print | close

What's the Difference Between Ada and SPARK?

Electronic Design

Claire Dross

Thu, 2015-11-19 11:03

Ada is a general-purpose language, like C++ or Java, supporting the usual features of modern programming languages, such as data encapsulation, object orientation, templates (called "generics"), exceptions, and tasking. Originally defined in 1983, it has undergone several revisions, the latest in 2012. What sets 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 correctness of software is critical: space, avionics, air-traffic control, railway, and military.

SPARK is a specialized subset of Ada designed to facilitate the use of formal methods, so that correctness of software or other program properties can be guaranteed with mathematics-based assurance. Therefore, SPARK is used in the same domains as Ada, by those who value the strong guarantees offered by formal methods.

Related

The Day of the Three Glitches

Ada/SPARK Fixes Crazyflie Nano Quadrotor

Requiem for a Bug – Verifying Software: Testing and Static Analysis

Because formal methods can be more cost-effective than testing in achieving high levels of confidence in software correctness, new domains where software is also critical, such as automotive and drone software, 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.

Ada

One of Ada's most prominent characteristics is a strong and powerful type system. Ada supports a wide range of user-defined types. It includes signed (with overflow semantics) and modular integer types (with wrapping semantics) of various sizes. Both types allow restrictions on the range of valid integers. Ada also supports enumeration types and array types indexed by any integer or enumeration type:

```
type My_Index is range 1 .. 100;
-- A signed integer type containing only integers from 1 to 100

type My_Color is (Red, Blue, Yellow);
-- An enumeration type, which is not the same as an integer in Ada

type My_Array is array (My_Index range <>) of My_Color;
-- A My_Array object is indexed by any value from My_Index (i.e., an integer between 1
```

```
ıd 100)
- and contains elements of type My_Color
```

Ada provides an important feature that enhances reliability: It automatically inserts runtime checks for a large class of common errors. Specifically, it dynamically checks 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 occurs, an exception is raised at runtime. If desired (for example, after verification provides confidence that the errors will not occur), these checks can be disabled for efficiency reasons:

```
I : My_Index := ;
-- 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 whose index range is 1 through 2 and where A(1)=Red and
A(2)=Blue

C : Color := A (I);
-- C is A's I-th element. An exception will be raised if I is greater than 2.
```

The current (2012) version of Ada goes one step further in promoting reliability by introducing support for contract-based programming. The most common forms of contracts are subprogram preconditions and postconditions. These are Boolean expressions that serve as contracts between callers and callees that must be true before and after every call to the subprogram, respectively. Contracts are useful for developing safe and secure software for several reasons. First, they enhance the program's readability and maintainability by supplying source-code documentation for a subprogram's assumptions and guarantees. Second, they enhance testing because 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.
Post => X > X'Old;
-- It is the responsibility of Increase's implementation to ensure that
-- the returned value of X is strictly greater than its initial value.
```

SPARK

SPARK, which is based on Ada, was also designed with safety, security, and reliability in mind, but it differs in that supports formal verification as well. SPARK has evolved alongside Ada from its first version, SPARK 83, which was based on Ada 83 (where it included a stylized comment syntax for verification-related annotations that augmented a subset of Ada), to the current version, SPARK 2014.

This latest version is a subset of Ada 2012 (and uses the contract mechanism built in to Ada) and allows several forms of static verification. In particular, the software developer can specify how information should flow through variables in the program and define functional properties about the program's behavior. All of these contracts, as well as the absence of errors and exceptions at runtime, can be verified statically.

To achieve this goal, some features of Ada that are not easily amenable to formal verification have been excluded from SPARK. Most notably, the forbidden features include pointers (but addresses are allowed). This restriction is motivated by the heavy syntax that would be required for a verifiable program using pointers, as well as the difficulties of automatic verification of such programs.

SPARK is not only a subset of Ada, but it also incorporates features that specifically support formal analysis. Among them are new types of contracts that enhance the user's power of annotation and, thus, the properties

t can be formally verified. For example, SPARK defines contracts describing which variables are used by a program and how information flows between the variables and the subprogram:

```
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.
```

SPARK also allows the programmer to define a subprogram's contract as a set of distinct cases, grouping values for which the subprogram should have the same behavior, in a manner similar to conventional 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 differently on three domains of X.
-- On negative values, it returns the opposite, on 0 it returns 0
-- and on positive values it return the same value as its input.
```

Using Ada with SPARK

Although technically Ada and SPARK are different languages, they work well together. The new features introduced in SPARK use standard Ada syntax (pragmas, aspects, and attributes) for its additional features. Ada and SPARK can be mixed at a fine-grained level—the programmer can combine Ada and SPARK code in different packages or subprograms, or inside a single package or subprogram (for example, between a subprogram's specification and its body, or between a package declaration's visible and private parts). This mixing helps alleviate the SPARK language restrictions. For instance, the programmer can use full Ada in places where the flexibility of pointers is more important than the ability to formally verify behavior:

```
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;
```

Ada and SPARK make an effective two-language team for writing safe, secure, and reliable software. SPARK adds static formal verification to the dynamic verification performed by Ada. The most critical parts of an application can be written in SPARK, allowing users to benefit from formal-verification techniques, while the full expressive power of Ada can be retained for those parts that require a more straightforward implementation.

In summary, Ada vs. SPARK is not a competition, but rather an effective and cooperative relationship: One language (Ada) offers strong features to provide confidence in reliability, safety, and security through traditional

ification methods (testing and review), and a compatible language (SPARK) offers strong features to
rantee reliability, safety, and security with formal methods.

Source URL: http://electronicdesign.com/dev-tools/what-s-difference-between-ada-and-spark