An Introduction to Dafny for Electrical Engineers

John Wickerson

October 29, 2020

Contents

Chapter 1

Introduction

This document provides an introduction to automated software verification using a tool called Dafny. It is aimed at an audience of electrical engineers. Some prior programming experience is assumed. No particularly complicated mathematics is needed, but an ability to trace program executions manually will be helpful.

Dafny is one of several tools available for verifying the functional correctness of programs. Others include VCC, VeriFast, Facebook Infer, Why, and Whiley. Dafny was invented around 2009 by Rustan Leino from Microsoft Research, and has been under development ever since. It is freely available to download.

https://github.com/dafny-lang/dafny

There is a growing interest in using tools like Dafny. In short, computer systems are becoming increasingly complicated and increasingly relied upon. It is therefore becoming ever more *difficult* and ever more *important* to guarantee that they behave correctly. One way to establish the correctness of a piece of software is to program it and verify it in a language like Dafny. Besides being an ordinary imperative program language, Dafny allows users to write *specifications* of their programs. Dafny verifies that the program adheres to its specification by generating a set of logical statements called *verification conditions*, and then automatically proving these using an automated theorem prover called Z3. This is all done without actually running the program, or even providing any of the program's inputs. This means that the verification result holds for *any* possible input.

Already, large pieces of software have been verified in this manner. For instance, Microsoft's *Hyper-V* system, which comprises about 60,000 lines of low-level C and assembly code, has been verified using a software verification tool called VCC.[1] And as verification tools (and the automatic theorem provers on

[1] https://www.microsoft.com/en-us/research/project/vcc-a-verifier-for-concurrent-c/

which they rely) become ever more powerful, we are likely to see them applied to many other pieces of software in the future.

The aim of this course, then, is simply to introduce electrical engineers to this important topic.

After getting you started with Dafny (Chapter 2), these notes present a few worked examples of how some small programs can be verified in Dafny (Chapters 3 and 4).

Acknowledgements I'd like to thank Dr Matt Windsor for his invaluable help in developing this material.

Chapter 2

Getting started

You can run Dafny in your web browser at `https://rise4fun.com/Dafny`. Alternatively, you can install it locally through the *Visual Studio Code* IDE – instructions for this are at `https://github.com/dafny-lang/dafny/wiki/INSTALL`.

If you use the online editor, you'll need to click the ▶ button to re-verify your program each time you make a change. If you use *Visual Studio Code*, your program will re-verify automatically each time you make a change.

If you use *Visual Studio Code*, you can export your Dafny code into an executable. This only works if your program has passed verification. To do so, press `F5` and then look at the 'Output' pane.

Throughout this document, I'll assume that you're using *Visual Studio Code*.

Chapter 3

First program: maximum of two numbers

Here is a simple Dafny program for returning the maximum of two integers.

```
method max(x:int, y:int) returns (r:int) {
  if x < y {
    return y;
  } else {
    return x;
  }
}
```

The syntax should be reasonably familiar, but there are a few bits worth noting:

- Output variables are named in the function's signature, a bit like in Verilog. There can be multiple output variables, separated by commas. Output variables (here `r`) can be assigned to, but input variables (here `x` and `y`) cannot.
- Outputs can be returned either as parameters of the `return` keyword, or by assigning to a named output variable. So, rather than `return y;`, I could have written `r := y; return;`. And note here that unlike in C, assignments are written using := rather than =.
- There's no need for parentheses around the conditions of `if` statements (or `while` statements). But their bodies do need to be enclosed in `{` braces `}`.

If we add a `Main` function

```
method Main() {
  var m:int := max(3,5);
  print m, "\n";
}
```

then we can press `F5` to see that the program appears to be working correctly – at least on this particular set of inputs. But what can we say about the program's behaviour on *any* input?

3.1 Formal specifications

Let us provide a specification for our program. We can do this using **`requires`** and **`ensures`** clauses. One thing that our program **`ensures`** (we hope) is that the returned value is greater than or equal to both of the inputs. We can express this as follows.

```
method max(x:int, y:int) returns (r:int)
  ensures r>=x
  ensures r>=y
{
  if x < y {
    return y;
  } else {
    return x;
  }
}
```

After you type this in, you should see 'Verified' in the status bar at the bottom of your screen.

Expressions introduced with the **`ensures`** keyword are called *postconditions*, because they are conditions that are guaranteed to be true *after* the program finishes executing. We can also write *preconditions* using the **`requires`** keyword – these are conditions that are required to be true *before* the program begins executing. This particular program does not need any preconditions.

(If you changed the `r>=y` postcondition into `r>y`, Dafny would soon object – it would say 'Not verified' in the status bar, and would list the problems it found in the 'Problems' pane.)

We have now confirmed that our `max` program meets its specification. But is the specification good enough? We can demonstrate an inadequacy in our specification by changing the code from **`return`** `y` to **`return`** `y+1`. Dafny confirms that the specification still holds, but our code is no longer doing what we might expect. Our specification is too *weak*.

We can strengthen our specification by adding another postcondition: that the returned value is either `x` or `y`.

```
ensures r==x || r==y
```

With this postcondition, **`return`** `y+1` will (happily) no longer work.

3.2 Beyond preconditions and postconditions

Is our specification now strong enough? Probably yes. There's nothing else worth saying about the relationship between the program's outputs and its inputs. Is there anything else worth specifying about our program? One thing we might want to know is that our program always terminates. In fact, Dafny has already checked this for us. To see this, observe what happens if you write a program that *doesn't* terminate. Try adding

```
while true {}
```

at the start of the program, and note that the verification fails.

If Dafny says that a program is 'Verified', it guarantees that whenever a method is called, if all of its preconditions are met, then:

1. the method will terminate,
2. when the method terminates, all of its postconditions will be met,
3. all array accesses made by the method will be within the array's bounds, and
4. all assertions in the method will pass.

Chapter 4

Second program: maximum of an array

Here is a program for calculating the maximum value in an array of integers.

```
1  method maxarray (A:array<int>) returns (r:int)
2  {
3    r := A[0];
4    var i := 1;
5    while i < A.Length
6    {
7      if r < A[i] {
8        r := A[i];
9      }
10     i := i+1;
11   }
12 }
```

Even before we have specified any postconditions, Dafny is unhappy: it is concerned that the array access `A[0]` on line 3 might be out of bounds. This concern is reasonable, because `A` might be an empty array (that is, `A.Length` is 0). We could rectify this by adding some code to return a special value (minus infinity, perhaps?) if the array is empty. However, on this occasion, we will follow an alternative approach: we will add the precondition

```
requires 0 < A.Length
```

which prevents the method being called on an empty array.

Dafny also needs to prove that the **`while`** loop always terminates. A general method for proving that a loop terminates is to devise a *measure*, which is an expression that (i) evaluates to a non-negative integer, and (ii) decreases on every iteration of the loop. In this case, we can choose `A.Length - i` as our measure. This will decrease by 1 on each iteration of the loop, and when it reaches 0, the loop will exit. Dafny is quite good at guessing measures on its own, but you can provide your own measure by writing

```
decreases A.Length - i
```

after the loop condition.

At this point, we can be sure that our program has no out-of-bounds array accesses and will always terminate. Let us now prove that it actually does something useful.

The postcondition

```
ensures exists j :: 0 <= j < A.Length && r == A[j]
```

can be read as: 'there exists a value `j` such that `j` is between zero (inclusive) and `A.Length` (exclusive) and `r` is equal to the value of array `A` at index `j`'. Or, in short: `r` is one of the elements in the array. We say that `j` is an *existentially quantified* variable, which means that the expression after the `::` is true for *some* values of `j`. (Had we written **`forall`** instead of **`exists`** then `j` would have been *universally quantified*, which means that the expression after the `::` would have to be true for *all* values of `j`.)

This postcondition is indeed met by our program, but Dafny is not able to determine this without further assistance. The problem is the **`while`** loop. Without knowing the inputs of a program, Dafny can't know how many iterations of a loop will be executed. There are an unbounded number of 'execution paths' through the program – one where the loop is not executed at all, one where it is executed once, one where it is executed twice, and so on. Plainly, Dafny cannot enumerate all of these, so it must find a different way to break the problem down.

4.1 Reasoning about loops

What it does is rely on *loop invariants*. A loop invariant is a condition that is true immediately before entering the loop and at the end of every iteration of the loop. It is worth being very precise about exactly where loop invariants are supposed to be true, and a flowchart is the best way to do this – see Figure 4.1.

With the help of loop invariants, Dafny reasons about loops in three steps.

1. It first tries to prove that the loop invariants are true before entering the loop, using whatever knowledge it has established about the program's state at this point.

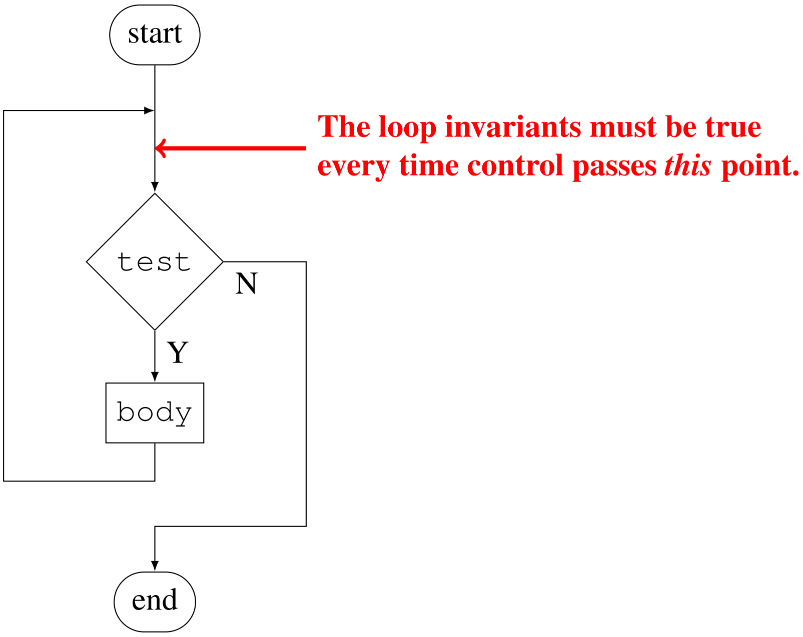
start
The loop invariants must be true
every time control passes *this* point.
test
N
Y
body
end

Figure 4.1: A flowchart for '`while test { body }`' that shows where the loop invariant must be true.

2. It then tries to prove that the loop invariants are *preserved* by an *arbitrary* iteration of the loop. This means that Dafny forgets everything it knows about the program state, and only assumes that the loop invariants and the test condition are true. It then tries to prove that if the loop's body is executed under these assumptions, the loop invariants will be reestablished.

3. After the loop exits, Dafny carries forward just two pieces of knowledge about the program's state: that the loop invariants are true, and that the test condition is false.

Since we have not specified a loop invariant in our code, all Dafny knows at the end of the loop is that `i` is not less than `A.Length`. This is not enough to be able to prove the postcondition.

Therefore, we must devise some loop invariants. What do we know about the state every time control passes the red arrow in Figure 4.1? What can we say about `i`, for instance? Upon entering the loop, `i` is 1, and by the end it is equal to `A.Length`. So one appropriate invariant is the following.

```
invariant 1 <= i <= A.Length
```

Dafny confirms that this is indeed a valid invariant, but it's not strong enough to deduce our desired postcondition. What it *does* give us is the ability to deduce that `i` must be equal to `A.Length` after the loop exits. The argument is that when the loop exits, we know that the above invariant is true, but we also know that the test condition is false. From `i <= A.Length` and `!(i < A.Length)` it follows that `i == A.Length`. Try adding

```
assert i == A.Length;
```

after the loop, and note that without the invariant above, Dafny cannot prove that this assertion will always be true.

Another observation: even though the value of `r` may change as we go round the loop, it is always the case that `r` is equal to one of the first `i` elements in the array.

```
invariant exists j :: 0 <= j < i && r == A[j]
```

This invariant *is* now strong enough to deduce our desired postcondition. The argument is that when the loop exits, we know the above invariant is true, but we also know that `i` is equal to `A.Length`. Combining these facts gives us exactly the postcondition we desired.

Our final source code is as follows.

```
method maxarray (A:array<int>) returns (r:int)
  requires A.Length > 0
  ensures exists j :: 0 <= j < A.Length && r == A[j]
{
  r := A[0];
  var i := 1;
  while i < A.Length
    invariant 1 <= i <= A.Length
    invariant exists j :: 0 <= j < i && r == A[j]
    decreases A.Length - i
  {
    if r < A[i] {
      r := A[i];
    }
    i := i+1;
  }
  assert i == A.Length;
}
```

As an exercise: devise another postcondition that says `r` is greater than or equal to every element in `A`. Devise a related loop invariant that allows Dafny to prove that your new postcondition is always true.

4.2 Predicates

Dafny provides *predicates* as a way to define abbreviations for common expressions. For instance, in our maxarray program, we can define a predicate called contains

```
predicate contains(A:array<int>, v:int)
  reads A
{
  exists j :: 0 <= j < A.Length && v == A[j]
}
```

and then use this predicate to simplify our postcondition.

```
ensures contains(A,r)
```

Note that predicates are required to specify which arrays they can read, via the **reads** keyword. (The reason is: if Dafny knows that a particular array is *not* read by a predicate, then it knows that any writes made by the program to that array cannot affect whether or not the predicate is true. This may reduce the number of times Dafny has to keep re-evaluating the predicate.) Methods don't have to provide **reads** clauses. However, had our maxarray method *written* to A rather than just *read* from it, it would need **modifies** A as part of its specification.

Better still, we can define a generalised contains_upto predicate

```
predicate contains_upto(A:array<int>, hi:int, v:int)
  reads A
  requires hi <= A.Length
{
  exists j :: 0 <= j < hi && v == A[j]
}
```

that is true when v is among the first hi elements of A. This predicate needs a precondition: that hi does not exceed A.Length. We can use this predicate to make our loop invariant more readable.

```
invariant contains_upto(A, i, r)
```

Observe, by the way, that contains_upto(A, A.Length, r) is then equivalent to contains(A, r).