorithm with Dafny," in *24th International Symposium on Formal Methods (FM)*, ser. LNCS, vol. 13047. Springer, 2021, pp. 445–462.

[8] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Sci. Comput. Program.*, vol. 69, no. 1-3, pp. 35–45, 2007.

[9] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Commun. ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[10] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Trans. Software Eng.*, vol. 45, no. 1, pp. 34–67, 2019.

[11] D. Gopinath, M. Z. Malik, and S. Khurshid, "Specification-based program repair using SAT," in *17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. LNCS, vol. 6605. Springer, 2011, pp. 173–188.

[12] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," *IEEE Trans. Software Eng.*, vol. 40, no. 5, pp. 427–449, 2014.

[13] J. McBroom, I. Koprinska, and K. Yacef, "A survey of automated programming hint generation: The HINTS framework," *ACM Comput. Surv.*, vol. 54, no. 8, pp. 172:1–172:27, 2022.

[15] K. Wang, A. Sullivan, and S. Khurshid, "Automated model repair for Alloy," in *33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. ACM, 2018, pp. 577–588.

[14] R. Clarisó and J. Cabot, "Fixing defects in integrity constraints via constraint mutation," in *11th International Conference on the Quality of Information and Communications Technology (QUATIC)*. IEEE Computer Society, 2018, pp. 74–82.

[16] S. G. Brida, G. Regis, G. Zheng, H. Bagheri, T. Nguyen, N. Aguirre, and M. F. Frias, "Bounded exhaustive search of Alloy specification repairs," in *43rd IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1135–1147.

[17] G. Zheng, T. Nguyen, S. G. Brida, G. Regis, N. Aguirre, M. F. Frias, and H. Bagheri, "ATR: Template-based repair for Alloy specifications," in *31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2022, pp. 666–677.

[18] J. Cerqueira, A. Cunha, and N. Macedo, "Timely specification repair for Alloy 6," in *20th International Conference on Software Engineering and Formal Methods (SEFM)*, ser. LNCS, vol. 13550. Springer, 2022, pp. 288–303.

[19] R. Abreu, A. González, P. Zoeteweij, and A. J. van Gemund, "Automatic software fault localization using generic program invariants," in *2008 ACM Symposium on Applied Computing (SAC)*, 2008, pp. 712–717.

[20] T. Ball, "The concept of dynamic analysis," *SIGSOFT Softw. Eng. Notes*, vol. 24, no. 6, pp. 216–234, 1999.

[21] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Trans. Software Eng.*, vol. 41, no. 5, pp. 507–525, 2015.

[22] J. W. Nimmer and M. D. Ernst, "Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java," in *1st Workshop on Runtime Verification (RV@CAV)*, ser. Electronic Notes in Theoretical Computer Science, vol. 55, no. 2. Elsevier, 2001, pp. 255–276.

[23] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin, "Quickly detecting relevant program invariants," in *22nd International Conference on on Software Engineering (ICSE)*. ACM, 2000, pp. 449–458.

[24] L. Zhang, G. Yang, N. Rungta, S. Person, and S. Khurshid, "Feedback-driven dynamic invariant discovery," in *2014 International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2014, pp. 362–372.

[25] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna, "Toward automated detection of logic vulnerabilities in web applications," in *19th USENIX Security Symposium (USENIX Security)*. USENIX Association, 2010, pp. 143–160.

[26] J. Cobb, J. A. Jones, G. M. Kapfhammer, and M. J. Harrold, "Dynamic invariant detection for relational databases," in *9th International Workshop on Dynamic Analysis (WODA@ISSTA)*. ACM, 2011, pp. 12–17.

[27] S. Grant, H. Cech, and I. Beschastnikh, "Inferring and asserting distributed system invariants," in *40th International Conference on Software Engineering (ICSE)*. ACM, 2018, pp. 1149–1159.

[28] H. Jiang, S. Elbaum, and C. Detweiler, "Inferring and monitoring invariants in robotic systems," *Auton. Robots*, vol. 41, no. 4, pp. 1027–1046, 2017.

[29] Y. Liu and Y. Li, "InvCon: A dynamic invariant detector for Ethereum smart contracts," in *37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022, pp. 160:1–160:4.

[30] C. Csallner, N. Tillmann, and Y. Smaragdakis, "DySy: Dynamic symbolic execution for invariant inference," in *30th International Conference on Software Engineering (ICSE)*. ACM, 2008, pp. 281–290.

[31] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *24th International Conference on Software Engineering (ICSE)*. ACM, 2002, pp. 291–301.

[32] S. Hangal, N. Chandra, S. Narayanan, and S. Chakravorty, "IODINE: A tool to automatically infer dynamic invariants for hardware designs," in *42nd Design Automation Conference (DAC)*. ACM, 2005, pp. 775–778.

[33] N. Polikarpova, I. Ciupa, and B. Meyer, "A comparative study of programmer-written and automatically inferred contracts," in *18th International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2009, pp. 93–104.

[34] T. W. Schiller and M. D. Ernst, "Reducing the barriers to writing verified specifications," in *27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2012, pp. 95–112.

[35] P. Spettel, "Delfy: Dynamic test generation for Dafny," Master's thesis, ETH Zurich, 2013.

[36] A. Fedchin, T. Dean, J. S. Foster, E. Mercer, Z. Rakamaric, G. Reger, N. Rungta, R. Salkeld, L. Wagner, and C. Waldrip, "A toolkit for automated testing of Dafny," in *15th International Symposium on NASA Formal Methods (NFM)*, ser. LNCS, vol. 13903. Springer, 2023, pp. 397–413.