Ada vs SPARK

Ada is a general-purpose language like C++ or Java. It supports usual features of modern programing languages, like data encapsulation, object orientation, genericity, tasking… It has been defined in 1983 and has been evolved regularly until now (the last version is from 2012). What tells Ada apart from other general-purpose languages is that it was designed from the start with reliability, safety, and security in mind. Not surprisingly, Ada is used in domains where the correction of software is critical: space, avionics, air traffic control, railway, military. SPARK is a specialized subset of Ada targeting the use of formal methods, so that correction of software can be guaranteed. Hence, SPARK is used in the same domains as Ada, by actors who value the strong guarantees offered by formal methods. Because formal methods can be more cost effective than testing for getting high levels of confidence in the correction of software, new domains where software becomes critical like automotive and drones are increasingly attracted to solutions like SPARK. In particular, formal methods provide a better solution than testing for defending against security attacks that exploit software vulnerabilities.

One of Ada’s most prominent characteristics is a strong and powerful type system. Ada offers support for a wide range of user defined types. It includes signed (with an overflow semantics in computation) and modular integer types of various sizes, on which an additional range may be defined, as well as enumerated types and array types indexed by any integer type.

**type** My\_Index **is new** Integer **range** 1 .. 100;

*-- A signed integer type containing only integers from 1 to 100*

**type** My\_Color **is** (Red, Blue, Yellow);

*-- An enumerated type, which is not the same as an integer in Ada*

**type** My\_Array **is array** (My\_Index **range** <>) of My\_Color;

*-- An array indexed by any subrange of integers between 1 and 100 and containing colors*

Another feature provided by Ada that enhances reliability is that it automatically inserts runtime checks for a whole class of common erroneous behaviors. In particular, it checks dynamically for accesses outside of an array’s bounds (an error commonly known as buffer overflow) and for integer values outside of their type’s range. If a violation is found, an exception is raised at runtime. These checks can be disabled using a compiler switch for efficiency reasons.

I : My\_Index := <Result of a complicated computation>;

*-- An exception will be raised if I’s value is not between 1 and 100.*

A : My\_Array (1 .. 2) := (Red, Blue);

*-- A is an array of colors ranging from 1 to 2*.

C : Color := A (I);

*-- C is A’s Ith element. An exception will be raised if I is bigger than 2*.

The last version of Ada goes one step further in this direction by introducing support for contract-based programing. The most common form of contracts is preconditions and postconditions of subprograms. Those are Boolean expressions that serve as a contract between a caller and a callee. They should hold respectively before and after every call to the subprogram. Contracts are useful for developing safe and secure software in the sense that they both enhance readability and maintainability by documenting the subprograms and facilitate testing as they can be checked at runtime.

**procedure** Increase (X : **in out** Integer) **with**

Pre => X <= Max,

*-- It is the responsibility of every caller of Increase to check that its argument is less than Max before the call.*

Post => X > X’Old;

*-- It is the responsibility of Increase’s implementation to ensure that the final value of X will be strictly greater than its initial value.*

SPARK is a based on Ada, so, it was also designed with safety, security, and reliability in mind. What is more, it is targeted at formal verification. SPARK has evolved alongside Ada from its first version SPARK 83 based on Ada 83, to the last version SPARK 2014 based on Ada 2012. It has been designed to allow several forms of static verification. In particular, users can specify how information should flow through variables in the program as well as functional properties about its behavior. SPARK has been designed so that all these annotations, as well as absence of errors and exceptions at runtime, can be verified statically, and, if possible, without too much annotation burden to the user. To achieve this goal, some features of Ada that are not easily amenable to formal verification have been excluded from SPARK. Most notably, forbidden features include pointers (but references and addresses are allowed). This restriction is motivated by the important amount of annotations required to specify a program using pointers, as well as the difficulties of verification automation in this domain.

SPARK is not only a subset of Ada. It also introduces new features specifically dedicated to formal analysis. Among them, we find new contracts that enhance the user’s power of annotation and therefore the properties that can be formally verified. In particular, SPARK provides new contracts describing what are the variables used by the subprogram and how information flows from one to another.

**procedure** Swap\_X\_And\_Y **with**

Globals => (In\_Out => (X, Y)),

*-- Swap\_X\_And\_Y modifies the global variables X and Y.*

Depends => (X => Y, Y => X);

*-- The final value of X depends only on the initial value of Y and the final value of Y depends only on the initial value of X.*

It also allows defining a subprogram’s contract as a set of different cases, regrouping values on which the subprogram should have the same behavior, a bit like test cases.

**function** Absolute\_Value (X : Integer) **return** Natural **with**

Pre => X /= Integer’First,

*-- Absolute\_Value should not be called on the smallest value of Integer as it would cause an overflow*

Contract\_Cases => (X < 0 => Absolute\_Value’Result = - X,

X = 0 => Absolute\_Value’Result = 0,

X > 0 => Absolute\_Value’Result = X);

*-- Absolute\_Value behaves the same on three domains of X. On negative values, it returns the opposite, on 0 it returns 0 and on positive values it is the identity.*

Though Ada and SPARK are technically different languages, they work well together. First, the new features introduced in SPARK use Ada syntax for compiler specific additions (namely pragmas, aspects, and attributes). Then, Ada and SPARK can be mixed at a fine-grained level. For example, we can switch from Ada to SPARK between two packages or subprograms or inside a single package or subprogram (between a subprogram’s specification and its body or between a package declaration’s public and private parts for example). This possibility is in particular valuable to work around the restrictions introduced by SPARK by switching back to full Ada.

**package** Abstract\_Pointer **with** SPARK\_Mode **is**

*-- Here we are in SPARK*

**type** My\_Pointer **is private**;

*-- Users of this package cannot see what My\_Pointer is. They must use subprograms to access its content.*

**function** Access\_Pointer (P : My\_Pointer) **return** Value;

**function** Create\_Pointer (V : Value) **return** My\_Pointer;

**private**

**pragma** SPARK\_Mode (Off);

*-- We are now in Ada*

**type** My\_Pointer **is access all** Value;

*-- My\_Pointer is in fact a pointer!*

**end** Abstract\_Pointer;

To conclude, Ada and SPARK are a good duo for writing safe, secure, and reliable software. SPARK adds formal verification techniques on top of the dynamic verification performed in Ada. The most critical parts can be written in SPARK, allowing users to benefit from formal verification techniques, while the full expression power of Ada can be retained for parts where it allows a more straightforward or more efficient implementation. In a nutshell, there is no one winner in the Ada vs SPARK comparison, but two: a language (Ada) offering strong features to ensure reliability, safety, and security with traditional testing and review techniques; and a compatible language (SPARK) offering strong features to guarantee reliability, safety, and security with formal methods.