The Use and Anatomy of an Auto-Active Verifier

Rustan Leino

Lecture 1 - *June 30, 2026*

1 Auto-Active Verification

The term *auto-active* is a somewhat oxymoronic shorthand for a verification style that combines automated (machine-generated, in the standard sense) and interactive (human-guided, with computer assistance) verification.

When building a verifier, there are several possible approaches. Typically, programs are translated into a mathematical representation, such as a collection of logical formulas, called an *encoding*. Program properties are then expressed and proved by reasoning within this mathematical structure using rewrites, proof rules, or other logical techniques. These proofs can be carried out in an interactive proof assistant, where the user works directly with the encoding. While the encoding may be well suited to the tactics provided by the proof assistant, it is generally not accessible to the average programmer.

The high-level idea behind tools such as Dafny is to lift the programmer's interaction with the verifier from the level of the mathematical encoding to the level of the programming language itself. Rather than reasoning directly about the encoding, users write specifications and proofs alongside the program. In this auto-active verification style, user interaction takes place at the source-language level, embedding specifications and proofs directly within the program.

2 Adoption of Dafny

2.1 Education

Dafny is used in educational settings for introductory courses in program verification, specification, and general formal reasoning. For over a decade, Dafny has been taught at universities such as

Compiled By:
Namratha Gangamreddypalli, Jasper Geer, Max Gross, Abigail Pribisova,
Hans Sardar, and Konami Shu

Cornell, Imperial, and Maryland.

2.2 Industrial use

A prominent use case for Dafny has been the verification of the AWS authorization engine. In AWS, incoming requests must be checked against authorization policies before they are processed. This policy evaluation is implemented in Dafny and verified against a specification. Since AWS processes billions of requests, even a very small error rate can affect a large number of users.

At the beginning of the project, there were concerns that adding verification might slow request handling (with up to a 10% slowdown considered acceptable). Instead, the verified implementation turned out to be roughly three times faster. The primary reason was that the algorithms implemented in Dafny were more efficient, and the formal guarantees made developers more willing to adopt algorithms that they might otherwise have considered too risky. In this sense, verification can enable more confident, or "fearless," programming.

2.3 Research Projects

Beyond industrial applications, Dafny has been used in a number of research projects on formally verified systems. Microsoft projects such as *Ironclad*[1] and *IronFleet*[2] use Dafny to establish strong correctness guarantees for practical software systems, while *Armada*[3] develops methodologies for verifying concurrent programs.

Dafny has also been used for meta-theory. An upcoming paper at ITP 2026 formalizes the verification condition generator for B3, a new intermediate verification language. The work proves that B3's verification condition generator is sound with respect to a formal semantics of the language, with the semantics, proofs, and implementation all written in Dafny. This work shows the potential for Dafny to be a useful vehicle for programming languages meta-theory. While Dafny does not use B3 in its implementation, B3 is considered a "next-generation" Boogie and there are some initial efforts to integrate it into the implementation of Dafny. As a result, this work would be strong example of using Dafny to reason about itself and its verification condition generation.

3 Examples

3.1 Double

Procedures in Dafny are known as *methods*. A programmer declares a method as in Figure 1 with a name, input parameters, and output parameters. Both input and output parameters are bound in the body of the method; this means that output parameters, like `r`, can be assigned and read

```
1  method Double(x: int) returns (r: int)
2      ensures r == 2 * x
3  {
4      if x < 10 {
5          var y := 3 * x;
6          return y - x;
7      } else {
8          assert 0 <= x;
9          r := 0;
10         var i := 0;
11         while i < x
12             invariant 0 <= i <= x && r == 2 * i
13         {
14             r := r + 2;
15             i := i + 1;
16         }
17     }
18 }
19
20 method Triple(x: int) returns (r: int)
21     ensures r == 3 * x
```

Figure 1: A program that doubles its argument

inside the body as on line 14. `return` is syntactic sugar for assignment to the output parameter; an example is on line 6. Local variables may be introduced with `var`, as on line 5. Dafny supports control flow structures such as `if-else` statements, loops, and procedure calls.

The verifier does not require that every method have a body; take, for example, the method `Triple` on line 20. This is because Dafny reasons about method calls in a modular fashion, considering only the specification of that method. In the case of `Triple`, this specification is that the method always returns its argument times 3. This postcondition is described in an `ensures` clause, as is found on lines 2 and 21. Note that if we were to compile this program, `Triple` would be required to have an implementation.

Dafny checks that the postcondition of each method is satisfied by its implementation. However, without line 12, Dafny will reject this program. The reason is that Dafny requires us to supply a *loop invariant*, which is a condition that holds prior to each iteration of the loop. The loop invariant is assumed at the beginning of the loop body, for each iteration, including the final one in which the loop condition is false and control flow exits the loop. Often, the proper loop invariant is related to the postcondition of the method. In this case, the invariant contains the postcondition with `i` substituted for `x`.

3.2 Min

Let us define a method that takes a sequence of integers and returns its minimum element. We might first come up with the following program:

```
1  method Min(s: seq<int>) returns (m: int)
2      ensures forall i :: 0 <= i < |s| ==> m <= s[i]
3  {
4      m := 0;
5      var n := 0;
6      while n < |s|
7          invariant 0 <= n <= |s|
8          invariant forall i :: 0 <= i < n ==> m <= s[i]
9      {
10         if s[n] < m {
11             m := s[n];
12         }
13         n := n + 1;
14     }
15 }
```

To make what this method does clearer, we can add a postcondition to specify that `m` is an element of `s`. The invariant in line 11 came straight from this postcondition. We need to initialize `m` with `s[0]` to satisfy this specification.

We also introduce `requires` to describe the precondition of the method.

The corrected program is shown below:

```
1  method Min(s: seq<int>) returns (m: int)
2      requires |s| != 0
3      ensures forall i :: 0 <= i < |s| ==> m <= s[i]
4      ensures m in s
5  {
6      m := s[0];
7      var n := 0;
8      while n < |s|
9          invariant 0 <= n <= |s|
10         invariant forall i :: 0 <= i < n ==> m <= s[i]
11         invariant m in s
12     {
13         if s[n] < m {
14             m := s[n];
15         }
16         n := n + 1;
17     }
18 }
```

```
function Sum(s: seq<int>): int {
    if |s| == 0 then 0 else s[0] + Sum(s[1..])
}

method ComputeSum(s: seq<int>) returns (r: int)
    ensures r == Sum(s)
{
    r := 0;
    var i := 0;
    while i < |s|
        invariant 0 <= i <= |s|
        invariant r == Sum(s[..i]) // Dafny cannot prove that this is maintained.
    {
        r, i := r + s[i], i + 1;
    }
}
```

Figure 2: `ComputeSum`

It is common to change specifications when writing Dafny programs. The first specification we write is often not sufficient, and refining it is a normal part of the verification process.

3.3 Sum

Before looking at the example, we introduce two distinctions: methods vs. functions, and statements vs. expressions.

Dafny distinguishes between methods and functions. Functions correspond to mathematical functions: they are deterministic and cannot modify state, while methods can be non-deterministic and can have side effects. Methods are opaque, meaning they are reasoned about only through their specifications. On the other hand, functions can be either opaque or transparent.

Another thing to mention is statements and expressions. An expression return a value, while a statement may not. They have distinct syntax:

```
if .. {..} else {..}
```

is a statement, while

```
if .. then .. else ..
```

is an expression.

In the example, let us define a function `Sum` and verify the method `ComputeSum` actually returns the sum of the input. We might first come up with the program in Figure 2.

```
1  method ComputeSum(s: seq<int>) returns (r: int)
2      ensures r == Sum(s)
3  {
4      r := 0;
5      var i := 0;
6      while i < |s|
7          invariant 0 <= i <= |s|
8          // invariant r == Sum(s[..i])
9          invariant Sum(s) == r + Sum(s[i..])
10     {
11         // assert r + s[i] == s[0] + Sum(s[1..i + 1]);
12         // assert r + s[i] == s[..i + 1][0] + Sum(s[..i + 1][1..]);
13         // assert r + s[i] == Sum(s[..i + 1]);
14         r, i := r + s[i], i + 1;
15         // assert r == Sum(s[..i]);
16     }
17 }
```

Figure 3: `ComputeSum` corrected version

However, Dafny cannot prove the invariant on line 12(Figure 2). The problem is that we are adding the value of the `i`-th element on the "wrong end". Dafny can generate `Cons` for us automatically, so it is easy to get from `Sum(s)` to `Sum(Cons(n, s))`. However, this does not work when we add `n` to the end of the list, as in line 14(Figure 2). In that case, we need to prove a lemma about `Snoc(s, n)`. We can see this by unfolding `Sum`(line 13 to line 11 in Figure 3). Line 13 is the precondition of the statement in line 15(Figure 3).

To make the verification go through, we replace the invariant in line 8 to the one in line 9(Figure 3). The comment-out lines are not needed for the verification.

3.4 BoolSort

In this example, the method sorts a Boolean array. In the sorted array, everything to the left is false, and everything to the right is true.

With `modifies a`, the method is allowed to modify the array `a`. The first postcondition specifies that the array is sorted: once a true appears, all following elements must also be true. The second postcondition ensures that the array contains exactly the same elements after execution as before. Here, `multiset(a[..])` converts the array into a multiset. `old(a[..])` refers to the heap of the array before the method was executed.

For the loop invariant, we will make sure that anything to the left of `lo` is false and anything to the right of `hi` is true.

The following program satisfies its specification and is successfully verified. The comments will be explained later.

```
1  method BoolSort(a: array<bool>)
2      modifies a
3      ensures forall i, j :: 0 <= i < j < a.Length ==> a[i] ==> a[j]
4      ensures multiset(a[..]) == multiset(old(a[..]))
5  {
6      var lo, hi := 0, a.Length;
7      while lo < hi
8          invariant 0 <= lo <= hi <= a.Length
9          invariant forall i :: 0 <= i < lo ==> !a[i]
10         invariant forall i :: hi <= i < a.Length ==> a[i]
11         invariant multiset(a[..]) == multiset(old(a[..]))
12         // decreases hi - lo, lo < a.Length && a[lo]
13     {
14         if !a[lo] {
15             lo := lo + 1;
16         } else if a[hi - 1] {
17             hi := hi - 1;
18         } else {
19             a[lo], a[hi - 1] := a[hi - 1], a[lo];
20             // we can remove these statements
21             hi := hi - 1;
22             lo := lo + 1;
23         }
24     }
25 }
```

We can also remove the assignments on lines 21 and 22, since the variables will be updated in the next iteration of the loop. However, if we remove them, we need to provide a termination measure, shown on line 12. The `decreases` clause specifies a value that must decrease with every loop iteration. In this case, we can use a lexicographic pair as the termination measure.

References

[1] Chris Hawblitzel, Jon Howell, Jay Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX - Advanced Computing Systems Association, October 2014. URL `https://www.microsoft.com/en-us/research/publication/ironclad-apps-end-to-end-security-via-a`

[2] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jay Lorch, Bryan Parno, Justine Stephenson, Srinath Setty, and Brian Zill. Ironfleet: Proving practical distributed systems correct. In *Proceedings of the ACM Symposium on Operating Systems Prin-*

ciples (SOSP). ACM - Association for Computing Machinery, October 2015. URL https://www.microsoft.com/en-us/research/publication/ironfleet-proving-practical-distributed

[3] Jay Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. Armada: Low-effort verification of high-performance concurrent programs. In *International Conference on Programming Language Design and Implementation (PLDI)*, pages 197–210. ACM, June 2020. URL https://www.microsoft.com/en-us/research/publication/armada-low-effort-verification-of-high-