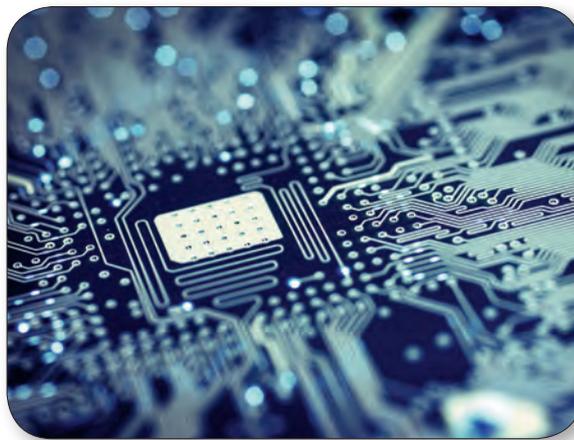


What you say is what you get: an Ada story

This article is contributed by AdaCore

Over the past 30+ years, Ada technology has matured into a unique toolset allowing programmers to achieve software reliability at a very affordable cost. It's available for small microcontrollers such as ARM Cortex-M, large x86 64 hosted systems, and many things in between. Time to give it a try?



■ At the risk of opening with platitudes, software development has become one of the most central engineering disciplines. Our lives are literally governed by software in quantities that defy imagination. As a matter of fact, our lives actually rely on this software. Drones, cars, medical devices, the list of things whose software can make the difference between life and death is growing by the day. How to avoid making this a list of threats? Truth is, software engineering is not about writing code. It's about specifying intent, implementing what's intended, and verifying that what's implemented corresponds to what's intended. And to do that, all environments and all languages are not equal.

Ada is a language that was first defined in the 80s, around the same time as C++. Unlike C++ though, which was designed as an evolution of C with extra expressivity capabilities, Ada was aimed at supporting the needs of high-criticality embedded software development. This led to a series of unique technical choices setting the language on a very different path. As a result, Ada tends to force programmers to be much more explicit when they write code, as the compiler will prefer to avoid compiling rather than guessing. Conversely, as opposed to other languages, Ada allows programmers to formally express intent that would be otherwise buried into comments.

This has two interesting consequences. First, many programming errors that would be difficult to avoid can be mitigated or eliminated. Second, it's actually possible to go much further in making explicit expectations and intent at the software levels and either automatically enforce it or verify correct implementation. Here's an example of what that may look like at low level:

```
type Percent is delta 0.1 digits 4 range 0.0 .. 100.0;
type Status is (On, Off, Error);

type Sensor is record
  Power    : Percent;
  Enabled  : Status;
end record
with Bit_Order           => High_Order_First,
  Scalar_Storage_Order => High_Order_First,
  Size                  => 32,
  Dynamic_Predicate     =>
    (if Sensor.Enabled = Off then Sensor.Power = 0.0);

for Sensor use record
  Power at 0 range 0 .. 15;
  Enabled at 0 range 30 .. 31;
end record;
```

This says quite a lot about a low-level data structure which is possibly used to map onto a device, or a data connection. It's a sensor structure composed of Percent, which is a

fixed-point value with a precision of 0.1, from 0 to 100 of 4 precision digits, stored in the first half-word of a 32 bits big-endian structure, and a flag that can only take 3 values, stored in the last 2 bits of that same structure. From this description, the compiler will actually verify that this representation can be implemented and if it does, guarantee that this specification will indeed be respected. Developers

familiar with development of low-level layers can already see macro and bitwise operations headaches vanishing thanks to these features. In addition to this structural information,

the type is associated with a constraint (or dynamic predicate) that states that when the flag Enabled is Off, Power has to be equal to 0. With assertion checking enabled, the compiler will generate consistency checks at appropriate places in the code, allowing the programmer to identify inconsistencies early.

On the opposite side of the spectrum, specification can be used to express functional constraints. The most common one is probably the precondition. The idea is that an assertion can be defined to be true when calling a function (or subprogram). A postcondition will specify the output of a function - its behaviour. Take for example:

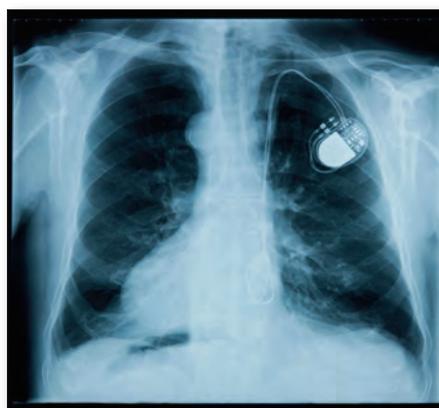
```
procedure Swap (A : in out An_
Array; I1 : in Integer; I2 :
in Integer)
    with Pre => I1 in A'Range
and I2 in A'Range,
        Post => A(I1)'Old =
A(I2) and A(I2)'Old = A(I1);
```

The above defines a simple swap procedure on two elements on the array. One may expect that the indexes are within the range of the array. To avoid out of range array access, it's tempting to write some defensive code within swap to detect the errant case and take appropriate measures - raise an exception or cancel the operation. But it should really be the responsibility of the caller to correctly call swap with two valid indexes. The precondition of this code allows the programmer to specify exactly that, thus removing the need for defensive code. It literally says I1 has to be within the range of A, and so does I2. On top of this, we also specify the outcome of the subprogram, that is the old value of A(I1) now equals to A(I2) and vice versa. So not only is there verification on the way in, to check that the code is properly called, but there's also verification at the point of return, to check that the operation is doing what is expected. Also, note the parameter mode in out on the array: that specifies that Swap will modify the array, as opposed to I1 and I2 which are in (i.e., only input).

Preconditions, postconditions and other kinds of functional contracts (such as dynamic predicates as shown earlier) can be verified in many different ways. The obvious one is to ask the compiler to generate actual checks. This can be done selectively. All assertions could be kept during testing for example, and only a minimal amount kept after deployment. Even when inactive, these contracts can

have merit just from the fact that they're written and accepted by the compiler - i.e. they refer to actual entities and consistent operations. And they also serve as useful information to the human reader.

There's however another way to do this verification that doesn't even involve executing the code: static analysis. Better yet, formal proof. The SPARK language is a subset of the Ada language that is fit for static verification. Within the subset, any potential error will be analyzed and those that may happen will be reported. Every contract will be looked at tak-



ing into account all possible values, and those that might not be valid will be reported. In particular, SPARK can guarantee that every single call always respects the preconditions, and that every single subprogram always respects its postcondition.

And there's more with Ada. Class hierarchies with verification that the behaviour defined at the root level is consistent with the behaviour of subclasses. Concurrency models with guarantees of absence of deadlocks. Specification of global variables - and more generally global states. Physical dimension consistency checking. Integer overflow elimination. The list goes on.

So who is using Ada today? While the language has a long-standing history in aerospace and defense, it's now frequently being adopted outside its original area by a growing number of companies in other embedded domains. Some are in the very enviable situation of having projects to start from scratch. Adopting Ada or SPARK is very easy then, it's merely a question of training and there's nothing fundamentally difficult for a typical

embedded developer. Most of those companies, however, have a significant investment in C or C++ code bases - or rely on components developed in those languages. So how do they migrate to Ada? Surely, bearing the costs of rewriting the whole code base offsets any benefits they may get.

Luckily, this is never a requirement. Ada is extremely amenable to language interfacing. As a matter of fact, often C and Ada compilers share most of their code (at least the part responsible for optimizing and generating assembly) making mixing those two an easy task. As an example, the swap function described already could actually come from a C library:

```
void swap (int * a, int i1,
int i2) {
    int tmp = a[i1];
    a[i1] = a[i2];
    a[i2] = tmp;
}
```

All we need to do is to tell Ada that this implementation is actually in C, specifying that it's imported, of a C convention, and linked to a symbol swap:

```
procedure Swap (A : in out An_
Array; I1 : in Integer; I2 :
in Integer)
    with Pre => I1 in A'Range
and I2 in A'Range,
        Post => A'Old (I1) =
A(I2) and A'Old (I2) = A(I1),
        Import,
        Convention => C,
        External_Name =>
"swap";
```

Hey, isn't that adding contracts to a C function? Note that although this interfacing code can be manually written, there are also binding generators available that would automatically take care of most of the burden (with the exception of addition of contracts). At the end of the day, only specific software components will be re-written in Ada, most legacy code can be kept. New modules developed in Ada can then be integrated with this legacy code. What you say is what you get. Software engineering is not about writing code, it's about specifying intent, implementing it and ensuring that what has been implemented corresponds to what has been specified, with as little effort as possible. ■