A Tutorial on Using Dafny to Construct Verified Software [*]

Paqui Lucio
The University of the Basque Country (UPV/EHU)
paqui.lucio@ehu.eus

This paper is a tutorial for newcomers to the field of automated verification tools, though we assume the reader to be relatively familiar with Hoare-style verification. In this paper, besides introducing the most basic features of the language and verifier Dafny, we place special emphasis on how to use Dafny as an assistant in the development of verified programs. Our main aim is to encourage the software engineering community to make the move towards using formal verification tools.

1 Introduction

The major goal of software engineers is to develop reliable software, that is to say error-free systems that accomplish the task for which they were designed. In the relatively short history of software systems, several very well known software bugs with infamous (some even tragic) results, have been reported. The dramatic consequences of these errors have served as an incentive for further progress towards more rigorous (i.e. mathematical) software design, verification and validation, in order to provide evidence of correctness and robustness of software. On the one hand, many important firms –such as Microsoft, Ericsson and Rockwell Collins– are increasing their investment in the development and use of technologies based on formal methods. On the other hand, the standards for certification of safety-critical software explicitly require the application of formal methods ([12]). Software certification documents for aerospace, avionics, railway, medical devices, etc. promote the use of formal methods, emphasizing that critical software needs mathematical guarantees of its safety. In particular, formal verification is mandated in the *Common Criteria for Information Technology Security Evaluation* for the highest assurance level (cf. [8]). The software engineering community has acquired substantial scientific knowledge, several formal methods, and a variety of technology for helping on the construction of trustworthy real systems. Indeed, in the last decades, many successful examples of large-scale industrial software have been reported, from which we would like to mention the Sel4 microkernel [13, 1], the Paris Mètro Line 14 [7] and the Rotterdam Storm Surge Barrier [21]. Moreover, the application of all the technology necessary to ensure reliability of safety-critical software may well be the major reason for the absence, in the last 10 or 15 years, of serious damage caused by software failures. A traditional perception of formal verification as an expensive and difficult task to be applied to "real-world" code could be provoking that only critical software is being verified. Experiences with current tools have demonstrated that this is not the case when the software is implemented in a programming language designed with software-reliability in mind. With a minimum training in logic and some ability to understand and construct rigorous proofs, the effort of learning and using a deductive verification tool is a justifiable investment. Verification tools provide practical help to detect the presence or absence of bugs, thus yielding high-quality software. There are many verification tools that can be used in today's software development practices, such as Why3 [11] and KeY [6]. For a large, although incomplete, list of tools we refer to [5]. The fact that

[*]This work has been partially supported by the Spanish Project TIN2013-46181-C2-2-R and the Basque Project GIU15/30 and grant UFI11/45.

automated verification is a research area in continuous evolution enables the development of better and better practical tools. Once we have the knowledge and technology to build up reliable software, why do not use them in the development of every software? To do this, software engineers with expertise in formal methods is required and, to training them, software engineering curricula should provide foundations and training in the design and implementation of algorithms and data structures based on formal principles, abstract specifications and formal reasoning. In other words, automated program verification is already mature enough to be made available in software industry. However, in the last decades and for different reasons, the software engineering curricula in some European universities (including Spanish ones) have experimented a drastic reduction contents and training related to mathematical and formal methods. In particular, formal verification of software has (totally or partially) disappeared from many curricula or has been made an elective subject. In addition, some fundamental subjects, such as formal logic, reasoning, semantics, etc. have undergone serious (even total) reductions. In our university there is a traditional mandatory subject that covers the topics of formal logic-based specification of programs, formal semantics based on the Hoare calculus and formal specification of data structures. In this subject, students must solve exercises (e.g. verify one-loop-programs, prove their termination, etc.) on paper. Building on this basic course, our curriculum offers an elective subject on "Formal Methods for Software Development" where we introduce tools for deductive software verification. In this elective subject, the main goal of the assignments is to produce automatically verified software, i.e. programs along with associated mathematical proofs of their properties, in particular their functional correctness. The recent steady progress in the automation of program verification led us to think that an introductory primer on the use of these tools could be given in the middle of the mandatory course. To learn how to use any particular tool, test-suite examples are very interesting, but sometimes the details of how the proofs were "discovered" can be even more interesting and illustrating. Finding a proof in an interactive verifier can be very challenging. When a proof fails, the immediate feedback is helpful, but sometimes it is non-trivial to find out what help the verifier needs to make it through. Expertise in proof guidance is extremely important. This paper is a tutorial that –besides introducing into the most basic features of Dafny– places special emphasis on how to use Dafny as an assistant in the development of verified programs. Indeed, it is intended to encourage newcomers to automated verification to start working with one of these tools, though we assume the reader to be relatively familiar with Hoare's logic. We focus on the strategy for detecting and fixing problems while the annotated programs are constructed. For that, we deal with very simple programming examples. This paper is not intended as a comprehensive review of Dafny, indeed we deal with a small subset of it. In the official website of Dafny[1] there is also an extensive online tutorial as well as a test-suite. In addition there are several papers covering a variety of applications and examples in Dafny, e.g. [15, 16, 17, 18]. All the annotated programs constructed in this tutorial can be found and tested in `http://rise4fun.com/Dafny/r3pF`.

2 Dafny and its Integrated Development Environment

Dafny [15] is a program verification tool which includes a programming language and specification constructs. The Dafny user creates and verifies both specifications and implementations. The Dafny programming language is object-based, imperative, sequential, and supports generic classes and dynamic allocation. The specification constructs include standard pre- and postconditions, framing constructs, and termination metrics. Dafny encourages using best-practice programming styles, in particular the *design by contract* approach. Pre- and postconditions state what properties have to be true at method entry and

[1] `http://research.microsoft.com/en-us/projects/dafny/`

exit, respectively, and are used to construct the specifications or contracts of methods. The method callers must stablish preconditions and can assume postconditions (after return). The method implementers can assume preconditions and must stablish postconditions. Dafny programs are statically verified for *total correctness*, i.e., that every terminating execution satisfies its contract (*partial correctness*) and that every execution does indeed terminate. Dafny's program verifier works by translating a given Dafny program into the *intermediate verification language* Boogie [3] in such a way that the correctness of the Boogie program implies the correctness of the Dafny program. Thus, the semantics of Dafny is defined in terms of Boogie. Boogie is a layer on which to build program verifiers for other languages. For example, the program verifiers VCC [9] for C, AutoProof for Eiffel [22], and Spec# [4] are built on top of Boogie. The Boogie tool is used to generate first-order verification conditions that are passed to a logic reasoning engine. In particular, for Dafny, they are passed to the satisfiability modulo theories (SMT) solver Z3 [10]. The Dafny integrated development environment (IDE) is an extension of Microsoft Visual Studio (VS). The IDE is designed to reduce the effort required by the user to make use of the proof system. For example, the IDE runs the program verifier in the background, thus providing design time feedback. It makes Dafny to be an auto-active verifier [16]: users will interact with the proofs, but there is automation. Specification and programming constructs can be used to create the proof. Dafny allows declarative calculations and inductive as well as co-inductive proofs. Also, verification error messages can have a lot of associated information and the user can get information about the possible values of variables for a reported error using the Boogie Verification Debugger (BVD) [14] that is deeply integrated into the Dafny IDE. The interested reader is referred to [20] for further information on the Dafny IDE.

3 A First Example of Annotated Program

In this section, we will start designing an annotated program to compute the factorial of a natural number. We intentionally use a *non-typical* invariant to obtain a verified algorithm that is different from the usual one, with the main aim of introducing several basic concepts and units in Dafny.

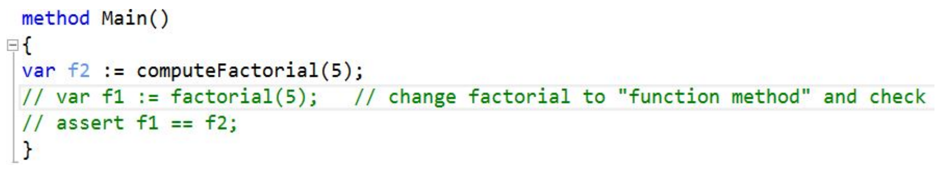
The basic unit of a Dafny program is the **method**. A method is a piece of executable code with a head where multiple named parameters and multiple named results are declared. As a first example we design an iterative method that calculates the factorial of a natural number from a very peculiar invariant aiming to provide interesting insights. Dafny also offers user-defined **function**s. By default in Dafny, functions can be used only in specifications, hence they do not generate code. To override this default, so that the compiler will generate code for a function, the function is declared with **function method**. A **predicate** is a boolean function, and a **predicate method** is a predicate for which code is generated. Dafny has built-in specification constructs for assertions, such as **requires** for preconditions, **ensures** for postconditions, **invariant** for loop invariants, **assert** for inline assertions. Multiple **requires** have the same meaning as their conjunction into a single **requires**. [2] The starting point is

```
function factorial (n:int): int
  requires n ≥ 0;
{
if n = 0 then 1 else n * factorial(n−1)
}

method computeFactorial(n:int) returns (f:int)
  requires n ≥ 0
  ensures f = factorial(n)
// annotated code will be designed from the invariant
// 0 ≤ i ≤ n−1 ∧ f * factorial(n−i−1) = factorial(n)
```

[2]The same will apply to multiple **ensures**, **invariant**, and **assert**.

where we have used a function `factorial` for specification purposes: it is used in the postcondition of the method and also in the intended invariant, which is a rather unusual one. The precondition of the function `factorial` serves to ensure the well-foundedness of the induction. The interested reader can check the error that Dafny reports if you erase/comment the above line 2. However, with the precondition $n \geq 0$, Dafny guesses that `n` is a termination metric, then the hover text **decreases** `n` appears by puting the cursor over the word `factorial`. That is, `n` is the expression whose strict decrease warrants the recursion termination. We will discuss that topic in Section 7.

If we would like to generate code for the function `factorial` we also could declare it as follows: **function method** `factorial (n: `**int**`): `**int** with the same requires clause and the same body. In that case, the result of the function method is not named, whereas in the method `computeFactorial` the result is named as `f`. They both also use two different algorithms to compute the same result:



```
method Main()
{
var f2 := computeFactorial(5);
// var f1 := factorial(5);   // change factorial to "function method" and check
// assert f1 == f2;
}
```

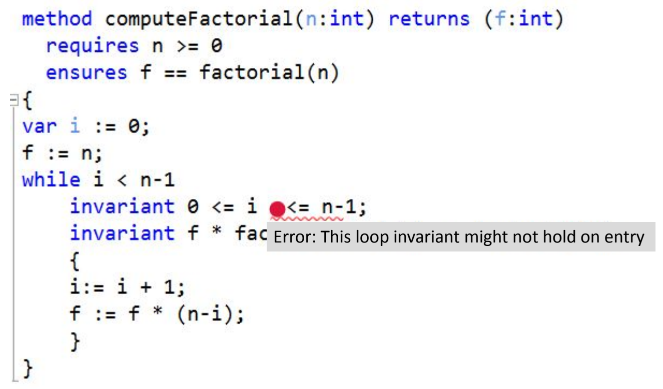
According to the invariant $0 \leq i \leq n-1 \wedge f * \text{factorial}(n-i-1) = \text{factorial}(n)$; the auxiliary variable `i` and the result `f` should be initialized by `0` and `n`, respectively. It is also easy to check that (in the iteration) after an assignment $i := i + 1$ there should be some assignment to `f` satisfying

$0 \leq i \leq n-1 \wedge f * \text{factorial}(n-i) = \text{factorial}(n)$;

$f := ?$

$0 \leq i \leq n-1 \wedge f * \text{factorial}(n-i-1) = \text{factorial}(n)$;

In this way we deduce that `f` should be assigned to be $f * (n-i)$. The resulting program is not yet correct. Indeed an error is displayed by a red dot, a red curly underline and the associated hover text (which appears when the cursor is placed over the red wavy line):



```
method computeFactorial(n:int) returns (f:int)
  requires n >= 0
  ensures f == factorial(n)
{
var i := 0;
f := n;
while i < n-1
    invariant 0 <= i <= n-1;
    invariant f * fac
    {
    i:= i + 1;
    f := f * (n-i);
    }
}
```

Error: This loop invariant might not hold on entry

Indeed, the case $n=0$ is in conflict with the assertion. A suitable fixing is

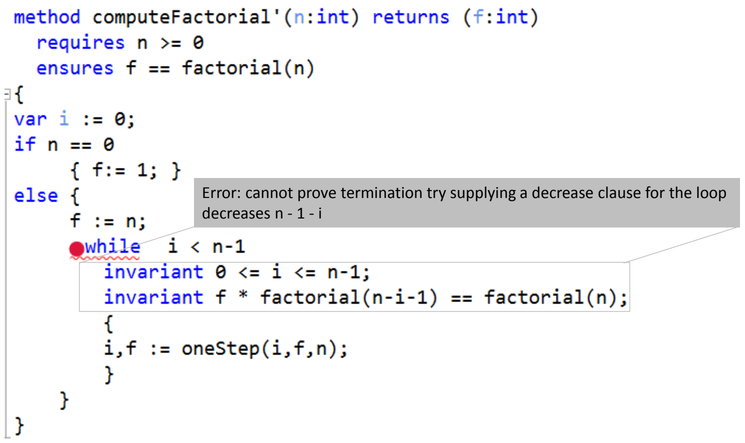
```
method computeFactorial(n: int) returns (f: int)
  requires n ≥ 0
  ensures f = factorial(n)
{
var i := 0;
if n = 0 { f:= 1;}
    else {
         f := n;
         while i < n−1
            invariant 0 ≤ i ≤ n−1;
            invariant f ∗ factorial(n−i−1) = factorial(n);
```

```
                {
                i:= i + 1;
                f := f ∗ (n−i);
                }
            }
}
```

Now, we are going to design the same program in a different way to illustrate how to use a specified (but not still implemented) method. This enables a kind of modular design by means of the fact that when a method M calls a method M', only the contract of M' (but not its body/code) is required in the correctness proof of the method M. For example, we can abstract the body of the iteration using a call to a lemma oneStep with contract

```
method oneStep(i:int,f:int,n:int) returns (i':int,f':int)
    requires 0 ≤ i < n−1 ∧ f ∗ factorial(n−i−1) = factorial(n)
    ensures 0 ≤ i' ≤ n−1 ∧ f'∗ factorial(n−i'−1) = factorial(n)
```

in this way the assignment i,f := oneStep(i,f,n); might work as the previous body of two assignments. However, we get the following error: [3]



```
method computeFactorial'(n:int) returns (f:int)
  requires n >= 0
  ensures f == factorial(n)
{
var i := 0;
if n == 0
     { f:= 1; }
else {
     f := n;
     while  i < n-1
        invariant 0 <= i <= n-1;
        invariant f * factorial(n-i-1) == factorial(n);
        {
        i,f := oneStep(i,f,n);
        }
     }
}
```

```
Error: cannot prove termination try supplying a decrease clause for the loop
decreases n - 1 - i
```

Though Dafny guesses that the expression that decreases at each step could be n−1−i, it is not able to prove that fact. Indeed, the contract of the method oneStep –the only available information– does not state anything that is related to the modification of the variables affecting the termination and that could be used to prove the corresponding verification conditions. Therefore, an additional **ensures** n−i' < n−i, or simply **ensures** i' > i, fixes the problem. To complete this *modular version* of a method for computing the factorial we should write a body for the method oneStep with the two assignments bellow:

```
method oneStep(i:int,f:int,n:int) returns (i':int,f':int)
        requires 0 ≤ i < n−1 ∧ f ∗ factorial(n−i−1) = factorial(n)
        ensures 0 ≤ i' ≤ n−1 ∧ f'∗ factorial(n−i'−1) = factorial(n)
        ensures i' > i
{
i':= i + 1;
f' := f ∗ (n−i');
}
```

Now, the error is repaired and the hover text over the word **while** of the method computeFactorial reads **decreases** n−1−i, but the interested user can write (next to the invariant) the annotation **decreases** n−i or any other correct expression which is strictly decreasing. Doing so, Dafny checks the verification conditions for the user-expression, instead of for the guessed one. We will go back to this example in Section 7 where we discuss more on termination.

[3]We have moved up the hover text for clarity.

4 Calling lemmas in program proofs

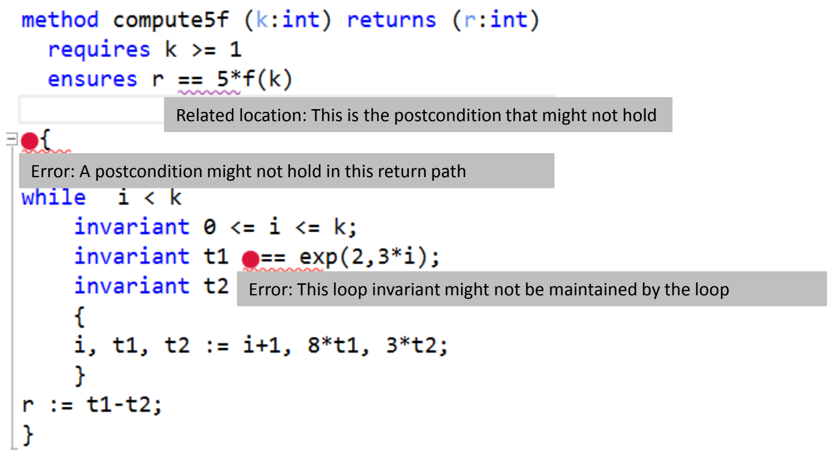
In this section, we illustrate how we guess and use lemmas that are required to verify a program. First, we design a verified method that computes the function $2^{3k}-3^k$ which is divisible by 5, hence it computes $5*f(k)$ where $f(k)=(2^{3k}-3^k)/5$. We start specifying the method as follows

```
method compute5f (k: int) returns (r: int)
  requires k ≥ 1
  ensures r = 5*f(k)

function f(k: int): int
  requires k ≥ 1;
{  (exp(2,3*k) − exp(3,k)) / 5  }

function exp(x: int, e: int): int
  requires e ≥ 0
{ if e=0 then 1 else x * exp(x,e−1) }
```

Then, we design an iteration that calculates both exponentials 2^{3n} and 3^n by iterating multiplication by 8 and by 3, respectively, in two new variables t1 and t2. That is, an iteration that preserves the invariant 0 ≤ i ≤ k ∧ t1 = exp(2,3∗i) ∧ t2 = exp(3,i) where i is also a new variable.

```
method compute5f (k: int) returns (r: int)
  requires k ≥ 1
  ensures r = 5 * f(k)
{
var i, t1, t2:= 0, 1, 1;
while i < k
        invariant 0 ≤ i ≤ k;
        invariant t1 = exp(2,3*i);
        invariant t2 = exp(3,i);
        {
        i, t1, t2 := i+1, 8*t1, 3*t2;
        }
r := t1−t2;
}
```

Dafny complains that the (unique) postcondition does not hold and also that the second conjunct of the provided invariant might not be maintained by the loop:


```
method compute5f (k:int) returns (r:int)
  requires k >= 1
  ensures r == 5*f(k)
                  Related location: This is the postcondition that might not hold
{
 Error: A postcondition might not hold in this return path
while  i < k
    invariant 0 <= i <= k;
    invariant t1 == exp(2,3*i);
    invariant t2 Error: This loop invariant might not be maintained by the loop
    {
    i, t1, t2 := i+1, 8*t1, 3*t2;
    }
r := t1-t2;
}
```

Dafny cannot complete the proof (that is, the SMT-solver Z3 cannot prove all verification conditions) by itself. Then, the user should provide assertions as hints. A *hint* is an assertion that the verifier is required to prove. Once the assertion is proved, it turns into a usable property for completing the correctness proof. Indeed, "**assert** φ" tells Dafny to check that φ holds (whenever control reaches that part of the code) and to use the condition φ (as a lemma) to prove the verification conditions beyond this program point. In order to help the user in the process of constructing proofs, in particular in the process of *guessing hints*, Dafny offers two features: the construct **assume** and the use of a declared (but yet not proved) **lemma**. Therefore, along the construction of a proof, we can introduce an assumed condition φ to check whether φ is the condition that Dafny needs to complete the proof. In other words, Dafny tries to complete the proof, assuming that φ is true, without having tried to prove φ. In order to check whether the proposed invariant is suitable for ensuring the postcondition, we write **assume** t1 =exp(2,3∗i); in line 12 and the assert of line 15 is not violated. Since the postcondition is not yet held, we add the assumption in line 16:

```
   method compute5f (k: int) returns (r: int)
     requires k ≥ 1
3    ensures r = 5∗f(k)
   {
   var i, t1, t2:= 0, 1, 1;
6  while i < k
           invariant 0 ≤ i ≤ k;
           invariant t1 = exp(2,3∗i);
9          invariant t2 = exp(3,i);
           {
           i, t1, t2 := i+1, 8∗t1, 3∗t2;
12         assume t1 = exp(2,3∗i);
           }
   r := t1−t2;
15 assert r =  exp(2,3∗k) − exp(3,k);
   assume r = 5 ∗ f(k);
   }
```

When Dafny succeeds with an assumption "**assume** φ", then it should be changed to "**assert** φ" to force Dafny to prove φ. If φ is proved, then the inline assertion is the required hint. In our example, we should help Dafny to prove the invariant-preservation verification condition related to the second conjunct of the invariant. Hence, we convert the **assume** in the above line 12 into an **assert** (see line 14 below) and, as expected, the assertion is violated. Therefore we use the weakest precondition technique to write **assume** 8∗t1 =exp(2,3∗(i+1)); (see line 11 below). Then, the assertion in line 13 (below) is verified.

```
   method compute5f (k: int) returns (r: int)
     requires k ≥ 1
3    ensures r = 5∗f(k)
   {
   var i, t1, t2:= 0, 1, 1;
6  while i < k
           invariant 0 ≤ i ≤ k;
           invariant t1 = exp(2,3∗i);
9          invariant t2 = exp(3,i);
           {
           assume 8∗t1 = exp(2,3∗(i+1));
12         i, t1, t2 := i+1, 8∗t1, 3∗t2;
           assert t1 = exp(2,3∗i);
           }
15 r := t1−t2;
   assume (exp(2,3∗k) − exp(3,k)) % 5 = 0;
   assert r = 5 ∗ f(k);
18 }
   }
```

For helping with the postcondition, when we swap **assume** r =5 ∗ f(k) to an assertion, it is violated. Since the assert **assert** r = exp(2,3∗k) − exp(3,k), is satisfied (not violated) we guess that Dafny "does not know" that exp(2,3∗k) − exp(3,k) is divisible by 5. Therefore, we write the assumption in line 17 below. But now, if we change both **assume** clauses (in lines 11 and 16) to **assert**s, both are violated. In line 11, we try to help Dafny by unfolding the assertion 8∗t1 == exp(2,3∗(i+1)) into

assert 8∗t1 = 8∗exp(2,3∗i) = exp(2,3∗i+3) = exp(2,3∗(i+1))

Then, the prover fails checking the equation 8∗exp(2,3∗i) = exp(2,3∗i+3). This is shown by a red point on the equality symbol. Similarly, in line 16, a red point is in its unique equality symbol. Then, we try to help Dafny to prove both properties: exp(2,3∗i)∗8 = exp(2,3∗i+3) and (exp(2,3∗k) − exp(3,k)) % 5 = 0. If Dafny could prove them, then Dafny would respectively use them to prove the postcondition and the invariant preservation, so that these assertions would be the required hints. Unfortunately, the assertions cannot be used as hints, but must be proved as separated lemmas. Hence, we write both lemmas and call them as follows:

```
   method compute5f (k: int) returns (r: int)
     requires k ≥ 1
3    ensures r = 5∗f(k)
   {
   var i, t1, t2:= 0, 1, 1;
6  while i < k
           invariant 0 ≤ i ≤ k;
           invariant t1 = exp(2,3∗i);
9          invariant t2 = exp(3,i);
           {
           expPlus3_Lemma(2,3∗i);
12         // assert t1∗8 = exp(2,3∗i)∗8 = exp(2,3∗i+3) = exp(2,3∗(i+1));
           i, t1, t2 := i+1, 8∗t1, 3∗t2;
           }
15 r := t1−t2;
   DivBy5_Lemma(k);
   // assert (exp(2,3∗k) − exp(3,k)) % 5 = 0;
18 }

   lemma expPlus3_Lemma (x: int, e: int)
21         requires e ≥ 0;
           ensures x ∗ x ∗ x ∗ exp(x,e) = exp(x,e+3);
   // to be proved
24
   lemma DivBy5_Lemma (k: int)
           requires k ≥ 1
27         ensures (exp(2,3∗k) − exp(3,k)) % 5 = 0
   // to be proved
```

We use each lemma call to validate the respective **assert**. That is, if a concrete lemma call works (in the sense that φ is satisfied) then we could comment (or drop) the assert. The interested reader could uncomment the above lines 12 and 19 to check out that the lemma call ensures the non-violation of the assertion. It is useful to keep commented assertions as explanations or program documentation.

The **lemma** declarations are like methods but do not need to be called at runtime, hence no code is generated for them, i.e. **lemma** is equivalent to **ghost method**.[4] Lemmas have only effect on the verification of the program, they are only necessary to help along the proof of the program. Lemmas, like methods, can have parameters, hence their re-usability by instantiation (of the parameters) is an advantage of lemmas with respect to inline asserts. In particular, the induction hypothesis of inductive lemmas is invoked as a lemma call (over smaller parameters). In Dafny, a lemma is a ghost method whose contract represents the property it warrants. Let us consider any lemma

```
lemma Ex_Lemma (x1: T1,...,xn: Tn)
  requires φ
  ensures ψ
{ body }
```

[4]There are more ghost entities in Dafny (see Section 7)

where x1,...,xn is the tuple of formal parameters and T1,...,Tn the tuple of respective types. The contract of lemma Ex_Lemma means **forall** x1,x2,...,xn • $\varphi \Longrightarrow \psi$, and its body is a proof of such property. A lemma call like Ex_Lemma(a) where a is the tuple of current parameters corresponds to

```
assert φ[a/x];
assume ψ[a/x];
```

That is, if the precondition –for the current parameters a– can be proved, we can assume that the postcondition holds –for the current parameters a. The assume clause is discharged whenever the lemma is proved.

A program-proof is not complete until all verification conditions have been discharged, i.e., all assume statements have been removed (or replaced by asserts), and all the lemmas have been proved. Hence, in our example, we should prove the lemmas, i.e. we should write their body.

5 Proving lemmas

In this section we focus on proving the two lemmas left unproved in the previous section. Sometimes Dafny is able to prove a lemma (even an inductive lemma) by itself. This means that the empty body (i.e. {}) serves as proof. Otherwise, lemma proofs are annotated code as the body of an executable (non-ghost) method. In particular, we can use properties as hints in lemma bodies. For example, for the following lemma (from the previous section) the empty body does not work:

```
lemma expPlus3_Lemma (x: int , e: int )
  requires e ≥ 0;
  ensures x * x * x * exp(x,e) = exp(x,e+3);
```

It is easy to see that it can be proved by means of three successive applications (or unfoldings) of the recursive definition of the function exp. Hence, we try to give just one hint as lemma body:

```
{
assert  x * x * x * exp(x,e) = x * x * exp(x,e+1) = x * exp(x,e+2) = exp(x,e+3);
}
```

and we succeed. Indeed also

```
{
assert  x * exp(x,e) = exp(x,e+1);
}
```

serves as hint, since Z3 automatically performs the other two unfoldings.

Dafny provides a special notation that is easy to read and understand: *calculations* [19]. A calculation in Dafny is an statement that proves a property. This notation was extracted from the *calculational method* [2], whereby a theorem is established by a chain of formulas, each transformed in some way into the next. The relationship between successive formulas (for example, equality, implication, double implication, etc.) is notated, or it can be omitted if it is the default relationship (equality). In addition, the hints (usually asserts or lemma calls) that justify a step can also be notated (in curly brackets after the relationship). Calculations are written inside the environment **calc**{ }. The grammar for calculations is:

```
CalcStatement := calc {
                      CalcBody
                      }
CalcBody := Line
            (Op Hint
             Line)*
Line := Expression ;
Op :=  ≤ | < | ≥ | > | ⟹ | ⟸ | ⟺ | ≠ | =
Hint := { (BlockStatement | CalcStatement)* }
```

where a BlockStatement is one or more assert clauses (or provisionally assume clauses), lemma calls and forall-statements.

Like in other proof assistants (e.g. Isabelle/HOL and Coq) and verifiers (e.g. Why3 and KeY), Dafny allows proofs to be written in different styles and with different levels of description of the outcome of every logical transformation. Therefore, proof readability and easy checking by humans is part of the work of the Dafny user. To illustrate that issue we will give two different proofs for the property DivBy5_Lemma. The first proof shows a pedantic number of details:

```
   lemma DivBy5_Lemma (k: int)
     requires k ≥ 1
3    ensures (exp(2,3*k) - exp(3,k)) % 5 = 0
   {
   if k=1 {
6        // the base case is automatically proved
         }
     else {
9        calc {
              (exp(2,3*k) - exp(3,k)) % 5;
              =
12            (exp(2,3*(k-1)+3) - exp(3,(k-1)+1)) % 5;
              = {
                expPlus3_Lemma(2,3*(k-1));                  // lemma call
15              }
              (exp(2,3*(k-1))*8 - exp(3,(k-1))*3) % 5;
              =
18            (3*( exp(2,3*(k-1)) - exp(3,k-1) ) + 5* exp(2,3*(k-1))) % 5;
              = {
                DivBy5_Lemma(k-1);                   // lemma call for IH
21              // assert (exp(2,3*(k-1)) - exp(3,k-1)) % 5 = 0;       // IH
                }
              0;
24            }
           }
   }
```

The above proof is divided into two cases, as indicated by an if statement. The base case, for k=1, is automatically proved in lines 5-7. The other case uses a calculational proof that proves that the expression exp(2,3∗k) − exp(3,k) is a multiple of 5. In the calculation we use two lemma calls as hints in lines 14 and 21. The latter calls this lemma itself recursively. This call is treated in accordance with programming rules: the precondition of the callee is checked, termination is checked, and then the postcondition can be assumed. In effect, this sets up a proof by induction, where the recursive call to the lemma obtains the inductive hypothesis. Next, an alternative proof with only the essential hints:

```
lemma DivBy5_Lemma_Simplified(k: int)
  requires k ≥ 1
  ensures (exp(2,3*k) - exp(3,k)) % 5 = 0
{
if k⟩1 {
      expPlus3_Lemma(2,3*(k-1));         // lemma call
      DivBy5_Lemma_Simplified(k-1);      // lemma call for IH
    }
}
```

A user can decide to keep as many details as are deemed helpful for human understanding and to elide those that seem more like clutter. The second proof may take more head-scratching for a human to understand. Provided enough hints are supplied for Dafny to complete the proof, the trade-off between clarity and clutter is up to the user and depends on how many details the user wants to show for human readers.

6 A verified bubble sort method

In this section we explain a verified implementation of the well-known bubble sort algorithm. The aim of this example is twofold. It gives us the opportunity to introduce predicates, arrays, mutable states and framing in Dafny, whereas we detail a more interesting example of modular design based on the design-by-contract paradigm.

To start, we should define a predicate `sorted` whose parameter is an array of integers and a method `bubbleSort` as follows:

```
  predicate sorted (a: array⟨int⟩)
          requires a ≠ null
3         reads a

  method bubbleSort (a: array⟨int⟩)
6         requires a ≠ null
          modifies a
          ensures sorted(a)
```

Arrays are a built-in part of the language Dafny, with their own type **array**⟨T⟩, where T is another type. Arrays are objects, hence mutable, that are dynamically allocated, hence they can be null. We write **requires** a ≠ **null** to state that both the predicate `sorted` and the method `bubbleSort` cannot be applied to a null object. For dealing with a mutable state, every method specification should say which objects are allowed to be changed, every function (in particular predicate) specification should detail the set of mutable objects which it depends on. This is called *framing*: the former is specified by a **modifies** clause, like the one in the above method `bubbleSort` (see line 7) and the latter by a **reads** clause, like the one in the predicate `sorted` (see line 3). So, **reads** and **modifies** clauses provide a set of objects. Methods are allowed to read whatever they want, so their **reads** clauses do not need to be specified. Functions/Predicates are not allowed to modify objects, so **modifies** clauses cannot be specified. The **modifies** clause says that the method is allowed to modify the state of any of those objects. The **reads** clause says that the function is allowed to depend on the state of any of those objects.

A complete specification of any sorting method should say that the final object state is a permutation of the initial object state, but we intentionally delay the permutation part of the specification until the code with the sorting property has been verified. Of course, our code will only swap elements in the array.

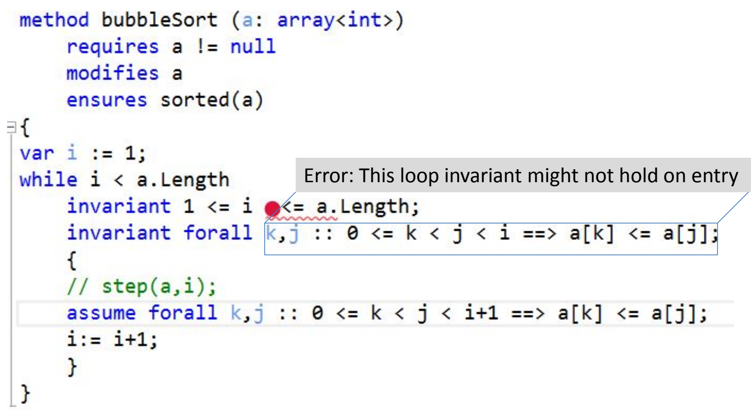
By now, we can define the predicate `sorted` by **forall** i,j • 0 ≤ i < j < a.Length ⟹ a[i] ≤ a[j].

Then, we have to write annotated code for the method `bubbleSort` which we would like it to work as shown in the following example:

$$\begin{array}{llllll}
\text{Step } i=1 & 7,\underline{2},6,3,4 & \Longrightarrow & \underline{2},\underline{7},6,3,4 & & \\
\text{Step } i=2 & 2,7,\underline{6},3,4 & \Longrightarrow & 2,\underline{6},\underline{7},3,4 & & \\
\text{Step } i=3 & 2,6,7,\underline{3},4 & \Longrightarrow & 2,6,\underline{3},\underline{7},4 & \Longrightarrow & 2,\underline{3},6,\underline{7},4 \\
\text{Step } i=4 & 2,3,6,7,\underline{4} & \Longrightarrow & 2,3,6,\underline{4},\underline{7} & \Longrightarrow & 2,3,\underline{4},6,\underline{7}
\end{array}$$

Hence, we get the following first version of annotated code:[5]

[5]where we have moved up the hover text for visibility.



```
method bubbleSort (a: array<int>)
    requires a != null
    modifies a
    ensures sorted(a)
{
var i := 1;
while i < a.Length
    invariant 1 <= i <= a.Length;
    invariant forall k,j :: 0 <= k < j < i ==> a[k] <= a[j];
    {
    // step(a,i);
    assume forall k,j :: 0 <= k < j < i+1 ==> a[k] <= a[j];
    i:= i+1;
    }
}
```

Error: This loop invariant might not hold on entry

It is easy to realize that $i \leq$ a.Length cannot be warranted at the loop entry since $i := 1$ is the loop initialization, unless $i > 1$ holds in entry. We can add a precondition a.Length >1, the interested reader could check that it works. Since any singleton array is sorted, a better thing to do is to adapt code to do nothing with singletons. At the same time, we realize that the sorted property is used in the precondition, in the invariant, in the assume clause, and will be used in the not yet defined method bubbleStep. So it would be useful to parameterize the predicate sorted. Consequently, we get the new version:

```
   predicate sorted (a: array⟨int⟩)
     requires a ≠ null
3    reads a
   {
   sortedBetween(a,0,a.Length)
6  }

   predicate sortedBetween (a: array⟨int⟩, lo: int, hi: int)
9    requires a ≠ null ∧ 0 ≤ lo ≤ hi ≤ a.Length
     reads a
   {
12 forall i,j • lo ≤ i < j < hi ⟹ a[i] ≤ a[j]
   }

15 method bubbleSort (a: array⟨int⟩)
     requires a ≠ null
     modifies a
18   ensures sorted(a)
   {
   if a.Length > 1
21         {
           var i := 1;
           while i < a.Length
24                 invariant 1 ≤ i ≤ a.Length;
                   invariant sortedBetween(a,0,i);
                   {
27                 // bubbleStep(a,i);
                   assume sortedBetween(a,0,i+1);
                   i:= i+1;
30                 }
           }
   }
```

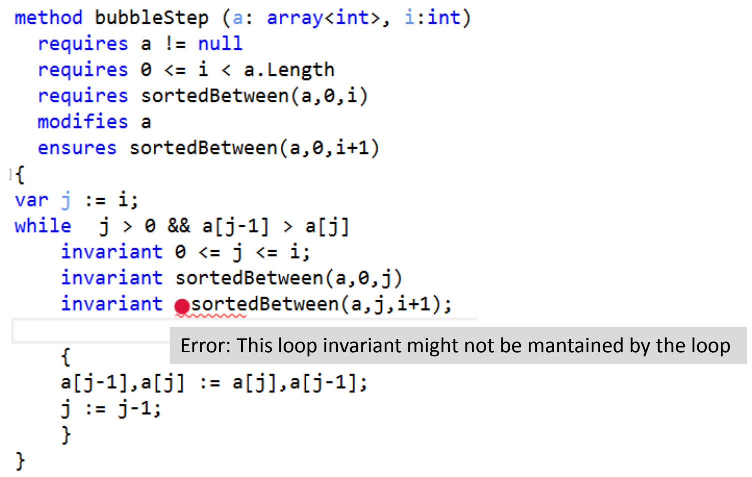
Then, we can comment the assume clause in line 29 and uncomment the lemma call bubbleStep(a,i) in line 28 whenever we add the following specification of the method bubbleStep:

```
   method bubbleStep (a: array⟨int⟩, i: int)
36   requires a ≠ null ∧ 0 ≤ i < a.Length ∧ sortedBetween(a,0,i)
     modifies a
     ensures sortedBetween(a,0,i+1)
```

Next, annotated code for the latter method is required. Consider the last step for $i = 4$:

$$2,3,6,7,\underline{4} \implies 2,3,6,\underline{4},\underline{7} \implies 2,3,\underline{4},6,\underline{7}$$

It is easy to see that a new variable j should be initialized to be i and should be decreased while a[j−1] >a[j] whereas these two elements are swapped. It is also easy to see that such process preserves the invariant sortedBetween(a,0,j) ∧ sortedBetween(a,j,i+1), but all that is not still correct:



```
method bubbleStep (a: array<int>, i:int)
  requires a != null
  requires 0 <= i < a.Length
  requires sortedBetween(a,0,i)
  modifies a
  ensures sortedBetween(a,0,i+1)
{
var j := i;
while   j > 0 && a[j-1] > a[j]
    invariant 0 <= j <= i;
    invariant sortedBetween(a,0,j)
    invariant ●sortedBetween(a,j,i+1);
                Error: This loop invariant might not be mantained by the loop
    {
    a[j-1],a[j] := a[j],a[j-1];
    j := j-1;
    }
}
```

Dafny can help us to analyze why this invariant might not be maintained by the loop. We use the previously explained assume/assert mechanism to guess the assumption in the line 15 below:

```
   method bubbleStep (a: array⟨int⟩, i:int)
     requires a ≠ null
3    requires 0 ≤ i < a.Length
     requires sortedBetween(a,0,i)
     modifies a
6    ensures sortedBetween(a,0,i+1)
   {
   var j := i;
9  while j > 0 ∧ a[j−1]  > a[j]
           invariant 0 ≤ j ≤ i;
           invariant sortedBetween(a,0,j)
12         invariant sortedBetween(a,j,i+1);
           {
           //assert a[j−1] > a[j] ∧ sortedBetween(a,j,i+1);
15         assume j+1 ≤ i  ⟹ a[j−1] ≤ a[j+1];
           a[j−1],a[j] := a[j],a[j−1];
           assert sortedBetween(a,j−1,i+1);
18         j := j−1;
           assert sortedBetween(a,j,i+1);
           }
21 }
```

We first assume the property in line 19 and, of course, everything works, but as an assertion (line 19) it is violated. Then, we fix this problem by assuming the property in line 17. Again, as an assertion line 17 is also violated. Now, we calculate that a[j] ≤a[j−1] ≤a[j+1] ≤ ...≤a[i] ≤a[i+1] should be true in line 15. Since the commented assertion in line 14 is true, we guess that we must assume that a[j−1] ≤a[j+1] with the guard j+1 ≤i for preventing "Error: index out of range". Again, changing that property to an assert, we get an "Error: assertion violation" Hence, we learn that the assumed property should be part of the invariant. Adding **invariant** j+1 ≤i ⟹a[j−1] ≤a[j+1] produces an out-of-range error on index j−1 because j is 0 at the end of the loop (but not inside the loop). Consequently, the invariant to be added is: 1 <j+1 ≤i ⟹a[j−1] ≤a[j+1].

```
predicate permutation (a:seq⟨int⟩,b:seq⟨int⟩)
{  multiset(a) = multiset(b)  }

predicate sortedBetween (a:array⟨int⟩, lo:int, hi:int)
  requires a ≠ null ∧ 0 ≤ lo ≤ hi ≤ a.Length
  reads a
{  forall i,j • lo ≤ i < j < hi ⟹ a[i] ≤ a[j] }

predicate sorted (a:array⟨int⟩)
  requires a ≠ null
  reads a
{ sortedBetween(a,0,a.Length) }

method bubbleSort (a: array⟨int⟩)
  requires a ≠ null
  modifies a
  ensures sorted(a)
  ensures permutation(a[..], old(a[..]))
{
if a.Length  > 1
        {
        var i := 1;
        while i < a.Length
                invariant 1 ≤ i ≤ a.Length;
                invariant sortedBetween(a,0,i);
                invariant permutation(a[..], old(a[..]));
                {
                bubbleStep(a,i);
                i:= i+1;
                }
        }
}

method bubbleStep (a: array⟨int⟩, i:int)
  requires a ≠ null ∧ 0 ≤ i < a.Length ∧ sortedBetween(a,0,i)
  modifies a
  ensures sortedBetween(a,0,i+1)
  ensures permutation(a[..], old(a[..]))
{
var j := i;
while j  > 0 ∧ a[j−1]  > a[j]
        invariant 0 ≤ j ≤ i ∧ sortedBetween(a,0,j) ∧ sortedBetween(a,j,i+1);
        invariant 1 < j+1 ≤ i ⟹ a[j−1] ≤ a[j+1];
        invariant permutation(a[..], old(a[..]));
        {
        a[j−1],a[j] := a[j],a[j−1];
        j := j−1;
        }
}
```

Figure 1: Bubble Sort

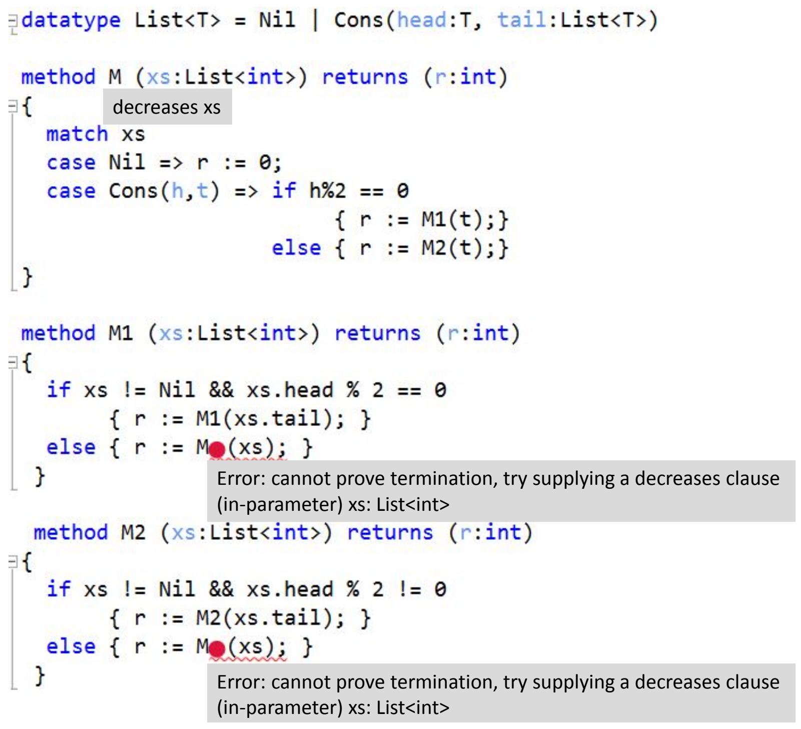
To complete the program as given in Figure 1, we should enrich the specification of the method bubbleSort with the permutation property. For that, we firstly should add a predicate expressing when a sequence of elements is a permutation of another sequence of elements. In Dafny, the sequence of elements in an array a is denoted by a [..] , which is equivalent to a [0.. a.Length]. Dafny sequences are an immutable value type. Value types can be stored in fields (i.p. var) on the heap, and used in real code in addition to specifications. Variables that contain a value type can be updated to have a new value of that type. Dafny also has multisets as inmutable value type. The built-in unary function **multiset** gives the multiset conversion of a sequence.

```
predicate permutation (a:seq⟨int⟩,b:seq⟨int⟩)
{ multiset(a) = multiset(b) }
```

Now, we can add the ensures clause: permutation(a [..], **old**(a [..])) to the method bubbleSort. The expression **old**(E) stands for the value of the expression E when evaluated on entry to a method. Consequently, it makes sense to use **old** in postconditions, and also in assertions (i.p. invariants) to refer to the entry value of a method parameter. Then, we should add also the fact permutation(a [..], **old**(a [..])) to the invariant, but then it should be also a postcondition of the method bubbleStep and, for that, it should be preserved also by the loop in this method. We add all these facts and Dafny proves all them.

7 Termination metrics

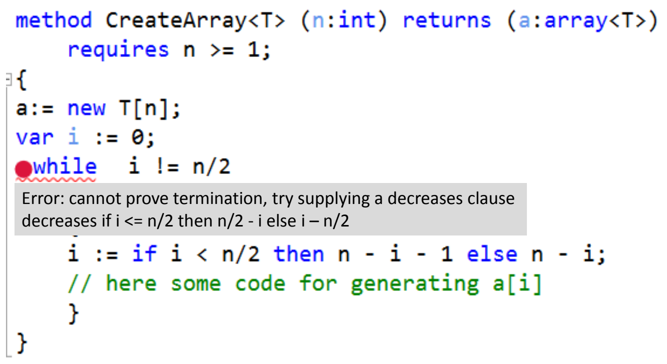
In this section we mainly give some hints on termination proofs, while introducing some other Dafny basic features, mainly datatypes and ghost variables. Dafny sets out to prove termination of all loops and of all recursion among methods and functions by means of **decreases** annotations. A decreases annotation specifies an expression whose value is compared for successive loop iterations and for caller and callee. Termination is ensured whenever the successive values become strictly smaller according to some well-founded order. Dafny has rules for guessing terminations metrics. For example, Dafny has successfully guessed a metric for every of the previous methods in this paper. In Section 3 we have already explained the guessed metric for function factorial and method computeFactorial. The interested reader can check that Dafny also guessed **decreases** k − i for the while loop in the method compute5f, **decreases** a.Length − i for the while loop in the method bubbleSort, and **decreases** j − 0 (that is, j) for the while loop in the method bubbleStep. Though the most common metrics are of type integer, other types of expressions also work, including e.g. (finite) sequences (whose well-founded order Dafny defines to be proper-prefix ordering). In particular, tuples are very useful as termination metrics. Dafny compare tuples lexicographically. If the guessed metric is not fine enough for proving termination, Dafny system asks the user to provide one through the hover text:



```
datatype List<T> = Nil | Cons(head:T, tail:List<T>)

method M (xs:List<int>) returns (r:int)
{       decreases xs
  match xs
  case Nil => r := 0;
  case Cons(h,t) => if h%2 == 0
                       { r := M1(t);}
                    else { r := M2(t);}
}

method M1 (xs:List<int>) returns (r:int)
{
  if xs != Nil && xs.head % 2 == 0
       { r := M1(xs.tail); }
  else { r := M●(xs); }
 }        Error: cannot prove termination, try supplying a decreases clause
          (in-parameter) xs: List<int>
 method M2 (xs:List<int>) returns (r:int)
{
  if xs != Nil && xs.head % 2 != 0
       { r := M2(xs.tail); }
  else { r := M●(xs); }
 }        Error: cannot prove termination, try supplying a decreases clause
          (in-parameter) xs: List<int>
```

The above example is really a skeleton or scheme[6] of mutual recursion that is interesting from the point of view of termination. It deals with the usual algebraic/inductive **datatype** of polymorphic lists[7] based on contructors Nil and Cons where, we have also declared the two usual destructors: head and tail. In methods M1 and M2 we use these destructors and the **if** statement to implement recursion, whereas the method M uses the **match** statement. For the method M, Dafny guessed that the in-parameter xs could serve to prove termination, provided that the other two methods also terminate. But the same expression xs, which is the only guessing in M1 and M2, is not fine enough to prove termination. In both cases, the **else** path is the issue, where the method M is called without modifying the parameter xs. The most direct solution is to use a pair of integer expressions as metric. For the method M we write **decreases** xs, 1, whereas **decreases** xs, 0 serves in the other two methods. Therefore, the first expression is a variable and the second is a constant (either 0 or 1), so a strict decrease happens if the value of the variable strictly goes down, or if the variable remains unchanged and the caller has the 1 and the callee has the 0.

Dafny *ghost* entities are used only during verification; the compiler omits them from the executable code, hence they can be used to help Dafny to complete a proof without jeopardizing execution cost. Ghost variables are useful when a non-computed value v has some interesting property that would allow to prove the required property, but v is not really needed in the real code. The following example is a bit contrived but illustrative nevertheless. The method (scheme) CreateArray generates an array a, using only one index i, to generate components in the following order: a [0], a[n−1],a [1], a[n−2],a [2], a[n−3],a [3], ... where n = a.Length.



```
method CreateArray<T> (n:int) returns (a:array<T>)
    requires n >= 1;
{
a:= new T[n];
var i := 0;
while  i != n/2
Error: cannot prove termination, try supplying a decreases clause
decreases if i <= n/2 then n/2 - i else i – n/2
    i := if i < n/2 then n - i - 1 else n - i;
    // here some code for generating a[i]
    }
}
```

It should be noted that T is a type variable and that the method name is followed by ⟨T⟩ to denote that the method depends on that variable i.e. is polymorphic.

Dafny heuristics yields an expression (see the above hover text) that indeed decreases but not strictly. A suitable solution for proving termination is to introduce some kind of counter as a ghost variable. That is, we declare and initialize: **ghost var** c := 1;. Then, the ghost variable c should be incremented at each iteration. Hence, we can envisage that n−c is a good metric for termination. For that, we should add an invariant stating that $0 \leq i \leq n \wedge c \leq n$. However, $c \leq n$ cannot be proved to be maintained by the loop. For helping Dafny to prove $c \leq n$, we should add some invariant property that relates the ghost variable c with i and n. There are many possible invariants that enable Dafny to prove termination. One possible invariant is depicted in line 9 below:

[6]To carry out a real task, the code should be completed with more actions after the method calls, but we would like to concentrate on termination of this kind of mutual recursion.

[7]In the three methods type variable T is instantiated to **int**.

```
method CreateArray⟨T⟩ (n: int) returns (a: array⟨T⟩)
  requires n ≥ 1;
{
a:= new T[n];
var i := 0;
ghost var c := 1;
while i ≠ n/2
        invariant 0 ≤ i ≤ n ∧ c ≤ n;
        invariant c = 2*(n−i) ∨ c = 2*i+1;
        decreases n−c;
        {
        i := if i < n/2 then n − i − 1 else n − i;
        // here some code for generating a[i]
        c := c + 1;
        }
}
```

Other interesting applications of ghost variables include to simplify specifications and to specify class invariants in OO programming.

8 Conclusion

Dafny is one of the many state-of-the-art tools that are currently being used in academic and industrial projects for the construction of reliable software. On the one hand, the Dafny language is easy to learn though it incorporates most of the good features of modern programming and specification languages. On the other hand, Dafny enables assertional proofs of correctness written as part of the program text. Correctness proofs explain the reasons of the program correctness. While proof ingredients are provided by the user, the proof steps themselves are carried out automatically by the verifier.

I hope that this tutorial contributes to encourage the teaching of tool-supported formal verification issues in Software Engineering curricula. I also hope that research and developments will continue aiming at making formal methods more applicable.

The development of formal verification tools (DFVT) is promoting many interesting research areas –such as invariant generation, quantification handling, etc.– closely related to the field of automated reasoning (AR). Both areas, DFVT and AR, are mutually benefiting from their achievements.

Acknowledgments. I am very grateful to the anonymous referees for their constructive comments.

References

[1] June Andronick, Ross Jeffery, Gerwin Klein, Rafal Kolanski, Mark Staples, He (Jason) Zhang, and Liming Zhu. Large-scale formal verification in practice: A process perspective. In *International Conference on Software Engineering*, pages 1002–1011, Zurich, Switzerland, jun 2012. ACM, doi:10.1109/ICSE.2012.6227120

[2] Roland Backhouse. The calculational method. *Information Processing Letters*, 53(3):121, 1995, doi:10.1016/0020-0190(94)00212-H

[3] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111 of *LNCS*, pages 364–387. Springer, September 2006, doi:10.1007/11804192_17

[4] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and verification: The Spec# experience. *Comm. ACM*, 54(6):81–91, June 2011, doi:10.1145/1953122.1953145

[5] Bernhard Beckert and Reiner Hähnle. Reasoning and verification: State of the art and current trends. *Intelligent Systems, IEEE*, 29(1):20–29, Jan 2014, doi:10.1109/MIS.2014.3

[6] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-oriented Software: The KeY Approach*. Springer, 2007, doi:10.1007/978-3-540-69061-0

[7] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. Météor: A successful application of B in a large project. In *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 20-24, 1999, Proc., Volume I*, pages 369–387, 1999, doi:10.1007/3-540-48119-2_22

[8] Common criteria for information technology security evaluation, 2012. CCMB-2012-09-001 Ver. 3.1 Rev. 4.

[9] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd Int. Conf., TPHOLs 2009*, volume 5674 of *LNCS*, pages 23–42. Springer, August 2009, doi:10.1007/978-3-642-03359-9_2

[10] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, March–April 2008, doi:10.1007/978-3-540-78800-3_24

[11] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems — 22nd European Symposium on Programming, ESOP 2013*, volume 7792 of *LNCS*, pages 125–128. Springer, March 2013, doi:10.1007/978-3-642-37036-6_8

[12] Gabriella Gigante and Domenico Pascarella. Formal methods in avionic software certification: The do-178c perspective. In *Proc. of the 5th Int. Conf. on Leveraging Applications of Formal Methods, Verification and Validation: Applications and Case Studies - Vol. Part II*, ISoLA'12, pages 205–215. Springer-Verlag, 2012, doi:10.1007/978-3-642-34032-1_21

[13] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, 2010, doi:10.1145/1629575.1629596

[14] Claire Le Goues, K. Rustan M. Leino, and Michał Moskal. The Boogie Verification Debugger. In Gilles Barthe, Alberto Pardo, and Gerardo Schneider, editors, *Software Engineering and Formal Methods — 9th International Conference, SEFM 2011*, volume 7041 of *LNCS*, pages 407–414. Springer, November 2011, doi:10.1007/978-3-642-24690-6_28

[15] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *LPAR-16*, volume 6355 of *LNCS*, pages 348–370. Springer, April 2010, doi:10.1007/978-3-642-17511-4_20

[16] K. Rustan M. Leino. Automating theorem proving with SMT. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving: 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, pages 2–16, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg, doi:10.1007/978-3-642-39634-2_2

[17] K. Rustan M. Leino. Developing verified programs with Dafny. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1488–1490, Piscataway, NJ, USA, 2013. IEEE Press, doi:10.1007/978-3-642-27705-4_7

[18] K. Rustan M. Leino and Paqui Lucio. An assertional proof of the stability and correctness of natural mergesort. *ACM Trans. Comput. Log.*, 17(1):6, 2015, doi:10.1145/2814571

[19] K. Rustan M. Leino and Nadia Polikarpova. Verified calculations. In Ernie Cohen and Andrey Rybalchenko, editors, *Verified Software: Theories, Tools, Experiments — 5th International Conference, VSTTE 2013, Revised Selected Papers*, volume 8164 of *LNCS*, pages 170–190. Springer, 2014, doi:10.1007/978-3-642-54108-7_9

[20] K. Rustan M. Leino and Valentin Wüstholz. The Dafny integrated development environment. In Catherine Dubois, Dimitra Giannakopoulou, and Dominique Méry, editors, *Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014*, volume 149 of *EPTCS*, pages 3–15, April 2014, doi:10.4204/EPTCS.149.2

[21] Ken Madlener, Sjaak Smetsers, and Marko C. J. D. van Eekelen. A formal verification study on the rotterdam storm surge barrier. In Jin Song Dong and Huibiao Zhu, editors, *ICFEM*, volume 6447 of *LNCS*, pages 287–302. Springer, 2010, doi:10.1007/978-3-642-16901-4_20

[22] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Nadia Polikarpova. AutoProof: Auto-active functional verification of object-oriented programs. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 9035 of *LNCS*, pages 566–580. Springer, 2015, doi:10.1007/978-3-662-46681-0_53