# The Faultless Way of Programming

## Principles, Patterns, Practices, and Peculiarities for Verification in Dafny

James Noble
kjx@programming.ac.nz
Creative Research & Programming
Wellington, New Zealand
*Also* Australian National University

Charles Weir
charles.weir@lancaster.ac.uk
Lancaster University
Lancaster, UK

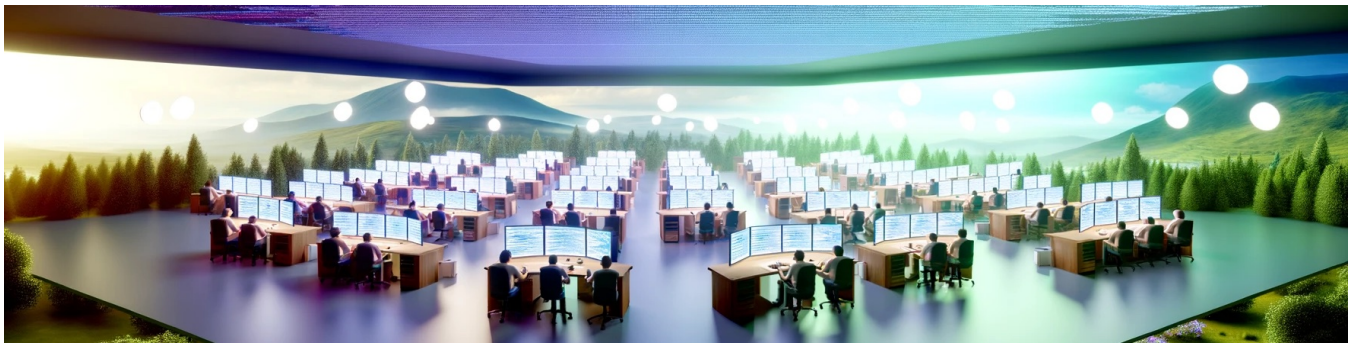

**Figure 1.** Teams of developers in perfect harmony

## Abstract

There is one faultless way of programming. It uses computer intelligence to validate computer code: formal verification. Yet for developers this faultless approach has remained alien, incomprehensible, and many miss out on its proven benefits. This set of patterns introduces Dafny to developers. Dafny provides a powerful way to incorporate formal verification into software that is integrated with languages like Java and C#, generating code that is provably free from defects and problems.

The patterns range from the Dafny design philosophy to concepts like ghost variables and implementation details such as the use of generative artificial intelligence. By offering an accessible approach to a difficult subject, they support developers in producing faultless code.

***Keywords:*** Patterns, Dafny, Correctness, Verification


*EuroPLoP 2024, July 3–7, 2024, Irsee, Germany*

ACM ISBN 979-8-4007-1683-6/24/07
https://doi.org/10.1145/3698322.3698340

**ACM Reference Format:**
James Noble and Charles Weir. 2024. The Faultless Way of Programming: Principles, Patterns, Practices, and Peculiarities for Verification in Dafny. In *29th European Conference on Pattern Languages of Programs, People, and Practices (EuroPLoP 2024), July 3–7, 2024, Irsee, Germany*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3698322.3698340

### Introduction

This paper is our experimental attempt at using the patterns form to address formal verification. Our intended readers are programmers wanting to learn formal verification, or at least to get some understanding of why it may be important in the future. So you might be a student towards the end of an undergraduate computer science qualification, or a practitioner who has not forgotten everything from your course on logic twenty years ago. Either way, you may be wondering how this could be relevant to your job in future.

Well, yes, it may well be. With the increasing importance of cybersecurity, and defect-free code, achieving code that does exactly what it says and nothing else is increasingly important. Verification tools are increasingly powerful, and artificial intelligence is making them easier to use. Formal verification is moving from only the realm of those with military budgets, to anyone who needs certainty.

This paper has patterns at different scales, ranging from a first, very general pattern that seeks to motivate the principle of **Formal Verification**, through to patterns that describe the thought process of developing Dafny programs. Being singers, we have named patterns after songs. We have drawn on the wealth of resources at `dafny.org`.

# 1 Pattern: Protocologorically Correct

*A.k.a. Formal Verification*

> We've a system to protect,
> Or the kingdom will be wrecked,
> Checked and double checked,
> And protocologorically correct.
> *From The Slipper and the Rose,*
> *by Richard M. Sherman and Robert B. Sherman.*

**Problem:** How can we be **as sure as possible** a program is correct?

In many systems, a small part of the code that is critical to the working of the whole. Examples include cryptographic algorithms, implementations of communications protocols, primitives to synchronise multithreaded working, control of vital hardware like medical devices or aircraft hardware, and much else. The code might be in any one of a number of languages, but it is essential that it works correctly.

You might write the code with extensive tests, ranging from comprehensive unit tests, through functional tests and system tests. That's great, but as Edgar W. Dijkstra wrote in "On the reliability of programs", [7]. *Program testing can be used very effectively to show the presence of bugs but never to show their absence.* Indeed, it is impossible to create enough unit tests and functional tests to be sure all conceivable cases have been covered.

Yet, in this case the amount of critical code is relatively small — hundreds of lines perhaps — or your development team is relatively large. And the algorithm is well understood, and probably documented in detail.

**Solution:** Write mathematical specifications of your program's expected behavior, then automatically verify that your program obeys your specification.

Formal verification tools analyse code *before it runs* — typically inside the development environment, or as a standalone compilation phase. For this to work, programs must contain not only executable statements, but also specification statements that describe how your code is supposed to behave.

The program may also require further assertions that relate the executable and specification statements: assertions describing the functionality they implement, that they will not get stuck in a loop, that buffers will not overflow, and that multi-threading problems will not happen.

These specification statements and assertions provide a different 'lens' on the code, and allow the program verification tool — the *verifier* — to check that the code conforms to those descriptions, often in remarkably complex ways.

Assuming a verifier runs successfully and approves the code, it confirms that executing the program will implement the specified behavior. For this to work in practice, programs, specifications, and internal assertions must typically be developed together, or top down from specifications to code — rather that taking an existing program and hoping to somehow verify it meets a specification *post-hoc*.

**Consequences:**

Verification success does *not prove the code works correctly in every way we want it to*; but it does prove that all the specification constructs in the code are satisfied. So the skill in formal verification programming is devising constructs that check everything that needs to be verified about the code.

Over the past 50 years, verification has made a number of significant breakthroughs, first with the development of interactive theorem provers in the 1960s [24], and then, at the turn of the millennium, with the development of Satisfiability Modulo Theory (SMT) solvers [3] — and (of course) ChatGPT and similar may offer at least a quantum leap in our ability to use such tools [23].

This means formal techniques must be an essential part of software engineering education. Unfortunately, formal methods are routinely hated by students due to perceived difficulty, mathematicity, and practical irrelevance — and by management staff for all those reasons plus the difficulty of finding academics willing (or even able) to teach the material [25, 26].

**Known Uses:**

CompCert C [2023] is a formally verified C compiler that won the ACM Software Systems award in 2021: a comprehensive study Yang et al. [30] found no bugs in the CompCert C compiler compared to the GCC [10] and LLVM [18] toolchains. This study motivated Airbus to adopt CompCert C to help ensure safety and enhance aircraft performance [9]. The seL4 project [14], awarded the ACM Software System Award in 2022, resulted in a formally verified high-assurance, high-performance operating system microkernel employed to protect an autonomous helicopter against cyber-attacks [27]. In order to secure communication, both Chrome and Android use formally verified cryptographic code [8]. Mozilla incorporated its verified cryptographic library for Firefox performance improvement [13].

**Implementation:**

As with contemporary general-purpose programming languages, formal verification tools and languages fall into two distinct groups: functional languages, which avoid mutable state and rely on higher-order ("dependent") types, such as Agda, Idris, and Roc (formerly Coq); and imperative/object-oriented languages which support mutable state and rely on Hoare logics. The imperative languages fall into two distinct types: special purpose languages designed for verification, such as Dafny or Whiley, and tools designed to verify programs written in other programming languages, such as JML for Java, SPARK Ada, Verifast for C, and Prusti for Rust [17].

## 2 Pattern: Strange but True

*A.k.a. Built-in Specification*

Strange, dear, but true, dear
*Song by Cole Porter*

**Problem:** How can we write a program that can be verified?

You have identified that you are have the situation described in the previous pattern: a relatively small, key, section of your code that must be algorithmicly correct. Yet...

- Chances are you already know at least one common imperative programming language. Why learn a new language if you don't have to?
- Verification tools can generally only cope with subsets of existing languages.
- Since most languages don't support specification statements, they have to be expressed in clunky form either as special comments or as annotations.

Indeed, most languages only support specification checking as 'assertions', run-time checks that the specification holds. Yet, this run-time checking takes time, and worse, it causes the software to fail if it finds an error—something very irritating to the users. Much better would be if we could be sure that the failure could not happen at all.

**Solution:** Validate the contracts using an SMP solver, and build it into the programming language

Special purpose languages can integrate specification statements tightly into programs, and verifiers tightly into the toolchains. Dafny [17, 21] is one such language for formal verification. Dafny was first developed in Microsoft Research (MSR) [22]; it is currently being developed with the support of the Amazon Automated Reasoning research group [1]. Dafny is statically typed, with programming language statements looking a bit like JavaScript or C#. Dafny also includes features drawn from imperative, functional, and object oriented programming. Dafny programs can be transpiled into C#, JavaScript, Python, C++, Java, or Go, and, critically, is well supported in Visual Studio Code [5].

Technically, the distinguishing feature of Dafny is that it supports code verification via design by contract [19, 20]. Dafny is based on the mathematical framework of Floyd-Hoare logic [12], where program executions are sets of states linked by executable statements, and the formal definition of each statement describes the mathematical relationship between preceding and succeeding states. As well as the usual imperative (variables, assignment, functions, data types, conditionals, loops) and object-oriented language features (classes, constructors, methods, fields), Dafny also includes verification constructs, such as method preconditions and postconditions, loop variants and invariants, assertions, and lemmas.

To develop a verified program, developers write Dafny code along with the specifications (pre-and post-conditions), and then add loop invariants and assertions that help the Dafny verifier prove the correctness of their code. While coding in VS Code, the Dafny static program verifier checks the functional correctness based on developers' defined specifications and annotations — and makes those checks as you type, like a spelling or grammar checker.

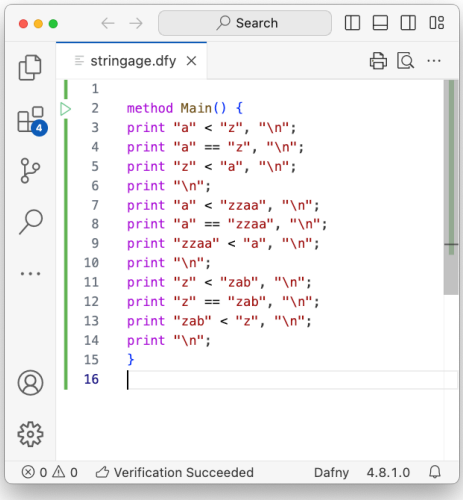


**Figure 2.** Dafny proving a simple program. Note the "Verification Succeeded" message, and the green bar on the left that shows verification status of each line in the program

**Implementation:**

To prove that methods' specifications hold, the Dafny program verifier first transforms the code into an intermediate verification representation [15] that encodes the verification conditions in predicate calculus [16], and then invokes the Z3 SMT solver [6] to prove the verification conditions. The validity of these verification conditions implies the correctness of a program's code [16].

Specification constructs can be very complex, and it is possible, even easy, to get them wrong or incomplete. So formal verification is usually used in conjunction with other ways of ensuring correctness:

- Code review, on 'pull request' or by pair programming, uses human checking.
- Unit testing uses automated testing; and
- System testing use human testing of the requirements.

Dynamic checking is an alternative approach [19, 20] that checks specifications and assertions at run-time, and aborts the code if the checks fail. Runtime assertion checks are rarely used in conjunction with formal verification, although recent research is investigating the integration of static proof and dynamic checks [2, 29].

## 3 Pattern: Doh-Re-Mi

*A.k.a. Design by Contract*

> Let's start at the very beginning,
> A very good place to start.
> *Do-re-mi from The Sound of Music, by Oscar Hammerstein II*

**Problem:** How can we describe what our program should do?

For most programs, there are a bunch of things we would like them *not* to do: not to destroy the world; not to trigger the third world war; and not to leak all our bank account details to websites selling mail-order brides, recruitment firms masquerading as social networks, or advertising corporations disguised as search engines. More pragmatically, we would like our programs not to suffer race conditions, not access array elements that don't exist, and not to indirect through null pointers [11].

While the latter two problems can be addressed by type systems and higher level memory handling [28], other problems are more difficult to deal with:

- Before we can say if a program is correct, we need to know what being correct means for that program.
- We have a task our code needs to do;
- We need to be sure our program always carries out its tasks correctly.
- We hope to use a program verified to give us that assurance.

**Solution:** Specify what we want each function to achieve

Use design-by-contract [19] to write method pre- and post-conditions to describe the task your methods should perform. A precondition (a Dafny **`requires`** clause on a method definition) describes what *must* be true of any program state immediately before the method is called: a postcondition (a Dafny **`ensures`** clause on a method definition) describes what *must* be true of any program state immediately after the method has returned. Bertrand Meyer recognised the importance of procedure specifications way back in his famous article in Computer magazine from 1992 [19]. See Table 1.

Pre- and post- conditions have a double purpose. To verify a method body, Dafny assumes pre-conditions and must prove the body establishes the post-conditions, while to verify a method call, Dafny proves the pre-conditions then assumes the post-conditions.

**Example:**

Here's an example of a Dafny swap method, and using it to build a three-element sorting network. What's actually pretty damn cool is that given those specifications (postconditions) for swap and sort, Dafny is able to prove that the sort does actually, well, sort things correctly.

```
method swap(a : nat, b : nat)
   returns (y: nat, z: nat)
   ensures y <= z
   ensures (y == a && z == b) || (y == b && z == a)
{
   if a <= b { y, z := a, b; }
    else { y, z := b, a; }
}

method sort(a : nat, b : nat, c : nat)
  returns (x : nat, y: nat, z: nat)
   ensures x <= y <= z
   ensures multiset{a,b,c} == multiset{x,y,z}
{
  x, y := swap(a,b);
  y, z := swap(y,c);
  x, y := swap(x,y);
}
```

**Table 1.** Bertrand Meyer's original example of a contract

**Table 1. Example contract.**

| Party | Obligations | Benefits |
|---|---|---|
| Client | Provide letter or package of no more than 5 kgs. each dimension no more than 2 meters. Pay 100 francs. | Get package delivered to recipient in four hours or less. |
| Supplier | Deliver package to recipient in four hours or less. | No need to deal with deliveries too big, too heavy, or unpaid. |

**Known Use:**

Amazon's AWS has a rules-based permission system that validates calls to an AWS API. There are typically 2 billion such calls per second. Getting such rules correct is vital to their customers' trust in the system and so the core reasoning engine is implemented in Dafny.

**Implementation:**

Dafny reasons about each method individually, only looking "inside" a method body to check it fulfills its specifications. Only those specifications (*requires* and *ensures* clauses) are visible outside a method. This is key to Dafny's analyses scaling: complex, detailed analyses are only ever required of one method body at a time.

Dafny's `old` construct in preconditions accesses the value of a variable before a method was called. For example, here the postcondition says the `Inc` method will increase the counter by one unit.

```
method Inc()
 ensures counter == old(counter) + 1
 {
  counter := counter + 1;
 }
```

The *assume* statement provides global domination; Dafny will blindly accept *assumptions* from that point on, even when they contradict earlier assertions or deductions.

```
 x := 0;
 assume x == 1; // Doesn't cause a problem
```

```
assert x == 1;
```

Dafny reports warnings for assume statements, so programs that use them don't count as verified. We can use *assume* statements to 'patch' programs during development, and remove them (or replace with assertions) once the program is verified.

## 4 Pattern: With a Little Help

I get by with a little help from my friends
*Song by Paul McCartney of the Beatles*

**Problem:** How to make a function verifiable?

There comes a time in every programmer's life when dealing with individual objects isn't enough: one needs to deal with collections of objects; and indeed to process each element of a collection, perhaps to make a new collection, but equally perhaps to reduce the collection down to some descriptive statistics, or even a single datum, that in some sense characterises the collection as a whole.

For example, given a non-empty Dafny array of integers:

```
s := [100, 200, -37, 45, -19, 56, 657, -2]
```

how might we write a method to return the smallest?

First, in Dafny, we want an independent logical specification of what "the the smallest in the array" actually might be.

Specifically, we can say two things about it: all the elements in the array must be larger than or equal to it; and it must itself be in the array. So in Dafny, that becomes:

```
method FindSmallest(s: array<int>)
     returns (min: int)
  requires s.Length > 0 // Non-empty array
  ensures forall k ::
       0 <= k < s.Length ==> min <= s[k]
  ensures exists k ::
       0 <= k < s.Length && min == s[k]
{
   // Something goes here.
}
```

Now as software developers, it probably won't take us very long to work out an implementation for this:

```
{ min := s[0];
  for i := 1 to s.Length
  { if s[i] < min
    { min := s[i]; }
  }
}
```

Unfortunately, Dafny is not able to verify this method's behaviour, even though it is obvious to a human that it will return the correct result. In VSCode, a large red X appears by the function, and by hovering the mouse over the red-underlined areas we see a host of error messages including the following:

```
Error: a postcondition could not be proved on this return path
Could not prove: forall k :: 0 <= k < s.Length ==> min <= s[k]
```

**Solution:** Provide intermediate invariants and assertions to help Dafny figure it out

Loops are hard for the theorem prover to manage, so we need to add steps to help it. In this case, the prover cannot prove that all the elements in the array are greater than or equal to the result. So we add a 'loop invariant', as an intermediate step to guide it. This ensures that the prover both checks that the invariant is true as the loop is entered, and that it can use the invariant to prove further constraints within the loop. An ultimately intelligent prover will be able to construct its own invariants on the fly, but until we have that we have to supply them for ourselves.

```
{ min := s[0];
  for i := 1 to s.Length
  invariant forall k :: 0 <= k < i ==> min <= s[k]
  { if s[i] < min
    { min := s[i]; }
  }
}
```

Given the preconditions, Dafny can now figure out that the invariant is true at the start of the first loop, and then looking at the loop implementation, it can prove that it is true thereafter. The 'loop invariant' acts as a crutch for the theorem prover.

Of course, Dafny also complains that the other postcondition cannot be satisfied either, so we add a corresponding further loop invariant:

```
{ min := s[0];
  for i := 1 to s.Length
  invariant forall k :: 0 <= k < i ==> min <= s[k]
  invariant exists k :: 0 <= k < i && min == s[k]
  { if s[i] < min
    { min := s[i]; }
  }
}
```

And the job's done. Dafny can now prove the whole method; and we can sure that the implementation will invariably produce results that satisfy the 'ensures' conditions so long as its 'requires' conditions are satisfied.

**Implementation - Refactoring:**

Astute programmers may notice the repetition in the invariant clauses. We can use Dafny predicates to refactor out Boolean expression.. Predicates may require preconditions, like methods, but also need to tell Dafny when they read from memory (don't ask!):

```
predicate SmallestInArray( s: array<int>,
      length: int, min: int )
// Answers whether min is the smallest value
// in the first length elements of array s:
requires length <= s.Length
reads s
{
   forall k :: 0 <= k < length ==> min <= s[k] &&
   exists k :: 0 <= k < length && min == s[k]
}

method FindSmallest(s: array<int>)
     returns (min: int)
// Answers the smallest element in array s:
  requires s.Length > 0 //precondition
  ensures SmallestInArray( s, s.Length, min )
{ min := s[0];
  for i := 1 to s.Length
    invariant SmallestInArray( s, i, min )
  { if s[i] < min
    { min := s[i]; }
  }
}
```

**Implementation - Using Generative AI:**

Working out the intermediate steps varies from difficult to almost impossible, since often the only help Dafny provides is to indicate which postcondition is causing the problem.

Fortunately we now have good support, in the form of Generative Large Language Models. ChatGPT 4.o is remarkably good both at offering suggestions why the problem has occurred, and at offering solutions—albeit often not very good ones. We can integrate it into the system as CoPilot, or use the text interface directly.

Dafny is ideally suited for Generative AI queries, since each method is verified separately. So it is easy to pass the full context for the query: the method's code, the programmer's intention, and any error and context information.

For example, using ChatGPT directly, the query:

```
In ```method FindSmallest(s: array<int>)
   returns (min: int)
  requires s.Length > 0 // Non-   empty array
   ensures forall k ::
      0 <= k < s.Length ==> min <= s[k]
   ensures exists k ::
      0 <= k < s.Length && min == s[k]
{  min := s[0];
   for i := 1 to s.Length
   { if s[i] < min
      { min := s[i]; }
      }
   }```
Dafny cannot prove
   ```ensures forall k ::
       <= k < s.Length ==> min <= s[k]```.
How do I fix this, please?
```

returns helpful suggestions to add the loop invariant. We can extend queries to include error messages or other information such as 'despite what you just told me', which also seem to help.

It is trivial to validate the suggestions from ChatGPT; so long as the executed code and external conditions are unchanged, ChatGPT's suggestions, such as new loop invariants, can only affect verification. So if the Dafny verifier accepts them, they are a correct solution.
```

## 5 Pattern: Ghostbusters!

Who you gonna call? Ghostbusters!
*Song by Ray Parker Jr.*

**Problem:** How can I prove other properties of the code?

Some features of the code cannot be captured by postconditions of the kind we have seen so far. For example, if we are writing an encryption algorithm, we want to prevent the 'side channel attack' that becomes possible when different kinds of input cause the algorithm to take different amounts of time. Attackers can measure how long it takes to decrypt different inputs and deduce encryption keys from that. Which shows laudable cleverness to our minds, but hey!

For example, here is an example of the kind of thing such an encryption algorithm would need to do: a method to look up the index of a value in a sequence of values. To be a bit more elegant than the previous pattern, the implementation uses Dafny's sequence operations: 'in' and '!in' for inclusion, '|s|' for the length of the sequence, and 's[0..n]' meaning the first n elements.

```
method IndexOf( s: seq<int>, value: int)
           returns (index: int )
// Answers the index of value in s
requires value in s
ensures 0 <= index < |s|
ensures s[index] == value
ensures value !in s[0..index]
{
   for i := 0 to |s|
      invariant value !in s[0..i]
      { if s[i] == value
         { return i;
         }
      }
   return -1; // Not reached.
}
```

How might we assert that the amount of time the method takes depends only on the length of s?

**Solution:** Use ghost code to model non-functional properties

While we cannot know how long the method will take, we can be reasonably sure that the time taken by operations like getting and calculating values will not depend on the values involved. Instead, the processing time will depend only on the different paths of execution through the method. So we add 'ghost variables', which do not generate any run-time code, and are only used by the Dafny prover. These count the number of different paths of execution, so we can use an assertion to state that these are as expected on exit, adjusting the method accordingly:

```
method IndexOf2( s: seq<int>, value: int)
              returns (index: int )
// Answers the index of value in s
requires value in s
ensures 0 <= index < |s|
ensures s[index] == value
ensures value !in s[0..index]
{
   ghost var numIterations := 0;
   ghost var numFirstBranch := 0;

   for i := 0 to |s|
      invariant value !in s[0..i]
      { numIterations := numIterations + 1;
         if s[i] == value
         { numFirstBranch := numFirstBranch + 1;
            break;
         }
      }
   assert numIterations == |s|;
   assert numFirstBranch == 1;
   return i;
}
```

Ghost code is widely used in Dafny. Ghost variables can model the history of operations on a data structure to prove a property like 'operations are performed in a FIFO order'; track properties like 'the algorithm will always terminate'; or simplify complicated invariants by caching values. There are also 'ghost functions'—like the predicate in the previous pattern—which are used only by the theorem prover and do not generate any code.

**Known Use:**

AWS provide an Encryption SDK for customer use, available on multiple platforms and languages. The core implementation is in Dafny, which does make at least some use of ghost variables.

**Implementation:**

Unsurprisingly, Dafny complains quite a bit about the above code. After a lot of discussion with ChatGPT as described in the previous pattern, we achieve the following implementation:

```
method IndexOfConstTime( s: seq<int>, value: int)
                  returns (index: int)
   requires value in s
   ensures 0 <= index < |s|
   ensures s[index] == value
   ensures value !in s[0..index]
{
   ghost var numIterations := 0;
   ghost var numFirstBranch := 0;

   var result := -1;
   for i := 0 to |s|
      invariant result == -1 ==> value !in s[0..i]
      invariant result < |s|
      invariant result >= 0 ==>
                  value !in s[0..result]
      invariant result >= 0 ==> s[result] == value
      invariant numIterations == i
      invariant result != -1 ==> numFirstBranch==1
      invariant result == -1 ==> numFirstBranch==0
      { numIterations := numIterations + 1;
         if (s[i] == value) && (result == -1)
         { numFirstBranch := numFirstBranch + 1;
            result := i;
         }
      }
   assert numIterations == |s|;
   assert numFirstBranch == 1;
   return result;
}
```

Notice that we have not only had to provide a large number of helpers for Dafny; we have also had to change the real implementation to satisfy the timing constraints.

Purists will note that '(s[i] == value) && (result == -1)' actually involves multiple pathways since the second expression may not be evaluated. The solution, of course, is to use the bitwise '&' operator instead of the logical '&&' one. But Dafny's '&' does not work on booleans, so we shall need to fix it in the generated code. Sigh!.

Overall, though, Dafny's theorem prover has forced us to change our implementation, and we have proved the resulting implementation has the required timing property. Dafny has checked something that would be very hard to validate using testing, and proved it to be true. Yippee!

## 6 Conclusion

We have explored the philosophy of Faultless programming, introduced Dafny, and explored some of the benefits and approaches to using it. What might you do next?

We recommend you take the plunge, and learn Dafny properly! Set up a Visual Studio Code environment, download the Dafny support package, and get coding. There are excellent materials for learning the practical side of the language now you have the concepts:

We suggest you find a problem that interests you and work through it in a Dafny implementation. Look for a self-contained module, and one where the vital requirements are more functional than non-functional. And get Dafny coding!

Good luck!

## Acknowledgments

To the many people who have supported our work for well over twenty years:

To John Vlissides in particular, who remains very much alive in our hearts, and very much missed.

Charles, who had not heard of Dafny before writing this paper, thanks ChatGPT and all the example writers.

James thanks Rustan Leino and James Wilcox for all their help with Dafny; to our colleagues Marco Servetto for the "marcotron" weekly question system, to Royce Brown, Christo Muller, and the ECS technical staff for their support with the course automation; to Lindsay Groves, longtime custodian of Formal Methods at VUW through various iterations (COMP202, SWEN202, SWEN224, SWEN324); to the reviewers for their helpful comments; and above all to the students who choose to stay with SWEN324 in spite of everything.

This work was supported in part by the Royal Society of New Zealand Marsden Fund Grant VUW1815, CRP101, CRP2101, an Amazon Research Award, and a gift from Agoric.

## References

[1] Amazon. 2023. Automated reasoning. https://www.amazon.science/research-areas/automated-reasoning.

[2] Johannes Bader, Jonathan Aldrich, and Éric Tanter. 2018. Gradual Program Verification.. In *VMCAI*. 25–46.

[3] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. 2021. Satisfiability Modulo Theories. In *Handbook of Satisfiability - Second Edition*. Vol. 336. 1267–1329. https://doi.org/10.3233/FAIA201017

[4] CompCert. 2023. CompCert. https://github.com/AbsInt/CompCert. [Online], [Accessed: 2023-09-20].

[5] Dafny. 2023. dafny-lang. https://github.com/dafny-lang/dafny. [Online], [Accessed: 2023-09-20].

[6] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS*. 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[7] Edsger W. Dijkstra. 1970. On the reliability of programs (EWD303). In *E.W. Dijkstra Archive: the manuscripts of Edsger W. Dijkstra*. https://www.cs.utexas.edu/users/EWD/

[8] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2020. Simple High-Level Code For Cryptographic Arithmetic: With Proofs, Without Compromises. *ACM SIGOPS Oper. Syst. Rev.* 54, 1 (2020), 23–30. https://doi.org/10.1145/3421473.3421477

[9] Ricardo Bedin França, Denis Favre-Felix, Xavier Leroy, Marc Pantel, and Jean Souyris. 2011. Towards Formally Verified Optimizing Compilation in Flight Control Software. In *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*. Dagstuhl, 59–68. https://doi.org/10.4230/OASIcs.PPES.2011.59

[10] GCC. 2023. GCC, the GNU Compiler Collection. https://gcc.gnu.org/. [Online], [Accessed: 2023-09-20].

[11] C.A.R. Hoare. 2009. Null References: The Billion Dollar Mistake. Presentation to INFOQ Conference.

[12] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *CACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

[13] Kevin Jacobs and Benjamin Beurdouche. 2022. Performance Improvements via Formally-Verified Cryptography in Firefox. https://blog.mozilla.org/security/2020/07/06/performance-improvements-viaformally-verified-cryptography-in-firefox/. [Online], [Accessed: 2023-09-20].

[14] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David A. Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: formal verification of an OS kernel.. In *SOSP*. 207–220. https://doi.org/10.1145/1629575.1629596

[15] Claire Le Goues, K Rustan M Leino, and Michał Moskal. 2011. The Boogie verification debugger. *SEFM, LNCS* 7041 (2011), 407–414.

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

[17] K. Rustan M. Leino. 2023. *Program Proofs*. MIT Press.

[18] LLVM. 2023. The LLVM Compiler Infrastructure. https://llvm.org/. [Online], [Accessed: 2023-09-20].

[19] Bertrand Meyer. 1992. Applying 'Design by Contract'. *Computer* 25, 10 (1992), 40–51.

[20] Bertrand Meyer. 2009. *Touch of Class*. Springer.

[21] Microsoft. 2023. The Dafny Programming and Verification Language. https://dafny.org/. [Online], [Accessed: 2023-09-20].

[22] Microsoft. 2023. Microsoft Research. https://www.microsoft.com/en-us/research/. [Online], [Accessed: 2023-09-20].

[23] Md Rakib Hossain Misu, Cristina V. Lopes, Iris Ma, and James Noble. 2024. Towards AI-Assisted Synthesis of Verified Dafny Methods. In *Proc. ACM Softw. Eng. 1, FSE"*. https://doi.org/10.1145/3634763

[24] R.P. Nederpelt, J.H. Geuvers, and R.C. de Vrijer (Eds.). 1994. *Selected Papers on Automath*. North Holland.

[25] James Noble. 2024. Learn 'em Dafny. In *Dafny Workshop at POPL* (London, England). https://popl24.sigplan.org/details/dafny-2024-papers/11/Learn-em-Dafny

[26] James Noble, David Streader, Isaac Oscar Gariano, and Miniruwani Samarakoon. 2022. More Programming Than Programming: Teaching Formal Methods in a Software Engineering Programme. In *NASA Symposium on Formal Methods*.

[27] seL4 Project. 2023. The seL4® Microkernel. https://sel4.systems/. [Online], [Accessed: 2023-09-20].

[28] Marco Servetto, Julian Mackay, Alex Potanin, and James Noble. 2013. The Billion-Dollar Fix. In *ECOOP*. 205–229.

[29] Jenna Wise, Johannes Bader, Cameron Wong, Jonathan Aldrich, Éric Tanter, and Joshua Sunshine. 2020. Gradual verification of recursive heap data structures. In *OOPSLA*.

[30] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *PLDI*. https://doi.org/10.1145/1993498.1993532