Towards Progressive Program Verification in Dafny

Ismael Figueroa
Escuela de Ingeniería Informática
Pontificia Universidad
Católica de Valparaíso
Valparaíso, Chile
ismael.figueroa@pucv.cl

Bruno García
Escuela de Ingeniería Informática
Pontificia Universidad
Católica de Valparaíso
Valparaíso, Chile
bruno.garcia.a@mail.pucv.cl

Paul Leger
Escuela de Ciencias Empresariales
Universidad Católica del Norte
Coquimbo, Chile
pleger@ucn.cl

ABSTRACT

Program verification is a tool for the development of software that is free from defects and satisfies its functional specification. It suffers from two issues that have already been addressed in the field of type systems. First, it has a rigid focus on full-program verification. Also, it provides weak support for "partial" verification, relying on unproven lemmas that may lead to unsound runtime behavior. These issues are also present in Dafny, a verification-first programming language with full support for expressive pre-conditions and post-conditions that must be verified statically. Inspired by recent research on the field of *gradual program verification* in the Coq proof assistant, in this paper we present a first proposal for *progressive verification* in Dafny by presenting a minimal proof-of-concept implementation. Progressive verification means that programmers can use a new keyword to specify post-conditions that are assumed to be true during verification, but that will be tested at runtime by automatically-generated checks. We argue that our approach is correct by construction, by relying on the architecture of Dafny itself, and we also discuss several issues regarding the threats to the proposed approach as well as perspectives for further development.

CCS CONCEPTS

• **Theory of computation → Program verification**;

KEYWORDS

program verification, Dafny, progressive verification

ACM Reference Format:
Ismael Figueroa, Bruno García, and Paul Leger. 2018. Towards Progressive Program Verification in Dafny. In *XXII Brazilian Symposium on Programming Languages (SBLP 2018), September 20–21, 2018, SAO CARLOS, Brazil.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3264637.3264649

1 INTRODUCTION

The specification, implementation and verification of *correct* software is part of the never-ending labor of software engineering. In essence, there are two main tasks [2]: *specification*, which amounts to declare *what it means for the software to be correct*, and *verification*, where we must decide whether a given implementation *satisfies the specification.* In the field of programming languages, and more specifically in the context of language-based verification [29], we find that type systems [24] and deductive program verification (DPV) [14] are designed as systems that use source-level annotations to specify correctness properties, which are then checked by decision procedures based on expressive logical systems.

On one hand, deductive program verification asserts the functional correctness by means of pre- and post-conditions expressed in a *program logic*, where source-code entities are bound to logical specifications. Then, the logical entailment of those conditions results in the formal proof that the implementation satisfies the specification. Nevertheless, despite its usefullness, DPV still suffers from two issues that are mostly solved in the field of type systems:

- Current approaches to DPV are too rigid, focusing mostly on full-program verification [29]. This requires a binary choice between the upfront development, specification and verification of a whole program, which is indeed a costly thing to do, or the almost complete lack of specification and verification.
- In cases that support "partial" verifications, this is usually done by relying on unproven axioms or lemmas—assumed to always hold—but that may easily introduce unsound behavior at runtime.

These issues reflect the traditional, conservative, but also *punitive* approach to sound verification. Indeed, users must be extra careful when using any kind of escape mechanism, and must resort to external reasoning to re-establish theoretical confidence in their implementation. On the other hand, although type systems have faced similar issues to those described above, the active research on *gradual typing* [30] has been quite succesful in managing the tension between conservative static guarantees and optimistic assumptions that are checked as runtime assertions. Indeed, recently Tanter and Tabareau [35] devised a simple mechanism for *gradual certification* in the Coq proof assistant [36], whereby unproven but *decidable* axioms are translated into runtime checks upon program extraction.

Based on the developments within the field of type systems, and inspired by Tanter and Tabareau [35], in this paper we propose a mechanism for *progressive verification of programs* in the context of the Dafny [20] language. Similar to [35], in our work we introduce a new **assures** specification to specify *executable post-conditions*, similar to those in E-ACSL [11], that must be verified at runtime, but that can be used for static reasoning during program verification.

SBLP 2018, September 20–21, 2018, SAO CARLOS, Brazil
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6480-5/18/09.
https://doi.org/10.1145/3264637.3264649

To make our work tangible we have chosen to extend the Dafny [20] programming language with **assures**, in order to test the practical consequences of our approach.[1] Overall, we believe this is a non-trivial research and development challenge given that:

- Dafny supports several programming features not inmediately present in Coq: imperative and mutable state, classes, objects, references, among others.
- In constrast to Coq, which has explicit support for specifying decidable properties, in Dafny we need a syntax-based approach that guarantees that a logical formula can be translated into a corresponding runtime check.

Consequently, our proposal introduces the idea of progressive program verification in a practical verification-aware programming language. We use the term *progressive*—instead of *gradual*—because the latter has a specific technical meaning in the state of the art [3, 31], which does not correspond to the approach presented here.

We also want to remark that choosing Dafny was not arbitrary: Dafny provides a complete environment for programming and verification, with a special focus on making verification *accessible* to programmers and students [21]. We share this lofty goal and we envision the introduction of progressive verification as a way to further ease the introduction of program verification to programmers. This paper reports our current progress towards progressive verification in Dafny, making the following specific contributions:

- Proposes the concept of *progressive verification* inspired by developments in the field of gradual typing and verification.
- Presents a minimal proof-of-concept implementation in Dafny, where post-conditions can be marked as dynamically-checked by using the **assures** keyword.
- Presents an argument and perspective about the challenges for establishing the formal correctness of the approach, as well as an in-depth discussion of the limitations, threats to validity and perspectives for the development of the approach.

The rest of this paper is as follows: we first present a brief background on the core concepts of deductive program verification (Section 2) to then present our contribution by means of examples in Dafny itself (Section 3). Afterwards, we present a thecnical overview (Section 4), to then present a discussion regarding correctness and other open questions and issues (Sections 5 and 6). Finally, we relate our work to the relevant state-of-the-art (Section 7) and conclude with a discussion about current and future work (Section 8).

2 DEDUCTIVE PROGRAM VERIFICATION

Software quality is a crucial and desirable property of software, not only from the academic point of view, but also from an economic standpoint [32]. While there are many approaches to software quality, such as processes and methodologies [26, 34] and test-based approaches [16, 23], in this paper we focus on the field of ***language-based software verification*** **(LBSV)** [2, 29]. Furthermore, we focus on two definitions of quality: *functional correctness* and *freedom from defects*. The former refers to a system that indeed does what it is supposed to do, by means of a *formal specification*; the latter asserts that the software will not present defects such as unexpected program crashes, information leaks, or similar defects.

The essential characteristic of LBSV is that ***the source code of the programs is used as the mathematical object under study***. Hence, the source code itself is used to provide formal proof or evidence that the software indeed completely avoids certain kinds of errors. This is in contrast to other formal techniques, such as, for instance, model checking. Within LBSV we found several disciplines such as type systems, static or dynamic contracts, as well as deductive program verification. This paper focuses on the latter.

Deductive Program Verification (DVP) [14, 20] is the process of stating program functional correctness as a set of logical statements that are then proved, with as much automation as possible. As a language-based approach to program verification, DPV relies on program annotations that bind specific language constructs and expressions to their logical specifications. Such annotations may be implemented not only through special comments on the source code, as in ACSL [6], JML [1], but also as in Dafny where it is an ingrained part of the programming language itself [20].

The expressive power of DPV lies on the *program logics* involved in the specification of the source code constructs, which determine the kind of reasoning and the extent of properties that can be proven. The modern approach is to use *separation logics* [2], which allows developers to specify and reason about variables, objects, pointers, mutable data-structures, and other entities used in most imperative and object-oriented languages. Separation logics requires the annotation of *pre-conditions* and *post-conditions* bound to a given command or instruction. Preconditions must hold before executing the instruction, whereas the correct execution of the instruction must fulfill the post-conditions. Hence, a verified program is that in which it is possible to demonstrate the logical entailment of all pre- and post-conditions, for all the sequential commands in a program. Separation logics also allows developers to reason about memory locations in the conditions, the command, and the rest of the heap.

The logical formulas allowed into the conditions—hence the expressive power of DPV—are determined by a fragment of first-order logic alongside *logical theories* for supporting reasoning about bitvectors, arrays, arithmetic, and other models relevant to computing. In general, reasoning is automated using a *Satisfiability Modulo Theory solver* [5]—*SMT solver* for short—which is a specialized software for checking satisfiability of logical formulas in the context of a set of given theories. The general processing pipeline of DPV is composed of at least the following three steps:

1. **Parsing and Type Checking:** the standard stages of a compiler, where an abstract syntax tree (AST) is parsed from source code, and is typechecked according to the rules of the actual programming language. Most notably, specification formulas are checked, as they must have boolean type.
2. **Verification:** the typed AST is inspected and all the specifications, that is the pre- and post-conditions, are collected as a *set of verification conditions* that must be satisfied. The set of verification conditions is given to a SMT solver. This step is successful when all conditions are satisfied.
3. **Translation/Execution:** after successful verification the typed AST is translated for its eventual execution. The translation may be into machine code, another language such as

[1]Implementation and benchmarks available at: http://zeus.inf.ucv.cl/~ifigueroa/doku.php/research/progressive-verification-dafny

```
method Find(a: array<int>, key: int)
  returns (index: int)
  requires a != null
  ensures index == -1
     ==> forall k :: 0 <= k < a.Length
     ==> a[k] != key
  ensures index >= 0 && index < a.Length
     ==> a[index] == key
  {
    // ... user-defined code,
    // which might fulfill the specification
  }
```

Figure 1: Specification of `Find` in terms of pre-conditions and post-conditions, using requires and ensures respectively. During compilation Dafny will try to verify that the implementation actually fulfills the specification.

```
method Find(a: array<int>, key: int)
  returns (index: int)
  requires a != null
  ensures index == -1 ==> ...
  ensures index >= 0 && index < a.Length  ==> ...
  {
    var i := 0;
    while i < a.Length {
      if a[i] == v {
        return i;
      }
      i := i+1;
    }
    return -1;
  }
```

Figure 2: Implementation of linear search, without verification hints. This code does not compile as Dafny is not able to deduce that the postconditions actually hold.

a C or even C#, or to virtual machine bytecode, for instance for the LLVM or JVM. In general, type and verification information is erased for performance reasons, hence it is not available at runtime.

3 PROGRESSIVE VERIFICATION BY EXAMPLE

Dafny [20] is an imperative, sequential, class-based programming language with first-class support for the specification of safety and functional correctness properties. Its specification language features: pre- and post-conditions, heap assertions, and reasoning about program termination. In its implementation, Dafny closely follows the aforementioned DPV pipeline (Section 2): after parsing and typechecking, the compiler extracts a set of verification conditions stated in the Boogie [4] intermediate verification language. These conditions are then solved using Microsoft's Z3 SMT solver [10]. Regarding execution, Dafny programs are compiled

```
method Find(a: array<int>, key: int)
  returns (index: int)
  requires a != null
  ensures index == -1 ==> ...
  ensures index >= 0 && index < a.Length ==> ...
  {
    var i := 0;
    while i < a.Length
     invariant 0 <= i <= a.Length
     invariant forall k :: 0 <= k < i
               ==> a[k] != v
    {
      if a[i] == v {
        return i;
      }
        i := i+1;
    }
    return -1;
  }
```

Figure 3: Implementation of linear search that correctly uses loop invariants to show that the post-conditions hold.

```
method Find(a: array<int>, key: int)
  returns (index: int)
  requires a != null
  assures index == -1 ==> ...
  assures index >= 0 && index < a.Length  ==> ...
  {
    var i := 0;
    while i < a.Length {
      if a[i] == v {
        return i;
      }
      i := i+1;
    }
    return -1;
  }
```

Figure 4: Implementation of linear search using progressive verification with the assures keyword. This code compiles and can be composed with existing methods.

into the C# language, which enables the interoperation of verified modules with other elements in the .NET platform.

As our running example, let us consider the code fragment in Figure 1, adapted from the Dafny tutorial [28]. The code is annotated using the proper Dafny constructs. Notice the use of the standard **requires** and **ensures** keywords to specify pre- and post-conditions respectively. The given specification requires the array to be valid, *i.e.* not null, whereas the post-conditions consider two mutually-exclusive scenarios:

(1) **The value is not found and the method returns `-1`.** In this case, the method ensures that none of the elements in the array is equal to the value being searched.

(2) **The value is found and the method returns the `index` value**. In this case, the method ensures that `index` is within valid bounds, considering the length of the array, and that element `a[index]` actually holds the value being searched.

As usually taught in first year programming courses, linear search is the most straightforward implementation for method `Find`. Now let us consider the almost-classic implementation in Figure 2 that any aspiring programmer should be able to produce. Unfortunately, even though we intuitively know that this implementation is actually *correct*—which it is—Dafny is still unable to automatically derive this just from the source code.[2] At this point programmers who are not expert in the field of verification may find that the only "solution" is to discard the post-conditions altogether, losing all the benefits of function contracts.

On the other hand, motivated programmers should learn more about how to verify their algorithms in Dafny, which could eventually lead them to the correct implementation, shown in Figure 3.

We see that in terms of line of code the difference is minimal, however the time and effort required from the programmer might be considerable. We also see that there is no middle ground: either we fully specify and correctly implement—with all verification hints and tricks—or we give up on the use of function contracts altogether. Here is where our approach to progressive verification comes in handy. From a programmer's point view, progressive verification enables one to use the **assures** keyword to defer the check to runtime, as shown in Figure 4.

4 TECHNICAL OVERVIEW

We now provide a general overview of the technical aspects in the current state of our implementation. We first describe the processing pipeline of Dafny, making references to the essential classes involved in our work. Then we outline the specific changes required for the current implementation of **assures**.

Dafny Processing Pipeline. Upon invoking the Dafny compiler, the `DafnyDriver` class controls the processing of all involved files. First, it parses and checks all the command line arguments, and it also performs some sanity checks on the file names. As a result, a `files` variable contains the paths to all (valid) files involved in the build. In the second step, the `DafnyMain.Parse` method is called to perform the parsing and typechecking steps. When this is successful, a variable `dafnyProgram` holds the internal representation of the code. This variable is an instance of `Dafny.Program`, which in turns relies on the parser and scanner that are automatically generated from the `Dafny.atg` file; this file is written in the syntax of the Coco/R compiler generator [15]. The next step is verification, by invoking the `DafnyDriver.Boogie` helper method that manages the translation and execution of the SMT solving phase through the Boogie intermediate language. Finally, in the compilation step, `dafnyProgram` is translated into a C# program by following a standard inductive algorithm on the structure of the syntax tree.

[2]In order to focus on our actual contribution, we work on an archived version of Dafny 2.0.0.00922, as it provides a static target for our efforts. Current improvements may actually be able to prove this particular method, nevertheless the example serves to illustrate our point.

```
method _Find__ass_(x: array<int>, v: int)
  returns (y: int)
  requires x != null
  decreases x, v
{
  var i := 0;
  while i < x.Length
    decreases x.Length - i
  {
    if x[i] == v {
      return i;
    }
    i := i + 1;
  }
  return -1;
}

method Find(x: array<int>, v: int) returns (y: int)
  requires x != null
  decreases x, v
{
  y := _Find__ass_(x, v);
}
```

Figure 5: Simple method wrapping for methods using assures. The original source code goes into the hidden method `_Find__ass`, which is invoked by the newly facade method `Find`. On compilation, we generate the runtime checks after the invocation of `_Find_ass_`.

Implementation of assures. In general terms, implementing a new feature in the specification language is a hugely crosscutting task. It requires changes at the syntax level, at the program AST representation level, and then at the semantic level. Fortunately, given the solid design of the Dafny implementation, so far we only had to modify 4 source files/classes:

- `Dafny.atg`: for adding the **assures** keyword to the syntax. This generates the proper parser and scanner files.
- `Dafny.Program`: for adding an internal representation in which methods contain a collection of *assured* post-conditions. For now this just replicates what is already done for post-conditions defined with **ensures**.
- `Dafny.Resolver`:contains the code for semantic checking, *e.g.* typechecking and other additional checks. Here we only check that **assures** expressions must be of boolean type, again mimicking what was already the case for **ensures** expressions.
- `Dafny.Compiler`: here we modify the translation of methods with **assures** post-conditions, in order to include runtime checks for each of them.

Our approach consists in a simple form of method wrapping, in which the parser generates a *normalized shell method* in which runtime checks are generated during compilation time. In Figure 5 we show how this process works for the code in Figure 4.

We borrow our implementation strategy from E-ACSL [11], hence the source method `Find` generates two new methods. First,

```
public static void @Find(
   BigInteger[] @x
   BigInteger @v,
   out BigInteger @y)
  {
    @y = BigInteger.Zero;
    TAIL_CALL_START: ;
    BigInteger _out0;
    @__default.@__Find____ass__(@x, @v, out _out0);
    @y = _out0;
    // POSTCONDITION CHECK HERE
    if(! (!((@y).@Equals(/* condition check */)))) {
      System.Console.Write("Postcondition failed");
    }
    if(! (!(((@y) >= (new BigInteger(0))) &&
      ((@y) < (new BigInteger((@x).@Length)))) ||
      (((@x)[(int)(@y)]).@Equals(@v)))) {
      System.Console.Write("Postcondition failed");
    }
  }
```

Figure 6: Simplified extract of the C# code generated after compiling from Dafny sources. The code includes the call to the hidden `_Find__ass` **method as well as the actual runtime checks generated from the post-conditions. For space reasons we omit the long condition check in the first** `if` **statement.**

the hidden method `_Find__ass_` with the original implementation from the source code. Then, a new `Find` method, used as a facade that simply invokes `_Find__ass_`. This method is instrumented upon compilation, as shown in Figure 6. Its structure is always the same: an invocation to the hidden method, whose values are return in as many variables as necessary. We keep the names of return variables, because they must in scope for the generated runtime checks. The parsing and checking stage already checks the naming of post-conditions to avoid scenarios with incorrect scoping.

5 TOWARDS CORRECTNESS OF PROGRESSIVE VERIFICATION

After showing the motivation and illustrative examples for progressive verification, we now present a discussion on the crucial issue of formal correctness. First we describe how verification correctness is established in standard, *i.e.* non-progressive, Dafny programs. As we explain below, we argue this is obtained by a correct-by-construction approach. Then, we present an argument about how the correctness of progressive Dafny programs is obtained, which depends on the correct translation of the logical specifications inside the **assures** clauses.

Correctness of standard Dafny programs. In a standard Dafny program, correctness is established by the following:

- A type-checked and valid Dafny program is transformed into a program in the Boogie specification language.
- The Boogie program is sent for the verification to the SMT solver, which extracts and tries to demonstrate a set of validation constraints.
- Upon successful verification, the Dafny program is compiled into the C# target language, erasing all verification-related information, such as ghost variables and other constructs.

Hence, the correctness of the standard verification of Dafny programs relies on the following:

(1) The translation from Dafny to Boogie must be sound. All Dafny programs must be expressable as Boogie programs, and all verification conditions must be equivalent in the target Boogie program.
(2) The extraction of verification constraints and its verification must be equivalent to the verification of the conditions in the Dafny source file.
(3) The compilation from Dafny to C# also must be sound; it should not introduce additional operations that may affect the verifications already performed.

To the best of our effort, we have not managed to find a formalized core calculus for Dafny that formally defines these correctness properties. Indeed, the seminal papers that describe Dafny [20, 21] directly refer to the implemented language and its application. Despite this situation, we hold to the hypothesis that Dafny is correct by construction, and by extensive testing, although the implementation itself is not formalized yet.

Correctness of progressive Dafny programs. Considering all of the above, the question is how we can guarantee the correctness of progressive verification in Dafny, at least up to the same confidence attained for standard Dafny programs. Based on the current state of our work, we present the following argument for the correctness of our approach:

- Regarding the validity of progressive Dafny programs, all **assures** expressions are constructed in the AST in the same way as the **ensures** expressions are. This includes the syntactical validity, as well as the correct typechecking of the specifications as boolean expressions.
- The translation from Dafny to Boogie is left unmodified, thus we are not introducing any errors or incorrect behaviors that may affect the correctness of verification. In principle, the parser should introduce axioms for each assured post-condition, which would also be fed to the verifier and would allow static reasoning based on unverified assumptions. We are working currently in this point. This and other limitations are discussed in Section 6.
- The compilation from Dafny to C# is mostly left unmodified. Our changes only come into play on the compilation of methods that have one or more **assures** expressions: here we use an internal method of Dafny itself to translate the boolean specifications into executable runtime checks.

Correctness of executable specifications. Following this argument, the most crucial step is the compilation of boolean expressions into runtime checks. There are two important problems that must be addressed:

(1) Can the logical specification be translated into a boolean source-level expression?
(2) Is it possible to actually *execute* the translated boolean expression in a manner equivalent to the logical specification?

This is exactly the problem of the correctness of executable specifications, such as those presented in E-ACSL. Indeed, we take the approach of E-ACSL to solve both problems. For the first issue we observe that the specification-level language is equivalent to the boolean source-level language. This comes from the fact that in the Dafny implementation all specifications must be correctly typed as boolean expressions in order for the program to be valid, and these boolean expressions are written in the same syntax of all other parts of the program.

The second issue is a bit more complex. For instance, let us consider the assured expression of the Find method (Figure 4):

```
index == -1 ==>
      forall k :: 0 <= k < a.Length ==> a[k] != key
```

Intuitively, this can be checked at runtime by executing the following code fragment:

```
bool b = (index == -1);
for(int k = 0; k <= a.Length; k++) {
   b = b && a[k] != key;
} // now b holds the result of this check
```

However, executions like these are not possible for unbounded integer ranges, nor for any range of float numbers, due to the inability to enumerate them. For instance, the following expressions are logically true, but are not executable:

```
// k has no upper bound, execution would never finish
forall k :: 1 < k ==> k*k > k
// cannot enumerate range of float numbers
forall x :: 0.0 <= x < 1.0 ==> x*x < x
```

Guarded specification expressions. Following E-ACSL [11], we specify a restricted subset of logical expressions that can always be translated into executable boolean expressions. We term them as *guarded specification expressions* (GSE). GSEs are inductively defined on the syntax of the language itself as follows:

- Any expression without quantification is a GSE if all its sub-expressions are also GSE.
- An expression with universal quantification is a GSE if and only if the quantifier is guarded.
- A universal quantifier is guarded if and only if it involves a bounded integer range. Such a range is one that is syntactically equal to an expression such as `lb < ... < ub` or such as `ub > ... > lb`, where `ub` and `lb` are respectively the upper and lower bounds. The dots denote any number of integer values or variables that are chained in the expression.
- For now we do not support existential quantification, but the same restrictions regarding bounded ranges apply.

We advocate for the correctness of our approach by exploiting the correctness of Dafny itself in addition to the correct specification of executable specifications, which follows from the inductive translation of formulas to executable code, under the restriction of guarded quantification.

6 DISCUSSION

We now discuss several issues about progressive verification, the current limitations of our implementation, what are the threats to the validity of the approach, as well as several perspectives and open issues that are yet to be resolved in future work.

Limitations of the current implementation. Our current implementation serves as a minimal proof-of-concept that introduces the ideas behind progressive verification in Dafny. A first limitation is that **assures** is only supported for post-conditions in function contracts, when it could also be introduced in other places such as invariant declarations, or even function pre-conditions. Another point which we are addressing currently is the introduction of the assured conditions as unproven lemmas or axioms for the Boogie program created in the verification step. This is crucial because it enables the integration of assured expressions in the realm of the static verification reasoning. We also do not support the use of predicates or lemmas in the body of the assured post-conditions, although they are mechanisms that enable the modular specification of programs. As a consequence of these limitations, we are not yet able to translate the whole Dafny test suite for the proper performance evaluation of **assures**. Nevertheless, although we recognize that our results are very preliminar, we consider that the core of the proposal is faithfully reflected in this paper, and that the aforementioned limitations do not pose any fundamental challenge beyond the engineering work required to implement them.

Threats to validity. There are important threats to the validity of progressive verification that must be properly addressed in upcoming work. The main issue is how to establish the formal correctness of the approach. As outlined before in Section 5, we rely on the correct implementation of Dafny itself. However this is not necessarily sufficient, as there is a need for a proper formalization and certification of the correctness of the system. This in itself is quite a challenge, because, in constrast to the previous work in Coq [35], which uses Coq itself for the certification, we must investigate the proper theoretical approach for an object-oriented language with verification such as Dafny. Other threats to validity arise from the potential adoption of progressive verification by developers. For instance, it is known that industry is moving towards the use of optional types, *e.g.* as in Dart [9] or MyPy [19], which do not incur any costs for runtime verification. In contrast, the performance of gradual typing and related approaches is a real concern for practical adoption. Yet another threat is whether the cost of learning how to verify programs in standard Dafny is high enough to justify the hybrid approach presented in this work.

Perspectives and Open Issues. There are several open issues that must be addressed during the development of progressive verification. A first issue is deciding whether the use of **assures** must be left to the programmer, or if it is more conveniente to work towards the development of some adaptive verification system that, based on the set of verification constraints that could not be statically proven, generates the necessary runtime checks. This is a very attractive option that opens interesting challenges in the field of SMT-based constraint verification. An additional problem is how to assess the expresiveness of the restricted specification language for the executable conditions. Although we took the same approach as in E-ACSL, there could be room for improvements that allow for sound approximations for instance for unbounded ranges, or for dealing with float values. Another issue is related to the performance penalties that can be incurred due to the runtime checks, and whether or not we should allow for those checks to be turned off during production builds of the software, because the software can potentially run into unsound behavior that was not addressed by static verification. Indeed, it is interesting to determine how progressive verification can serve as a pay-as-you-go mechanism to balance the tradeoff between verification effort, performance, and developer satisfaction. Finally, another interesting issue is how to evaluate the impact of progressive verification in terms other than runtime performance such as, for instance, programmer adoption, quality, quantity and complexity of the code written with and without progressive verification, and other qualitative factors regarding developer productivity. Given that our main motivation is to make program verification accessible to a broader group of developers, we must empirically study and address these concerns.

7 RELATED WORK

There is a vast literature on program verification and the variety of techniques used for this purpose. We now we briefly overview several related developments that influence our work, and that we consider relevant to correctly contextualize our contributions.

Gradual typing. The idea of graduality arised in the seminal work of Siek and Taha [30], where the authors introduce the dynamic type "?". During typechecking, this type may correspond to any arbitrary and *consistent* type. Consistency is a relation for finding *plausible* types that could replace "?" to form a statically-typed program. The main philosophy of gradual typing is subsumed by the motto *trust but verify*, meaning that programs that *might be correct at runtime* are not rejected by the typechecker. Instead, the compiler inserts runtime checks to ensure the safe execution of the code. Although there is a performance tradeoff, due to the introduction of runtime checks, the impact should only be proportional to the usage of the dynamic type "?".

Gradual Certified Programming. Based on gradual typing, Tanter and Tabareau [35] proposed a mechanism for *Gradual Certified Programming in Coq.* Coq [36] is a widely-used proof assistant, based on a highly expressive functional programming language featuring *dependent types* [25], which are used both for programming and for proving. In addition, Coq has *program extraction* features, in which a certified program is translated into OCaml. Then, the core idea of gradual certified programming is to enable developers to introduce *conjectures* in the proofs. A conjecture is a decidable property that is not yet proven, but will be assumed to be true, in order to construct formal proofs. Being decidable, they can be transformed into runtime checks upon extraction.

Java Modelling Languages. Although our work is inspired in part by the executable conditions present in E-ACSL, the seminal work on modelling and specification languages is due to the Java-based modelling languages such as JML [1], JCML [33] and AspectJML [27]. Indeed, JML is cited as the main inspiration of the ACSL specification language [6] itself. These modelling languages define a formal syntax for the specification of function contracts, through special comments that can be parsed and analyzed alongside the original source code. AspectJML uses the paradigm of aspect-oriented programming [17] to further modularize the implementation or realization of the contract mechanisms.

ACSL and Executable ACSL. ACSL [6] is a specification language for the C programming language that is heavily inspired by the design of JML. Like JML, ACSL is a formal language for the specification of function contracts. Due to the nature of the C language, ACSL allows developers to write specification for low-level features, such as memory accesses, as well as higher-level behavior such as return values or rich conditions for data structures such as linked lists. ACSL is a central part of Frama-C [8]: a rich and extensible framework for the analysis of C programs. The E-ACSL language is a subset of ACSL that introduces a set of restrictions to make the specifications *executable*. In practice this requires the use of *guarded quantification* when using ∀ and ∃ quantifiers in the specification formulas. Guarded quantification requires bounded and well-defined intervals for integer ranges, which can then be translated into C code. The complete specification of E-ACSL is more complex as it also introduces a memory monitoring system, and an automatic translator to C, which our work currently does not address. To the best of our knowledge, although E-ACSL could be used for a progressive verification mechanism as shown in our work, we are not aware of any systematic effort to do so. Indeed, a key difference between ACSL and Dafny is that, as stated in its website, "ACSL allows you to write complete specifications. But it does not force you to", while Dafny is stricter about this.

Design by Contract. The Design-by-Contract methodology [22] was introduced in the context of the Eiffel language. The notion of a contract, which is commonly understood nowadays, consists in the specification of pre-conditions and post-conditions, as well as code invariants. Eiffel features dynamic verification of contracts, which can be turned off to avoid the performance impact of the dynamic checks. Compared to Eiffel, Dafny features a mechanism for static contracts, powered by the SMT-solving machinery, but the specification of conditions is quite reminiscent, if not the same, as in Eiffel and many other languages with contract mechanisms.

Embedded Contracts. Embedded Contracts [13] is another language-based approach for the specification of contracts, that uses the programming language itself as the means to write the contract specifications. Overall, this enables the reuse of all the infrastructure regarding IDEs and tools around the language to be used also for the support of contracts. In [13] these embedded contracts are extracted and transformed into runtime checks, even though they discuss potential strategies for static checking of contracts. What we do is quite similar, by extracting the annotated postconditions and turning them into a runtime check. However one difference in our approach is that the post-condition is also meant to be available to the SMT solver for static verification reasoning.

Gradual Refinement Types. Refinement types [7, 37] are an extension of type theory that decorates basic types with logical predicates that can be collected and verified by an SMT-solver. As an extension of refinement types [18] defined *gradual refinement types*, allowing the use of *gradual formulas* with incomplete information that must be checked at runtime. Our work is not directly related to gradual refinement types, as we have only implemented a dynamic post-condition checking.

Gradual Program Verification. Influenced by the developments in gradual typing research, [3] defines a formal proposal for gradual program verification. In their context, gradual verification also includes the notion of incomplete formulas, which are optimistically filled with plausible and verifiable facts, whose checking is deferred at runtime. In contrast to this work, we do not consider yet the idea of imprecise formulas. Finally, although this work is deeply theoretical in nature, we believe it is also an important reference for setting the direction of future work.

8 CONCLUSIONS AND FUTURE WORK

Borrowing from recent developments in the fields of gradual typing and gradual program verification we present a first approach towards what we call *progressive verification* in the context of the Dafny language. The core idea is to make program verification more accesible to programmers trained in the traditional imperative and object-oriented languages such as a Java or C#. This is done by introducing the **assures** keyword that allows developers to specify post-conditions that are assumed to be true, and thus can be used for static verification reasoning, but that will be checked at runtime. Although the idea in itself is not novel in itself, we believe there is a gap to be filled: the need for a practical, accesible platform for developers with all levels of expertise in program verification.

REFERENCES

[1] 2018. The Java Modeling Language (JML). https://www.eecs.ucf.edu/ leavens/JML/index.shtml.

[2] José Bacelar Almeida, Maria João Frade, Jorge Sousa Pinto, and Simão Melo de Sousa. 2011. *Rigorous Software Development: An Introduction to Program Verification* (1st ed.). Springer-Verlag.

[3] Johannes Bader, Jonathan Aldrich, and Éric Tanter. 2018. Gradual Program Verification. In *Proceedings of the 19th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2018) (Lecture Notes in Computer Science)*, I sil Dillig and Jens Palsberg (Eds.), Vol. 10747. Springer-Verlag, Los Angeles, CA, USA, 25–46.

[4] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2006. *Boogie: A Modular Reusable Verifier for Object-Oriented Programs.* Springer Berlin Heidelberg, Berlin, Heidelberg, 364–387. https://doi.org/10.1007/11804192_17

[5] Clark W Barrett, Roberto Sebastiani, Sanjit A Seshia, Cesare Tinelli, et al. 2009. Satisfiability modulo theories. *Handbook of satisfiability* 185 (2009), 825–885.

[6] Patrick Baudin, Pascal Cuoq, Jean-Cristophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. 2018. The ANSI/ISO C Specification Language (ACSL). https://frama-c.com/acsl.html.

[7] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. 2011. Refinement types for secure implementations. *ACM Transactions on Programming Languages and Systems* 33, 2, Article 8 (Jan. 2011), 8:1–8:45 pages.

[8] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-C. In *Software Engineering and Formal Methods*, George Eleftherakis, Mike Hinchey, and Mike Holcombe (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 233–247.

[9] Dart Team. 2013. Dart Programming Language Specification. Version 0.41.

[10] Leonardo De Moura and Nikolaj Bjørner. 2008. 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 (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

[11] Mickaël Delahaye, Nikolai Kosmatov, and Julien Signoles. 2013. Common Specification Language for Static and Dynamic Analysis of C Programs. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC '13)*. ACM, New York, NY, USA, 1230–1235. https://doi.org/10.1145/2480362.2480593

[12] Len Erlikh. 2000. Leveraging Legacy System Dollars for E-Business. *IT Professional* 2, 3 (May 2000), 17–23. https://doi.org/10.1109/6294.846201

[13] Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. 2010. Embedded Contract Languages. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC '10)*. ACM, New York, NY, USA, 2103–2110. https://doi.org/10.1145/1774088.1774531

[14] Jean-Christophe Filliátre. 2011. Deductive Software Verification. *International Journal on Software Tools for Technology Transfer* 13, 5 (Oct. 2011), 397–403. https://doi.org/10.1007/s10009-011-0211-0

[15] Mössenböck Hanspeter, Löberbauer Markus, and WößAlbrecht. 2018. The Compiler Generator Coco/R. http://www.ssw.uni-linz.ac.at/Coco/.

[16] David Janzen and Hossein Saiedian. 2005. Test-Driven Development: Concepts, Taxonomy, and Future Direction. *Computer* 38, 9 (Sept. 2005), 43–50. https://doi.org/10.1109/MC.2005.314

[17] G. Kiczales, J. Irwin, J. Lamping, J. Loingtier, C.V. Lopes, C. Maeda, and A. Mendhekar. 1996. Aspect Oriented Programming. In *Special Issues in Object-Oriented Programming*. Max Muehlhaeuser (general editor) et al.

[18] Nico Lehmann and Éric Tanter. 2017. Gradual Refinement Types. In *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2017)*. ACM Press, Paris, France, 775–788.

[19] J. Lehtosalo and contributors. 2018. MyPy - optional static typing for Python. http://mypy-lang.org.

[20] K. Rustan M. Leino. 2010. *Dafny: An Automatic Program Verifier for Functional Correctness.* Springer Berlin Heidelberg, Berlin, Heidelberg, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

[21] K. R. M. Leino. 2017. Accessible Software Verification with Dafny. *IEEE Software* 34, 6 (November 2017), 94–97. https://doi.org/10.1109/MS.2017.4121212

[22] Bertrand Meyer. 1992. Applying "Design by Contract". *Computer* 25, 10 (Oct. 1992), 40–51. https://doi.org/10.1109/2.161279

[23] D North. 2006. Behavior Modification: The evolution of behavior-driven development. *Better Software* 8, 3 (2006).

[24] Benjamin C. Pierce. 2002. *Types and programming languages.* MIT Press, Cambridge, MA, USA.

[25] Benjamin C. Pierce (Ed.). 2005. *Advanced Topics in Types and Programming Languages.* MIT Press, Cambridge, MA, USA.

[26] Roger Pressman. 2010. *Software Engineering: A Practitioner's Approach* (7 ed.). McGraw-Hill, Inc., New York, NY, USA.

[27] Henrique Rebêlo, Gary T. Leavens, Mehdi Bagherzadeh, Hridesh Rajan, Ricardo Lima, Daniel M. Zimmerman, Márcio Cornélio, and Thomas Thüm. 2014. AspectJML: Modular Specification and Runtime Checking for Crosscutting Contracts. In *Proceedings of the 13th International Conference on Modularity (MODULARITY '14)*. ACM, New York, NY, USA, 157–168. https://doi.org/10.1145/2577080.2577084

[28] Microsoft Research. 2018. Dafny – guide. https://rise4fun.com/Dafny/tutorial.

[29] Tim Sheard, Aaron Stump, and Stephanie Weirich. 2010. Language-Based Verification Will Change The World. In *Proceedings of the FSE/SDP Workshop on the Future of Sofware Engineering Research (FoSER 2010)*. 343–348.

[30] Jeremy Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Proceedings of the Scheme and Functional Programming Workshop.* 81–92.

[31] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, Vol. 32. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Asilomar, California, USA, 274–293.

[32] Ian Sommerville. 2006. *Software Engineering: (Update) (8th Edition) (International Computer Science).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[33] Umberto Souza da Costa, Anamaria Martins Moreira, Martin A. Musicante, and Plácido A. Souza Neto. 2012. JCML: A specification language for the runtime verification of Java Card programs. *Science of Computer Programming* 77, 4 (2012), 533 – 550.

[34] Ioannis G. Stamelos and Panagiotis Sfetsos. 2007. *Agile Software Development Quality Assurance.* IGI Global, Hershey, PA, USA.

[35] Éric Tanter and Nicolas Tabareau. 2015. Gradual Certified Programming in Coq. In *Proceedings of the 11th ACM Dynamic Languages Symposium (DLS 2015)*. ACM Press, Pittsburgh, PA, USA, 26–40.

[36] The Coq Development Team. 2015. *The Coq proof assistant reference manual.* http://coq.inria.fr Version 8.5.

[37] Hongwei Xi and Frank Pfenning. 1998. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '98)*. ACM Press, 249–257.