7
Interfacing with SPARK

It is often infeasible or even undesirable to write an entire program in SPARK. Some portions of the program may need to be in full Ada to take advantage of Ada features that are not available in SPARK such as access types and exceptions. It may be necessary for SPARK programs to call third-party libraries written in Ada or some other programming language such as C. Of course SPARK's assurances of correctness cannot be formally guaranteed when the execution of a program flows into the non-SPARK components. However, mixing SPARK and non-SPARK code is of great practical importance. In this chapter we explore the issues around building programs that are only partially SPARK. In Chapter 8 we look at how combining proof with testing can verify applications that are not all SPARK.

7.1 SPARK and Ada

In this section we discuss mixing SPARK with full Ada. Calling SPARK from Ada is trivial because SPARK is a subset of Ada and thus appears entirely ordinary from the point of view of the full Ada compiler. Calling full Ada from SPARK, however, presents more issues because the limitations of SPARK require special handling at the interface between the two languages.

7.1.1 SPARK *Mode*

Conceptually each part or construct of your program is either "in SPARK" or "not in SPARK." If a construct is in SPARK, then it conforms to the restrictions of SPARK, whereas if a construct is not in SPARK, it can make use of all the features of full Ada as appropriate for the construct. It is not permitted for SPARK constructs to directly reference non-SPARK constructs. For example, a subprogram body that is in SPARK cannot call a subprogram with a non-SPARK declaration. However, as declarations and bodies are separate constructs, it is permitted for a SPARK subprogram body to call a subprogram with a SPARK declaration even if the body of the called subprogram is not in SPARK.

It is up to you to mark the SPARK constructs of your program as such by specifying their SPARK *mode*. This is done using the SPARK_Mode pragma or SPARK_Mode aspect as appropriate. The SPARK mode can be explicitly set to either On or Off. If the SPARK mode of a construct is not mentioned at all, then its value is taken from some appropriate enclosing construct. For library level units,[1] if the SPARK mode is not specified explicitly, it is taken to have the special value Auto. We describe the effect of automatic SPARK mode in Section 7.1.3.

You may not change the SPARK mode on a fine grained basis such as between different subexpressions of a single expression or between individual statements of a subprogram. Roughly, the SPARK mode setting can only be changed on the granularity of packages or subprograms. Specifically, there are six locations for which we can specify a SPARK mode:

- Immediately within or before a library-level package specification
- Immediately within a library-level package body
- Immediately following the **private** keyword of a library-level package specification
- Immediately following the **begin** keyword of a library-level package body
- Immediately following a library-level subprogram specification
- Immediately within a library-level subprogram body

If you desire for your entire program to be SPARK, you can change the default by specifying

```
pragma SPARK_Mode (On);
```

as a *configuration pragma* to your compiler. Such pragmas are used by the compiler to control various features of the entire compilation. The mechanism by which configuration pragmas are applied and the scope over which they operate is compiler specific. Should you use a configuration pragma to turn on SPARK mode for the entire program, you can still turn it off for specific parts of your program as necessary. The configuration pragma only changes the default, it does not prohibit you from creating a program that is a mixture of SPARK and non-SPARK code.

There are several important use cases that are supported. In this section we will examine some of these cases. In the examples that follow, we explicitly specify the SPARK mode. Later we will describe some important consistency rules on the way SPARK mode must be used and the effect of not specifying a SPARK mode explicitly.

One of the most important cases is one in which a package specification is in SPARK and yet the corresponding package body is not in SPARK. This allows you to call subprograms in a non-SPARK package from SPARK code. As an example, consider the following abbreviated specification of the variable package Interval_Tree that encapsulates an object that stores real intervals in a structured way. A more realistic version of this package would include additional interval tree operations.

```
package Interval_Tree
  with
    SPARK_Mode     => On,
    Abstract_State => Internal_Tree,
    Initializes    => Internal_Tree
is
   type Interval is
      record
         Low  : Float;
         High : Float;
      end record;

   -- Inserts Item into the tree.
   procedure Insert (Item : in Interval)
     with
       Global  => (In_Out => Internal_Tree),
       Depends => (Internal_Tree => (Internal_Tree, Item)),
       Post    => Size = Size'Old + 1;

   function Size return Natural
     with
       Global => (Input => Internal_Tree);

   -- Destroys the tree. After this call, the tree can be reused.
   procedure Destroy
     with
       Global => (In_Out => Internal_Tree),
       Post   => Size = 0;

end Interval_Tree;
```

Interval trees are useful for efficiently determining if a given interval overlaps any of a set of existing intervals along with similar operations. Here, for purposes of illustration, an interval is represented simply as a record holding two floating point values. The specification carries the SPARK_Mode aspect set to On to indicate that it is intended to be in SPARK. The use of On is optional;

if SPARK_Mode is mentioned at all, it is assumed to take the value On unless otherwise specified. A SPARK_Mode pragma could also have been used instead. The two forms have the same meaning. The one you use is a matter of style.

The subprogram declarations in the package specification are decorated with the usual SPARK aspects such as Global, Depends, and Post. Although this example does not illustrate it, preconditions could also be provided. These aspects are used by the SPARK tools in the usual way during the analysis of code that calls the subprograms in the package.

Here is the complete body of package Interval_Tree. This body only shows the implementation of the subprograms declared in the example specification, along with the necessary supporting type declarations.

```
with Ada.Unchecked_Deallocation;
package body Interval_Tree
   with
      SPARK_Mode => Off
is
   type Tree_Node;
   type Tree_Node_Access is access Tree_Node;
   type Tree_Node is
      record
         Data        : Interval ;
         Maximum     : Float;
         Parent      : Tree_Node_Access := null;
         Left_Child  : Tree_Node_Access := null;
         Right_Child : Tree_Node_Access := null;
      end record;

   -- Intantiate a procedure for deallocating node memory
   procedure Deallocate_Node is new Ada.Unchecked_Deallocation
                                       (Object => Tree_Node,
                                        Name   => Tree_Node_Access);
   type Tree is
      record
         Root  : Tree_Node_Access := null;
         Count : Natural := 0;
      end record;

   T : Tree;  -- The actual tree in this variable package

   procedure Insert (Item : in Interval ) is
    New_Node : Tree_Node_Access; -- local to Insert, global to Subtree_Insert
    procedure Subtree_Insert (Pointer : in not null Tree_Node_Access) is
    begin
       if Item.Low <= Pointer.Data.Low then
          if Pointer.Left_Child = null then
             Pointer.Left_Child := New_Node;
             New_Node.Parent := Pointer;
          else
             Subtree_Insert (Pointer.Left_Child);
             Pointer.Maximum :=
               Float'Max (Pointer.Maximum, Pointer.Left_Child.Maximum);
          end if;
       else
          if Pointer.Right_Child = null then
             Pointer.Right_Child := New_Node;
             New_Node.Parent := Pointer;
          else
             Subtree_Insert (Pointer.Right_Child);
             Pointer.Maximum :=
               Float'Max (Pointer.Maximum, Pointer.Right_Child.Maximum);
          end if;
       end if;
    end Subtree_Insert;

 begin
    New_Node := new Tree_Node'(Data    => Item,
                               Maximum => Item.High,
                               others  => <>);
    if T.Root = null then
       T.Root := New_Node;
    else
       Subtree_Insert (T.Root);
    end if;
    T.Count := T.Count + 1;
 end Insert;

 function Size return Natural is
 begin
    return T.Count;
 end Size;

 procedure Destroy is

    procedure Deallocate_Subtree (Pointer : in out Tree_Node_Access) is
```

```
      begin
         if Pointer /= null then
            Deallocate_Subtree (Pointer. Left_Child );
            Deallocate_Subtree (Pointer. Right_Child );
            Deallocate_Node (Pointer );
         end if ;
      end Deallocate_Subtree ;

   begin
      Deallocate_Subtree (T.Root);
      T.Count := 0;
   end Destroy;

end Interval_Tree ;
```

The tree data structure in this variable package body uses pointers[2] (access types) and dynamically allocated memory to allow the tree to grow as large as necessary during program execution. As these constructs are not legal in SPARK, the package body is explicitly declared to be not in SPARK by setting the SPARK_Mode aspect to Off. Setting the SPARK mode of the body explicitly is required in this case. Because the package's specification is explicitly marked as being in SPARK, the tools will assume the body is as well unless told otherwise.

Because the package body is not in SPARK, it is not necessary to refine the abstract state declared in the specification. Furthermore, the SPARK tools will not verify that the subprograms in the body conform to their declared data and flow dependency contracts nor will the tools attempt to prove that the code is free from runtime errors and that subprograms always obey their declared postconditions. The onus is on the programmer, together with proper testing, to ensure the correctness of the code.

It is important to note, however, that pre- and postconditions are part of Ada, and thus the Ada compiler will, as usual, include runtime checks that verify the postconditions (in this example), depending on the assertion policy in force at the time the package specification is compiled. If a postcondition fails, the Assertion_Error exception[3] will be raised, which would then propagate into SPARK code. This would also be true for any unhandled Constraint_Error or Storage_Error exceptions or, for that matter, any other unhandled exceptions that might be raised in the package body.

This seems problematic because SPARK code cannot define any exception handlers to deal with exceptions from the non-SPARK code it calls. One might hope that all such cases would be caught during testing, but even if not, it is still possible to call the SPARK code from a high level "main" subprogram that is not in SPARK and that includes exception handlers for any unexpected exceptions raised by low-level non-SPARK subprograms.

It would also be possible for the subprograms in the package body to catch all exceptions they might generate (aside from the Assertion_Error exceptions raised by pre- and postcondition failures) and translate them into error status codes. However, such an approach might not be ideal if the package is to also be called by non-SPARK code where exceptions handling is natural and convenient. As we show in Section 7.1.2, this can be managed by creating a special wrapper package.

The previous example illustrated a variable package, but what if you wanted to provide interval trees as a type package? That would allow the user to create multiple, independent tree objects. This can be done, as usual, by declaring a private type to represent the trees themselves along with a private section in the package specification to detail to the compiler the nature of the tree type. However, that private section requires the use of non-SPARK constructs, access types in this example. Fortunately, it is possible to mark just the private section of the specification as not in SPARK as follows:

```
private with Ada.Finalization;
package Interval_Trees
   with SPARK_Mode => On
is
   type Interval is
      record
         Low  : Float;
         High : Float;
      end record;

   type Tree is limited private;

   -- Inserts Item into tree T.
   procedure Insert (T : in out Tree; Item : in Interval)
     with Global => null,
          Depends => (T => (T, Item));

   function Size (T : in Tree) return Natural
     with Global => null;

private
   pragma SPARK_Mode (Off);
   type Tree_Node;
   type Tree_Node_Access is access Tree_Node;
   type Tree_Node is
      record
         Data        : Interval ;
         Maximum     : Float;
         Parent      : Tree_Node_Access := null;
          Left_Child : Tree_Node_Access := null;
         Right_Child : Tree_Node_Access := null;
      end record;

   type Tree is new Ada.Finalization . Limited_Controlled with
      record
         Root  : Tree_Node_Access := null;
         Count : Natural := 0;
      end record;

   overriding procedure Finalize (T : in out Tree);

end Interval_Trees ;
```

In this case it is necessary to use the pragma form of SPARK_Mode as Ada does not allow aspects to be applied to just the private section of a package specification. The pragma SPARK_Mode (Off) must appear at the top of the private section marking the entire private section as not in SPARK.

This example also makes use of Ada's facility for automatic finalization of objects[4] so there is no need for the programmer to call a "destroy" procedure in this case. To do this, it is necessary to with Ada. Finalization . However, because that package is only needed to support the private section, it can be introduced using the special form of the **with** clause we introduced in Section 3.5.2. The **private with** makes the package's resources only available in the private part.

It is also possible to mark individual subprograms as being in SPARK. This is useful in cases where a package specification needs to make use of non-SPARK constructs, yet some of the subprograms in the package can still be given SPARK declarations. SPARK code can then call the subprograms with SPARK declarations even though it might not be able to use all the facilities of the enclosing package.

Similarly, the bodies of subprograms in a package body can be marked as in or out of SPARK as appropriate. This allows you to use the SPARK tools on code where it makes sense to do so without creating an unnatural design by artificially avoiding full Ada features for other subprograms in the same package.

There is an important consistency rule regarding SPARK mode that says once SPARK mode is turned off, you cannot turn it back on again for any subordinate construct. To illustrate, packages are considered to have four parts:

1. The visible part of the specification
2. The private part of the specification
3. The body
4. Elaboration code in the body that appears after **begin**

The consistency rule means that if SPARK mode is explicitly turned off for one of the parts, it cannot be turned on again in a later part. For example, if the body of a package has its SPARK mode off, you cannot then turn SPARK mode back on in the elaboration code for that package.

Furthermore if SPARK mode is turned on for a part, it is assumed to be on for all following parts unless it is explicitly turned off. Thus, setting SPARK_Mode to On in the specification of a package declares all parts of the package to be in SPARK unless SPARK_Mode is explicitly set to Off for a later part.

7.1.2 *Wrapper Packages*

As previously described, it is permitted for SPARK to call code in a library package with a non-SPARK body provided the specification of that package, or at least of the subprogram being called, is in SPARK. Even if the declarations of the called subprograms are not specifically marked as being in SPARK, it may still be permitted for SPARK code to call them using automatic SPARK mode as described in Section 7.1.3.

However, errors in the non-SPARK library package are likely to be reported by way of exceptions as that is the normal method of error handling in full Ada. Thus, some means of translating exceptions into the status codes required by SPARK-style error handling is needed. Furthermore if the non-SPARK library package has a non-SPARK specification, it may still be possible with suitable translations and conversions to make it callable from SPARK. Doing these things requires constructing a wrapper package that provides a SPARK specification and contains subprograms that just forward their calls to the underlying library package.

As an example, consider the Interval_Tree package described in Section 7.1.1. That package happens to have a SPARK specification, yet exceptions arising in the body of the package may still propagate into SPARK code. To deal with this, we create a new package Interval_Tree_Wrapper with a specification very similar to that of Interval_Tree shown as follows:

```
with Interval_Tree ;
package Interval_Tree_Wrapper
   with
      SPARK_Mode    => On,
      Abstract_State => Underlying_Tree,
      Initializes   => Underlying_Tree
is

   type Interval is new Interval_Tree . Interval ;

   type Status_Type is (Success,    Insufficient_Space ,   Logical_Error );

   -- Inserts Item into the tree .
   procedure Insert (Item     : in   Interval ;
                     Status : out Status_Type)
     with
       Global  => (In_Out => Underlying_Tree),
       Depends => (Underlying_Tree => (Underlying_Tree, Item),
                   Status => Underlying_Tree),
       Post    => (Size'Old = (if Status = Success then Size - 1 else Size ));

   function Size return Natural
     with
       Global => (Input => Underlying_Tree);

   -- Destroys the tree . After this call the tree can be reused .
   procedure Destroy
     with
       Global => (In_Out => Underlying_Tree),
       Post   => Size = 0;

end Interval_Tree_Wrapper ;
```

The main difference between this specification and that for Interval_Tree is that a status type is introduced and subprograms that might fail – Insert in this case – are modified to return a status indication.

The wrapper package also introduces its own types for the types provided by the underlying package (type Interval in this case). This allows the wrapper package to be self-contained without its clients needing to **with** the underlying package.

Notice also that the SPARK aspects for Insert are slightly different than for the underlying version of the procedure. The flow dependency contract adds a dependency of Status on Underlying_Tree. Furthermore, the postcondition is changed to assert that the size of the tree increases if and only if the subprogram returns successfully.

It would be more natural to write the postcondition as

```
Post => (if Status = Success
            then Size = Size'Old + 1
            else Size = Size'Old);
```

However, Ada disallows use of the 'Old attribute with a prefix of a function call in a context where it might not be evaluated, such as in a branch of a conditional expression. Recall that 'Old implies the value of the prefix expression is saved when the subprogram is entered. It is undesirable to call a function when the result might not be needed. Here, Size'Old is used in both branches of the conditional expression. Nevertheless the rules of Ada forbid it in this case.

The Underlying_Tree state abstraction is intended to represent the internal tree of the underlying package. Strictly speaking, Interval_Tree_Wrapper has no internal state so declaring any Abstract_State for it is a "lie." The SPARK tools will not notice the lie because the body of Interval_Tree_Wrapper is, as you will see, not in SPARK and thus not examined by the tools. However, if the only access to the underlying package is through the wrapper, the lie is not really a problem because the wrapper package, in effect, assumes the internal state of the wrapped package, just as wrapping paper can be said to contain the same present as the box it wraps.

In this case, however, Interval_Tree has a SPARK specification and could potentially be called directly by SPARK code willing to pass exceptions through to a higher level. For the SPARK tools to properly understand the relationship between that other code and the wrapper package, it is necessary to tell the truth about what the wrapper package is doing. This can be done by defining no Abstract_State on Interval_Tree_Wrapper and instead explicitly referencing Interval_Tree . Internal_Tree in the SPARK aspects where Underlying_Tree is mentioned previously.

The body of the wrapper package contains relatively simple subprograms that wrap the subprograms of the underlying library package. Here is the implementation of Insert in the wrapper package:

```
procedure Insert (Item   : in  Interval ;
                  Status : out Status_Type) is
begin
   Interval_Tree . Insert ( Interval_Tree . Interval (Item));
```

```
      Status := Success;
   exception
      when Storage_Error =>
         Status := Insufficient_Space ;
      when others =>
         Status := Logical_Error ;
   end Insert ;
```

Exceptions are converted to status codes in this case, but no other work is done aside from a trivial type conversion from the wrapper's type Interval to the underlying package's type Interval . In general, the wrapper subprograms could transform the parameters in arbitrary ways before calling the underlying subprogram or transform the results of the underlying subprogram before returning.

Creating a wrapper package can be tedious and must be done carefully to prevent errors arising that the SPARK tools cannot detect. However, it has the advantage of not requiring any access to the source code of the underlying library package body.

7.1.3 *Automatic* SPARK *Mode*

So far we have discussed SPARK_Mode as a binary valued aspect that can either be On or Off. However, there is also an automatic setting that allows the SPARK tools to determine the SPARK mode of a construct on their own. This Auto setting cannot be explicitly specified; it is only implied under certain circumstances that we outline here.

An important use case of automatic SPARK mode is when you attempt to make use of existing library packages in a SPARK program. The Ada standard library is a particularly noteworthy case, but any library written without SPARK in mind is at issue. Many entities declared by such a library may be perfectly reasonable SPARK. If the tools required an explicit SPARK mode on the specification of the library, you would have to wait until the library vendor provided a SPARK-aware update to the library before you could use it in your SPARK program. That might never happen.

To see an example of automatic SPARK mode, consider the following function Contains. This function accepts a string and a character and returns true if and only if the given string contains the given character.

```
with Ada.Strings.Fixed;
function Contains (S  : in String;
                   Ch : in Character) return Boolean
   with SPARK_Mode => On
```

```
is
   Search_String  :  String (1 .. 1) := (others => Ch);
   Result_Index   :  Natural;
begin
   Result_Index := Ada.Strings.Fixed.Index (Source  => S,
                                            Pattern => Search_String,
                                            From    => S'First);
   return Result_Index /= 0;
end Contains;
```

This function is in its own compilation unit and is thus a library level function that is not nested inside any package. Its SPARK_Mode is explicitly set to On. However, it makes use of a function Index from the Ada standard library package Ada.Strings.Fixed. This function searches a string for a specified substring and returns the index where the substring appears or zero if it does not appear.

However, the library level specification of Ada.Strings.Fixed does not contain an explicit SPARK_Mode setting and is thus processed in automatic SPARK mode. The tools determine that the declaration of function Index is in SPARK and thus allow that function to be called from SPARK code. The tools synthesize data and flow dependency contracts for the function as described in Section 4.5. However, the body of Ada.Strings.Fixed is not analyzed and, thus, the tools are not able to synthesize the Global aspect of function Index and produce a warning to this effect. By default, this warning prevents the tools from attempting to prove the body of Contains.

The documentation for function Index makes it clear that it does not read nor write any global data. The assumption made by the SPARK tools is thus true for it. The tools can be run in such a way as to continue even if warnings are issued, and that is appropriate to do in this case. However, one verification condition is not proved. In particular, passing S'First to Index may raise Constraint_Error because in the case when S is empty, the bounds may be outside the range of the Positive subtype.

One way to handle this is to treat the empty string as a special case. Alternatively, we decide to prohibit calling Contains on empty strings by adding a precondition:

```
with Ada.Strings.Fixed;
function Contains2 (S  : in String;
                    Ch : in Character) return Boolean
   with
      SPARK_Mode => On,
      Pre => S'Length > 0
```

```
is
   Search_String : String (1 .. 1) := (others => Ch);
   Result_Index  : Natural;
begin
   Result_Index := Ada.Strings.Fixed.Index (Source  => S,
                                            Pattern => Search_String,
                                            From    => S'First);
   return Result_Index /= 0;
end Contains2;
```

In this version all verification conditions are proved.

There is one additional detail to consider when using library packages in this manner. The Index function may raise various exceptions depending on the values of its arguments. Thus, as when calling any code that is not in SPARK, we need to be mindful of the possibility that exceptions may be raised despite the fact that no runtime errors will arise from the body of Contains2 itself (provided its precondition is honored). We can ensure no exceptions are raised by encoding the preconditions of Index directly into the SPARK code as an assertion. Furthermore, the postcondition of Index can be expressed as an assumption.

```
with Ada.Strings.Fixed;
function Contains3 (S  : in String;
                    Ch : in Character) return Boolean
   with
      SPARK_Mode => On,
      Pre        => S'Length > 0,
      Post       => Contains3'Result = (for some I in S'Range => S (I) = Ch)
is
   Search_String : String (1 .. 1) := (others => Ch);
   Result_Index  : Natural;
begin
  -- An empty search string causes Index to raise Pattern_Error
  -- A starting point for the search that is out of bounds raises Index_Error
   pragma Assert (Search_String'Length > 0 and S'First in S'Range);
   Result_Index := Ada.Strings.Fixed.Index (Source  => S,
                                            Pattern => Search_String,
                                            From    => S'First);
   -- Index returns zero if it does not find the string or else it returns
   -- a position in the string where the pattern starts
   pragma Assume
     (if Result_Index = 0 then
        (for all J in S'Range => S (J) /= Ch)
      else
         (Result_Index in S'Range and then S (Result_Index) = Ch));
   return Result_Index /= 0;
end Contains3;
```

This example illustrates how pre- and postconditions for library subprograms can be provided, in effect, without modifying the source code of the library packages. Of course a wrapper package around Index could also be used to provide pre- and postconditions in a more natural way.

We note that this example is unrealistic in the sense that if you really wanted to write a function like Contains3, it could be trivially done using Ada's quantified expressions directly:

```
function Contains4 (S  : in String;
                    Ch : in Character) return Boolean is
  (for some J in S'Range => S(J) = Ch);
```

However, the example illustrates the general approach to using automatic SPARK mode to call library subprograms that were not written with SPARK in mind.

7.2 SPARK and C

SPARK has an important role to play in safety-critical embedded systems where failure of the software can cause major loss of investment or serious injury. However, most embedded systems are written today in the C programming language – a language notorious for being difficult to use correctly. Yet despite this, a large amount of carefully built and very well tested C code exists that SPARK developers might want to reuse.

To make use of an existing C library from SPARK, you first need an Ada compiler that has an "associated" C compiler. In the case of the GNAT Ada compiler, one such C compiler is gcc, but there can potentially be many C compilers that would be compatible with a given Ada compiler. In general, it is necessary for the Ada compiler to be aware of the C compiler so that data layout and calling conventions can be properly matched whenever information crosses from Ada to C and vice versa. Also, the Ada compiler must match the facilities in package Interfaces.C, described shortly, with the facilities provided by the associated C compiler.

Although many of the details that arise when interfacing SPARK and C are outside the scope of this book, we now show an example that illustrates several important points. The Ada standard mandates certain features for any implementation that wishes to provide a C interface, and we endeavor to use just those features. Individual Ada implementations can provide additional features that are not required by the standard and, hence, are less portable. We make use of one such additional feature that we will highlight later.

Our example is contrived but is realistic enough to be illustrative. Suppose there is an existing C library that manages message packets in some kind of communication system. We focus first on the C header file that declares certain types and two example functions:

```
#ifndef MESSAGE_H
#define MESSAGE_H

   #include <stddef.h>

   typedef unsigned short nodeid_t;       // 16 bit network node ID numbers.
   typedef unsigned long  sequenceno_t;   // 32 bit packet sequence numbers.

   struct PacketHeader {
       nodeid_t      source_node;
       nodeid_t      destination_node;
       sequenceno_t  sequence_number;
   };

   enum error_code {
       SUCCESS = 1, INVALID_DESTINATION, INSUFFICIENT_SPACE
   };

   // Returns the Fletcher checksum of the given buffer.
   unsigned short compute_fletcher_checksum(const char *buffer, size_t size);

   // Installs the given header into the buffer. Returns an error code as
   // appropriate. If an error is detected the buffer is left unchanged.
   enum error_code install_header(
       char *buffer,
       size_t size,
       const struct PacketHeader *header);

#endif
```

In this hypothetical system, each node in the "network" is represented by a 16-bit node identifier. Message packets have headers consisting of the source and destination node addresses and a unique sequence number. A function for computing the Fletcher checksum over a data array is given. Here, the array is treated in the usual C style as a pointer to the first element and a separately provided size. A second function is declared that copies a header structure into a packet buffer, providing some checks to ensure the sanity of the operation.

This example illustrates the use of type aliases (C's `typedef` declaration), structures, arrays, enumeration types, and functions that take these parameters in a natural C style. The implementation of the example functions is straightforward but not shown here as it is not important for our purposes. In some cases the implementation of the library functions may not be available anyway. However, any C programmer who wishes to use the library will have access to the header file(s) that describe it.

The first step is to write a SPARK package specification that declares the necessary types and subprograms. In effect, we translate the C header file to Ada following certain rules described in the Ada standard and extended by your specific Ada compiler. We will show the specification in stages starting with the type declarations:

```
with Interfaces.C;
package Messages
   with SPARK_Mode => On
is
   type Node_Id_Type is new Interfaces.C.unsigned_short;
   type Sequence_Number_Type is new Interfaces.C.unsigned_long;

   type Error_Code is (Success, Invalid_Destination, Insufficient_Space)
      with Convention => C;
   for Error_Code use (1, 2, 3);

   type Packet_Header_Type is
      record
         Source_Node      : Node_Id_Type;
         Destination_Node : Node_Id_Type;
         Sequence_Number  : Sequence_Number_Type;
      end record;
      with Convention => C;

end Messages;
```

The Ada standard does not require every Ada compiler to support an interface to C. However, if a compiler does support such an interface, it must also provide package `Interfaces.C`, which contains, among other things, definitions of types that match the built-in types used by the associated C compiler. For example, `Interfaces.C.int` has the same size and range of values as the associated C compiler's type `int`. This approach is necessary because there is no assurance, for example, that the Ada type `Integer` is compatible with the C type `int`. It happens that with the GNAT and `gcc` compilers the two types are compatible, but for maximum portability it is better to use the types in Interfaces .C.

The previous package specification, written with SPARK_Mode on, starts by introducing types for the node identifiers and sequence numbers. The names do not have to be the same as in the C header file; they only need to be used appropriately in the following declarations. In the original C, `nodeid_t` and `sequenceno_t` are aliases for C built-in types and can be mixed freely with other unrelated variables having those same types. However, the definitions in package Messages are for entirely new types that cannot be accidentally mixed. This is more robust and is an example of Ada's ability to increase the type safety of a preexisting C interface.

The enumeration is defined as an Ada enumeration with the Convention aspect set to C. This informs the Ada compiler that objects of that type should be represented compatibly with the associated C compiler's handling of enumerations. The ability to set the Convention aspect on an enumeration type is the only GNAT extension we use in this example. Because the C header file defines the enumeration starting at one instead of the default of zero, package Messages uses an *enumeration representation clause* to specify the values of the enumerators in a matching way.[5]

Finally, the C structure is modeled as an Ada record, again using the Convention aspect of C to ensure that the Ada compiler lays out the record in a manner that matches the associated C compiler's expectations for structures.

Armed with these type definitions we can now write declarations for the two C functions in our example. We will start with the checksum computing function because, being a pure function, it is easy to represent in SPARK:

```
function Compute_Fletcher_Checksum
           (Buffer : in  Interfaces .C. char_array ;
            Size   : in  Interfaces .C. size_t ) return  Interfaces .C. unsigned_short
   with
      Global        => null,
      Import        => True,
      Convention    => C,
      External_Name => "compute_fletcher_checksum";
```

The C version of this function takes an array of characters and does not modify that array as evidenced by the `const` in the declaration of the `buffer` parameter in the C header file. Package Interfaces .C contains a declaration of an unconstrained array type holding C-style characters. Thus, the first parameter of the function is of this type and has mode **in**. The Ada compiler will pass the actual parameter by its address as expected by the C function.

The aspects associated with the declaration include the Import aspect set to True, indicating that the function is actually written in a foreign language. The Convention aspect specifies which language is used. It is not necessary, or even legal, to write a body (in Ada) for an imported subprogram.

The External_Name aspect specifies the name of the function as created by the C programmer. The language C is case sensitive so this allows us to give the function an Ada-friendly name using mixed case while still connecting the declaration to the correct underlying C function.

In addition, the Global aspect indicates that the function does not modify any global data. As with packages that wrap non-SPARK Ada code, the SPARK tools cannot verify the truth of this assertion because they cannot analyze the C body of the function. Again, the onus is on the programmer to get it right. Presumably, the programmer read the documentation for the C library before writing the Global aspect. In effect, the programmer is transferring information from the C library documentation into a form that can be understood by the SPARK tools.

Before considering the second function in the C header file, it is worthwhile to step back and reflect on the package specification we have created so far. Although the specification is in SPARK, it makes visible use of various entities in package Interfaces.C. This informs everyone that the body of the package is written in C and thus exposes what should ideally be a hidden implementation detail. Furthermore, the clients of this package are forced to deal with the types in Interfaces.C to use the subprograms provided by the package even though clients should know nothing about C. In effect, the "C-isms" from the implementation of this package are leaking out into the rest of the program.

It is often useful to distinguish between thin and thick bindings to an existing library. A *thin binding* is a nearly literal translation of the existing library interface with little or no effort made to change the architecture or design of that interface. In contrast a *thick binding* is a reworking of the existing library interface to take good advantage of features in the client environment.

Our work so far uses Ada features to distinguish the types Node_Id_Type and Sequence_Number_Type from each other, but it is otherwise a thin binding. We can thicken the binding both to provide a more natural interface to SPARK clients and to completely hide all use of Interfaces.C. To do this we must create a wrapper package. The following specification shows one possibility:

```
pragma SPARK_Mode(On);
package Messages_Wrapper is
   type Checksum_Type is mod 2**16;
   function Compute_Checksum (Data : in String) return Checksum_Type;
end Messages_Wrapper;
```

No mention of Interfaces .C appears in this specification. Furthermore, because Ada arrays know their size, there is no reason for this function to take an additional size parameter as is common for C functions. Here is the body of this package:

```
pragma SPARK_Mode(On);
with Interfaces .C;
package body Messages_Wrapper is
   use type Interfaces .C. size_t ;   -- needed for visibility in precondition

   function Compute_Fletcher_Checksum
            (Buffer : in Interfaces .C. char_array ;
             Size   : in Interfaces .C. size_t ) return Interfaces .C. unsigned_short
      with
         Global        => null,
         Import        => True,
         Convention    => C,
         Pre           => Size = Buffer'Length,
         External_Name => "compute_fletcher_checksum";

   function Compute_Checksum (Data : in String) return Checksum_Type is
      -- Copy the Ada string Data into the C string Buffer
      Buffer : Interfaces .C. char_array :=
                  Interfaces .C.To_C (Item => Data,
                                      Append_Nul => False);
      Result : Interfaces .C. unsigned_short ;
   begin
      -- Call the C function whose Ada specification is above
      Result := Compute_Fletcher_Checksum (Buffer => Buffer,
                                           Size   => Buffer'Length);
      -- Return the Result converted to Checksum_Type;
      return Checksum_Type (Result);
   end Compute_Checksum;
end Messages_Wrapper;
```

The declaration of the imported C function now appears in the body instead of in a specification of its own. Appropriate type conversions are done to match the natural Ada types used by the wrapper package to the C-like types provided by Interfaces.C. The body of the preceding package is in SPARK, and the SPARK tools successfully prove that it is free of runtime errors. Thus, the type conversion from, for example, Interfaces.C.unsigned to Checksum_Type will always succeed without raising Constraint_Error. Furthermore, the precondition specified on the imported declaration will always be satisfied.

Writing a wrapper package is more work than just writing a specification with imported declarations, but it has the advantage of completely hiding the use of C inside the body of the wrapper package. The GNAT compiler has a command line option, `-fdump-ada-spec`, that automatically converts a C header file into an approximately correct Ada package specification. The binding created by this option is very thin and may require some adjustments before it even compiles. However, this feature provides a quick way to get started writing a thicker binding such as we just described.

Writing a SPARK declaration for the second C function in our example (`install_header`) is tricky. Because this C function modifies its parameter `buffer`, we cannot provide a direct SPARK function declaration. Recall that SPARK functions may not modify their parameters. Instead, we must write a SPARK procedure declaration that passes the buffer as an **in out** parameter and returns the status of the operation as an **out** parameter. This solution requires us to write a helper subprogram that calls the underlying C function `install_header` and returns the status through a parameter rather than as a function return value. The helper subprogram could be written in C or, alternatively, in full Ada. Here is the C version of the helper subprogram:

```
void install_header_helper (
   char *buffer,
   size_t size,
   const struct PacketHeader *header,
   enum error_code *status)
{
   // Call function install_header and save return value in local result
   enum error_code result = install_header(buffer, size, header);
   // Copy result to "out" parameter
   *status = result;
}
```

This function must be compiled with the associated C compiler and linked into the final program. However, it does not require any access to the source code of the original library function it wraps.

We can now write an Ada declaration as a procedure using the helper function:

```
-- Needed to make operators used in postcondition directly visible.
use type Interfaces.C.size_t;
use type Interfaces.C.char;
use type Interfaces.C.char_array;
procedure Install_Header (Buffer : in out Interfaces.C.char_array;
                          Size   : in     Interfaces.C.size_t;
                          Header : in     Packet_Header_Type;
                          Status :    out Error_Code)
   with
      Global  => null,
      Depends => (Buffer =>+ (Size, Header), Status => (Size, Header)),
      Post    => (if Status /= Success then
                     Buffer = Buffer'Old
                  else
                    (for all J in Buffer'First + 12 .. Buffer'Last =>
                             Buffer(J) = Buffer'Old (J))),
      Import        => True,
      Convention    => C,
      External_Name => "install_header_helper";
```

The buffer parameter is declared with mode **in out** because the logic of the C function is such that it returns the buffer with some of its elements unchanged. Thus, the buffer should be fully initialized before calling the procedure (the SPARK tools will ensure this is true). Notice also that the header record is passed as an ordinary **in** parameter. The Ada compiler will pass it to the underlying C function using a pointer as the C function expects. In this case the Ada procedure is given a natural name, but the External_Name aspect points the declaration to the helper function rather than the original.

The declaration includes several SPARK aspects to specify the data dependency and flow dependency contracts. A postcondition is also provided. As before, these contracts cannot be checked by the SPARK tools. In fact, because postconditions are ordinarily compiled into the body of the subprogram to which they apply, you might expect that even the postcondition would not be checked at runtime because the body of the procedure in this case is actually a C function. However, the GNAT Ada compiler will generate a stub for the declaration that surrounds the actual call to the C function, and that includes postcondition runtime checks. Thus, Assertion_Error might be raised at runtime, as usual, depending on the assertion policy in force.

Preconditions are handled similarly. Although the SPARK tools will endeavor to prove that any precondition is satisfied at each call site, the GNAT-generated stub also includes runtime checking for preconditions as described previously. The compiler's ability to create these stubs enhances the assurances of correctness obtained when testing mixed Ada/C programs.

In any case, assuming the underlying C function is correct, perhaps verified via testing, the SPARK tools will use the contracts on the declaration to help prove properties of the code that calls the procedure. This is another example of bringing information that might be in the C library documentation forward into the code itself.

As we mentioned previously, the helper subprogram could also be written in full Ada. The idea would be to write an imported declaration for the underlying C function that has an **in out** parameter, as allowed in full Ada 2012. A helper procedure could call the imported C function and perform essentially the same steps as the C helper above. In particular, it could write the value returned by the underlying function into an **out** parameter.

The helper procedure could be placed in its own package, or in a package serving as a thick binding to the C library as previously described, with pre- and postconditions applied to the helper procedure instead of to the imported declaration. However the body of the helper procedure cannot be in SPARK as it must use a non-SPARK declaration. We leave the details of this implementation as an exercise for the reader.

7.3 External Subsystems

In the previous sections we saw that a system need not be entirely written in SPARK. By providing SPARK interfaces to non-SPARK code, we can still make use of the analysis tools provided by SPARK. In this section we look at a higher level approach to interacting with hardware devices and other software subsystems that are external to our SPARK program. These *external subsystems* are by definition outside the control of our program and thus have behaviors the program can not fully anticipate. When SPARK is being used, it is especially important to properly model this situation so the SPARK tools can account for it. External variables and external state abstractions provide the necessary models.

7.3.1 *External Variables*

Memory mapped variables provide one method that programs use to interact with hardware.[6] The basic idea is that particular memory addresses resolve to hardware registers rather than to random access memory. Thus, a program can access hardware registers through ordinary variables. In Ada, memory mapped variables are defined by the Volatile and Address aspects. With the addition of SPARK aspects describing the external properties of memory mapped variables, we can use the SPARK tools to analyze the program's use of these variables.

If the external subsystem reads the value of a memory mapped variable at a time of its own choosing, that variable is said to have an *asynchronous reader*. Similarly, if the external subsystem updates a variable at a time of its own choosing, that variable is said to have an *asynchronous writer*. Notice that the terms asynchronous reader and asynchronous writer are from the point of view of the external subsystem. It is the external subsystem that is reading and writing.

SPARK provides two Boolean aspects to specify either or both of these possibilities:

Async_Readers: Any object for which Async_Readers is true may be read at any time (asynchronously) by hardware or software outside the program.

Async_Writers: Any object for which Async_Writers is true may be changed at any time (asynchronously) by hardware or software outside the program.

Async_Readers has no effect on either flow analysis or proof analysis and thus serves mostly a documentation purpose. Async_Writers has no effect on flow analysis but does have an effect on proof analysis. The proof tool takes into account that two successive reads of the same variable may return different results.

SPARK provides two related Boolean aspects that do control the flow analysis of external objects:

Effective_Reads: Indicates that the program's reading the value of a volatile variable has an effect on the external hardware or software subsystem. Effective_Reads can only be specified on a variable that also has Async_Writers set.

Effective_Writes: Indicates that the program's assigning a value to the variable has an effect on the external hardware or software subsystem. Effective_Writes can only be specified on a variable that also has Async_Readers set.

Both Effective_Reads and Effective_Writes have an effect on flow dependencies. Reading the former or writing the latter is modeled as having an effect on the value of the variable.

We typically set Effective_Reads to true for devices that provide a stream of input values such as mass storage devices and serial ports and to false for reading from devices such a sensors for which there is no significant relation between successive values. We typically set Effective_Writes to true.

Let us look at a simple example of how Effective_Reads change flow analysis.[7] In the following code fragment, Volatile_Value is a volatile variable for which Async_Writers is true. The hardware or software outside the program can change this volatile variable at any time.

```
if Count = 0 then
   My_Value := Volatile_Value;
end if;
My_Value := Volatile_Value;
```

Does the value of My_Value depend on the value of Count? If Effective_Reads is true, then My_Value will depend on Count. For example, when reading characters from a buffer, Count determines whether My_Value ends up with the first or second character. With Effective_Reads set to false, My_Value will not depend on Count. For example, when reading from a temperature sensor, My_Value will contain the most recent temperature.

The Ada language provides the concept of Volatile objects. Such objects behave as if all four of the SPARK aspects are true. SPARK allows you to refine the behavior of the program by specifying some subset of those aspects in cases where it makes sense to do so.

Let us look at some examples. Here is a definition package that defines a modular type and a single 8-bit volatile variable mapped to memory address $FFFF0000_{16}$:

```
with System.Storage_Elements;
package Numeric_Display
  with SPARK_Mode => On
is
   type Octet is mod 2**8;

   Value : Octet
      with
         Size     => 8,  -- Use exactly 8 bits for this variable
         Volatile => True,
         Address  => System.Storage_Elements.To_Address(16#FFFF0000#);

end Numeric_Display;
```

The variable Value is a single 8-bit memory mapped register at a specific memory address. This register controls a standard seven segment LED display. The register is given the Ada Volatile aspect, which means it has both asynchronous readers and writers. Furthermore, reads and writes are always effective. Consider the following program fragment that makes use of this memory mapped register:

```
Numeric_Display.Value := 0;
Numeric_Display.Value := 1;
```

Normally, flow analysis would warn us that the first assignment statement is unused. However, because the volatile variable Value has an asynchronous reader and writes to it are effective, this code does not generate a flow error. Each update is processed by the external system (a hardware device in this case).

Suppose now that the program wishes to read back the value in the control register. Variable X is type Numeric_Display.Octet:

```
X := Numeric_Display.Value;
X := Numeric_Display.Value;
```

This code also does not generate a flow error because the register is assumed to potentially change values between each read and each read is effective (could change the state of the hardware).

However, this characterization of our device is overly aggressive. Because the device does not change the value in the control register at all, reading it twice in succession will always produce the same value. Nor does reading the register controlling an LED display change that display. Consider instead the following declaration of the memory mapped register:

```
Value : Octet
   with
      Size => 8, -- Use exactly 8 bits for this variable
       Volatile          => True,
      Async_Readers      => True,
       Effective_Writes  => True,
      Address => System.Storage_Elements.To_Address(16#FFFF0000#);
```

Because two of the SPARK aspects are explicitly provided, the other two SPARK aspects default to false. The declaration is as if we wrote

```
Value : Octet
   with
      Size => 8, -- Use exactly 8 bits for this variable .
       Volatile          => True,
      Async_Readers      => True,
       Effective_Writes  => True,
      Async_Writers      => False,
       Effective_Reads   => False,
      Address => System.Storage_Elements.To_Address(16#FFFF0000#);
```

The meaning of this can be summarized as follows:

- *Async_Readers* => *True*. There is external hardware that may read Value at any time.
- *Effective_Writes* => *True*. Values assigned to Value have an effect on the external hardware.
- *Async_Writers* => *False*. There is no external hardware that writes to Value.
- *Effective_Reads* => *False*. It is a flow problem to read Value multiple times without the program doing any intervening writes.

This refined description of how the external register works is more robust than our original because it catches errors that using just Volatile would not. For example, flow analysis of this double assignment now tells us that the first assignment is unused:

```
   X := Numeric_Display.Value;
   X := Numeric_Display.Value;
```

7.3.2 *External State Abstractions*

Information hiding is an important principle of design. By hiding our design decisions, we can more easily change those decisions. For example, in the previous section we used a memory mapped variable to allow our program to display values on a seven segment LED display. Should we need to move this application to a processor that used port-based I/O rather than memory mapped I/O, we would probably need to change many parts of that program. By hiding the details of the hardware connection, we would only have to change the module containing those details.

SPARK provides *external state abstractions* to hide the details of the external interface. Here is a package specification that describes the properties of our external LED display while hiding the memory mapped external variable in the package body. We also provide a procedure that translates the ten digits into individual display segments.

```
package LED_Display
   with Spark_Mode      => On,
        Abstract_State  => (LED_State
                               with External => (Async_Readers    => True,
                                                 Effective_Writes  => True))
is
   subtype Digit_Type is Integer range 0 .. 9;
   procedure Display_Digit (Digit : in Digit_Type)
     with
        Global => (Output => LED_State);

end LED_Display;
```

The Abstract_State aspect defining the abstract state LED_State has more options than those you saw in Section 4.3.[8] Procedure Display_Digit references this abstract state in its Global aspect.

The declaration of LED_State includes the option External, which tells us that the actual state is maintained in hardware devices and/or other software subsystems that are external to our SPARK program. Finally, the two external properties Async_Readers and Effective_Writes are given for this external state object. These are the same properties we used with volatile variables in the previous section. Like a volatile variable, if none of the four properties is specified, all four properties are assumed to be true. And, as is the case here, if one or more properties are defined, the undefined properties are assumed to be false.

Here is the body of package LED_Display where we refine the abstract state LED_State, define the external variable that connects us to the hardware, and implement the procedure Display_Digit:

```
with System.Storage_Elements;
package body Led_Display
   with SPARK_Mode => On,
        Refined_State => (LED_State => Value)
is
   type Octet is mod 2 ** 8;

   Value : Octet
      with
        Size => 8, -- Use exactly 8 bits for this variable
        Volatile         => True,
        Async_Readers    => True,
        Effective_Writes => True,
        Address => System.Storage_Elements.To_Address(16#FFFF0000#);

   -- Segments 'a' through 'g' are in order from least to most significant bit.
   -- Active high.
   Patterns : constant array (Digit_Type) of Octet :=
                (2#0011_1111#, 2#0000_0110#, 2#0101_1011#, 2#0100_1111#,
                 2#0110_0110#, 2#0110_1101#, 2#0111_1101#, 2#0000_0111#,
                 2#0111_1111#, 2#0110_0111#);
   procedure Display_Digit (Digit : in Digit_Type)
      with Refined_Global => (Output => Value)
   is
   begin
      Value := Patterns (Digit);
   end Display_Digit;

end LED_Display;
```

We refined our abstract state LED_State to the volatile variable Value. We can refine an abstract state into several different concrete or abstract states. However, SPARK requires that all the external properties specified for our abstract state are realized in the refined state. These realizations of properties may be done by a single refined object as we did here or by a combination of objects. Should an external abstract state have no properties given, you must refine it into one or more objects that together realize all four external properties.

Let us look at a more complex example that uses an external state abstraction. The following package provides an interface to a single serial port. It provides procedures for opening and closing the port and procedures for reading and writing single bytes.

```
with Interfaces;
package Serial_Port
   with
      SPARK_Mode     => On,
      Abstract_State => (Port_State, (Data_State with External)),
      Initializes    => Port_State
is
   -- Types for configuring serial parameters.
   type Baud_Type      is (B2400, B4800, B9600, B19200);
   type Parity_Type    is (None, Even, Odd);
   type Data_Size_Type is (Seven, Eight);
   type Stop_Type      is (One, Two);

   -- Type for serial port data
   type Byte is new Interfaces.Unsigned_8;

   -- Type used to convey error codes.
   type Status_Type is (Success, Open_Failure, IO_Failure);

   -- Returns Open_Failure if port is already open or if the open fails.
   procedure Open (Baud   : in Baud_Type;
                   Parity : in Parity_Type;
```

```
                          Data_Size  : in   Data_Size_Type;
                          Stop       : in   Stop_Type;
                          Status     : out  Status_Type)
      with
         Global   => (In_Out => Port_State),
         Depends => ((Port_State, Status) =>
                        (Port_State, Baud, Parity, Data_Size, Stop));

   -- Returns IO_Failure if port is not open or if the underlying I/O fails.
   procedure Read (Item    : out Byte;
                   Status  : out Status_Type)
      with
         Global   => (Input => (Port_State, Data_State)),
         Depends => (Item => (Port_State, Data_State), Status => Port_State);

   -- Returns IO_Failure if port is not open or if the underlying I/O fails.
   procedure Write (Item    : in   Byte;
                    Status  : out Status_Type)
      with
         Global   => (Input => Port_State, Output => Data_State),
         Depends => (Data_State => (Port_State, Item), Status => Port_State);

   -- Has no effect if port is not open. Returns no failure indication.
   procedure Close
      with
         Global   => (In_Out => Port_State),
         Depends => (Port_State => Port_State);
end Serial_Port;
```

The specification of the package is in SPARK and declares two state abstractions. The first, Port_State, models the serial port hardware itself. It tracks if the port is open, the serial parameters that are in use and any other related information such as permissions or system level errors. This state is initialized in some way by the underlying system so the package declares that Port_State is automatically initialized.

The other state abstraction, Data_State, represents the external subsystem to which the serial port is connected and is thus declared to be an external state abstraction. Because none of the four external properties are specified, they all default to true. The external subsystem is assumed to do I/O asynchronously with the serial port and that, furthermore, all reads and writes to the port are effective. In particular, two successive reads from the port may return different values, and writing the same value to the port twice in succession is certainly useful.

The subprograms in the package are decorated with data and flow dependency contracts as usual, written in terms of the two state abstractions. When writing these contracts, it is important to keep clearly in mind what the state abstractions represent. For example, consider the contracts on the Read procedure:

```
-- Returns IO_Failure if port is not open or if the underlying I/O fails .
procedure Read (Item   : out Byte;
                Status : out Status_Type)
   with
      Global  => (Input => (Port_State, Data_State)),
      Depends => (Item => (Port_State, Data_State), Status => Port_State);
```

The Read procedure reports an error if the port is not open, and that information is part of Port_State. Furthermore, Read checks error information reported by the underlying runtime system, which is also contained in Port_State. Thus, the procedure inputs from Port_State and, furthermore, Port_State is used to derive both the value of Item and the resulting Status.

In addition, Read also inputs from the Data_State because it reads a value from the external subsystem connected to the serial port. That value is used to derive the output Item but does not participate in setting Status.

You might imagine that the body of Serial_Port contains one or more volatile variables connected to the serial port hardware. However, in this example, we decided to implement a version that runs under the Windows operating system. Instead of refining our external state to hardware variables, we make calls to the Windows application programming interface (API).

The body of Serial_Port is not in SPARK. It makes use of APIs that use non-SPARK features such as access types. The body of Serial_Port is written in Ada. However, it could be coded in C, in which case our specification would require Import and Convention aspects on the declared subprograms as described in Section 7.2. As usual, this requires that the programmer review the SPARK aspects in the package specification carefully as their refinement will not be checked by the SPARK tools.

To illustrate how a higher level package might use the low level interface to an external subsystem, consider the following abbreviated specification of a package Terminal. It is used to provide more convenient access to a standard serial terminal connected to the serial port.

```
with Serial_Port ;
package Terminal
   with Spark_Mode => On
is
   type Status_Type is (Success, Insufficient_Space , Port_Failure );
   procedure Get_Line (Buffer    : out String;
                       Count     : out Natural;
                       Status    : out Status_Type)
      with
         Global => (Input => (Serial_Port.Port_State,
                              Serial_Port.Data_State)),
         Depends => ((Buffer, Count, Status) => (Buffer,
                                                 Serial_Port.Port_State,
                                                 Serial_Port.Data_State));
end Terminal;
```

This specification only shows a single procedure, Get_Line, that reads a carriage return terminated string of characters from the terminal and installs them into the given buffer. It returns an error indication if it runs out of space before getting a carriage return character or if the underlying serial port reports an I/O error.

The procedure is decorated with SPARK data and flow dependency contracts written in terms of the state abstractions of the underlying serial port. This appears to be a violation of information hiding: the specification otherwise does not mention the serial port at all, leading one to suppose that it could support other kinds of communications media.

However, despite syntactic appearances, the use of the serial port is not hidden by this package. For example, it is important to open the serial port before calling Get_Line; the dependency of Get_Line on the port state must be declared. Thus, SPARK serves to make explicitly visible dependencies that are semantically visible in any case.

The body of package Terminal is straight forward and shown here in its entirety:

```
with Ada.Characters.Latin_1;
package body Terminal
   with Spark_Mode => On
is
   use type Serial_Port.Status_Type;

   procedure Get_Line (Buffer : out String;
                       Count  : out Natural;
                       Status : out Status_Type) is
      Value       : Serial_Port.Byte;
      Port_Status : Serial_Port.Status_Type;
   begin
      Buffer := (others => ' ');
      Count  := 0;
      Status := Success;
      loop
         pragma Loop_Invariant (Count <= Buffer'Length);

         -- Check to be sure there is space remaining in the buffer.
         if Count = Buffer'Length then
            Status := Insufficient_Space ;
            exit;
         end if;

         Serial_Port.Read (Value, Port_Status);

         -- Check to be sure a byte was successfully read.
         if Port_Status /= Serial_Port.Success then
            Status := Port_Failure;
            exit;
         end if;

         -- We are done if a carriage return is read.
         exit when Character'Val (Value) = Ada.Characters.Latin_1.CR;

         -- Convert Value (a Byte) to a character and append to Buffer
         Buffer (Buffer'First + Count) := Character'Val (Value);
         Count := Count + 1;
      end loop;
   end Get_Line;

end Terminal;
```

The main loop contains a loop invariant pragma that is needed to prove that no buffer overflows will occur. Notice that package Terminal adds useful functionality, completely in SPARK, to the interface of an external subsystem even though direct access to the subsystem is outside of SPARK. The strategy followed here was to wrap the external subsystem in a minimalistic package with a non-SPARK body and then implement as much functionality as possible in SPARK packages that use the subsystem wrapper package.

There are actually two problems with the implementation of Get_Line as previously shown, despite SPARK being able to prove the procedure free of runtime errors. See Exercise 7.13 for more information.

7.3.3 *Hierarchical External State Abstractions*

In Section 4.3.3 we showed how a hierarchy of state abstractions can help simplify a system with complex state. A hierarchy of state abstractions can also be used to simplify a system with complex external state. Let us look at an example.

Air density is perhaps the single most important factor affecting aircraft performance. Density altitude is a commonly used measure of air density. It is the altitude, relative to standard atmosphere conditions, at which the air density would be equal to the indicated air density at the place of observation. The density altitude at a location may be computed from the current temperature, air pressure, and humidity. Here is a specification for a package that uses external sensors to measure these three components and determine the density altitude:

```
package Density_Altitude
   with SPARK_Mode   => On,
         Abstract_State  => (Density_State with External  => Async_Writers)
is
   type Feet  is  range −5_0000 ..  100_000;

   procedure Read (Value :  out  Feet)
      with Global   => (Input => Density_State),
            Depends => (Value => Density_State);
end  Density_Altitude ;
```

This package encapsulates an abstract external state from which the density altitude is returned by a call to procedure Read. The abstract state, Density_State, is given the single external property Async_Writers. This property tells us and SPARK that Density_State will be updated by components external to our system. As the other three properties (Async_Readers, Effective_ Writes, and Effective_Reads) are not listed, they default to false.

We refine Density_State and the Global and Depends aspects of procedure Read in the package body:

```
with  Density_Altitude . Temperature_Unit,
      Density_Altitude . Pressure_Unit ,
      Density_Altitude . Humidity_Unit;
package body Density_Altitude
   with SPARK_Mode => On,
         Refined_State  => (Density_State => (Temperature_Unit.Temp_State,
                                               Pressure_Unit . Press_State ,
                                               Humidity_Unit.Humid_State)) is
   procedure Read (Value :  out  Feet)
      with  Refined_Global   => (Input  => (Temperature_Unit.Temp_State,
                                            Pressure_Unit . Press_State ,
                                            Humidity_Unit.Humid_State)),
              Refined_Depends => (Value => (Temperature_Unit.Temp_State,
                                            Pressure_Unit.Press_State,
                                            Humidity_Unit.Humid_State))
   is
      Temperature : Temperature_Unit.Degrees;
      Pressure    : Pressure_Unit.PSI;
      Humidity    : Humidity_Unit.Percent;
   begin
      Temperature_Unit.Read (Temperature);
      Pressure_Unit.Read (Pressure);
      Humidity_Unit.Read (Humidity);
      Value := Feet (Float (Temperature) + -- A stub for the real
                     Float (Pressure) +    -- equation that calculates
                     Float (Humidity));    -- density altitude
   end Read;

end Density_Altitude;
```

Density_State is refined into abstract states defined in three private child packages, one for each of our input sensors. The Refined_Globals and Refined_Depends are also refined into those abstract states. Values are read from each of the sensors and the density altitude is calculated.

Here is the specification of the private child package for humidity:

```
private package Density_Altitude.Humidity_Unit
   with Spark_Mode     => On,
        Abstract_State => (Humid_State
                             with External => Async_Writers,
                                  Part_Of  => Density_State)
is
   type Percent is range 0 .. 100;

   procedure Read (Value : out Percent)
     with Global  => (Input => Humid_State),
          Depends => (Value => Humid_State);

end Density_Altitude.Humidity_Unit;
```

The abstract state Humid_State of package Humidity_Unit has two options: Part_Of and External. The Part_Of option tells us and SPARK that the abstract state Humid_State is a constituent of the abstract state Density_State. The external property Async_Writers matches that of the abstract state Density_State that it is refining. At least one of the constituents of Density_State must have the external property specified for that abstract state.

Here is the body of the private child package for humidity:

```
with System.Storage_Elements;
package body Density_Altitude.Humidity_Unit
   with Spark_Mode     => On,
        Refined_State  => (Humid_State => Humid_Sensor)
is
   Humid_Sensor : Percent
     with Volatile        => True,
          Async_Writers   => True,
          Address         => System.Storage_Elements.To_Address (16#A1CAF0#);

   procedure Read (Value : out Percent)
      with Refined_Global   => (Input => Humid_Sensor),
           Refined_Depends => (Value => Humid_Sensor)
   is
   begin
      Value := Humid_Sensor;
   end Read;

end Density_Altitude.Humidity_Unit;
```

Here, the abstract state Humid_State is refined to the concrete external variable Humid_Sensor. Again, at least one constituent of the refined abstract state must have the same external property as the abstract state. The private child packages for the temperature and pressure sensors are nearly identical to the one for pressure.[9] Our job is done – all abstract states have been refined to concrete variables.

Summary

- Constructs such as packages and subprograms can be either "in SPARK" or "not in SPARK." The parts of your program that are in SPARK only use the facilities allowed by SPARK and obey SPARK's other restrictions.
- The SPARK_Mode aspect or pragma controls which constructs are to be processed as SPARK. By default SPARK_Mode is off, meaning that the compiler treats your entire program as Ada.
- SPARK_Mode can be turned on by default using a configuration pragma. Otherwise it can be turned on for a given construct using either the SPARK_Mode aspect or the SPARK_Mode pragma.

- The value of SPARK_Mode cannot be changed over small-scale constructs such as subexpressions of an expression or individual statements in a subprogram. SPARK_Mode can only be changed at the granularity of larger constructs such as subprograms and packages.
- It is permitted for code in SPARK to call libraries written in some other language such as C provided the programmer writes a SPARK declaration for each foreign operation. The programmer takes responsibility for ensuring that the library actually conforms to the SPARK aspects declared by the programmer; the SPARK tools cannot check them.
- When calling C from SPARK, it is sometimes necesary to write a small helper function that gathers the return value of the underlying C function and returns it as an "out" parameter. This allows you to write the interface to the function as a procedure and work around SPARK's restriction on functions with out parameters.
- Access to information external to the program such as external hardware subsystems is possible by declaring external variables. Commonly, a package is created that wraps the external subsystem with abstract state declared as an external variable.
- By default external variables are assumed to have asynchronous readers and writers, meaning that the external subsystem accesses those variables at a time outside the program's control.
- By default external variables are assumed to always have effective reads and writes, meaning that each read of the external variable may produce a new value and that every write is significant, even if the same value is written twice in succession.
- External variables with other combinations of asynchronous readers and writers and effective reads and writes can be declared to handle specialized circumstances.
- Hierarchical external state abstractions provide a way to simplify complex external states.

Exercises

7.1 What are the advantages of allowing parts of a program to not be in SPARK? What are the dangers of doing so?

7.2 The Ada compiler defaults to having SPARK_Mode off. Why? When writing a high-integrity program, you would normally want as much of the program to be in SPARK as feasible. How could you arrange for that to happen in a convenient and robust way?

7.3 Which of the following are permitted and which are disallowed? Here A and B are the names of two subprograms. They could be either procedures or functions or one of each.
   a. A's body is in SPARK and calls B, which has a SPARK declaration and a SPARK body.
   b. A's body is in SPARK and calls B, which has a SPARK declaration and a non-SPARK body.
   c. A's body is in SPARK and calls B, which has a non-SPARK declaration and a non-SPARK body.
   d. A's body is not in SPARK and calls B, which as a SPARK declaration and a SPARK body.
   e. A's body is not in SPARK and calls B, which has a SPARK declaration and a non-SPARK body.
   f. A's body is not in SPARK and calls B, which has a non-SPARK declaration and a non-SPARK body.

7.4 It is not permitted for a subprogram to have a non-SPARK declaration and a SPARK body. Why would this be disallowed?

7.5 Add a function Check_Overlap to package Interval_Tree described in Section 7.1.1 with the following profile:

```
function Check_Overlap(Item : in Interval ) return Boolean;
```

The function should return True if the given interval overlaps with at least one interval in the tree, otherwise it should return False. Your function should run in $O(\log n)$ time and have appropriate SPARK aspects.

7.6 Suppose you write a package with a non-SPARK body and a SPARK specification. Which of the following aspects that may appear on the subprogram declarations in the specification would actually be checked and when would that checking occur?
   a. Global
   b. Depends
   c. Pre
   d. Post

7.7 The wrapper package presented on page 266 shows imported declarations in the body of the wrapper package. Is it necessary for those declarations to be in the body or could they continue to be in a package specification of their own? If they could be in their own specification, modify the wrapper package presented to show how it would look. If they must appear in the body, explain why.

7.8 Write a full Ada version of the C helper function presented on page 267. An outline of how to proceed can be found at the end of Section 7.2. You may find it useful or appropriate to complete package Messages_Wrapper started on page 265. (Hint: The imported declaration of the underlying C function can be made local to the helper procedure. Thus, only the helper procedure needs to have SPARK_Mode set to Off.)

7.9 Pick a small C library that you have already written. Write a SPARK specification for your library and a demonstration program, in SPARK, that uses your library.

7.10 Repeat Exercise 7.6 for the case in which the package body is really a C library.

7.11 Define the phrases *effective read* and *effective write*. Why is it necessary to sometimes specify that reads and writes of external variables are always effective?

7.12 Write a package DIP that wraps a memory mapped register holding the state of eight DIP switches the user can adjust to provide input to your program. Your package should provide a subprogram for reading the switches and an appropriately declared external state abstraction.

7.13 The implementation of Get_Line in package Terminal suffers from two problems. First, a buffer of size zero will always cause the Insufficient_Space error to be returned. However, it should be possible to pass a zero-sized buffer successfully provided the user enters a blank line in response. Second, if the buffer is returned completely filled with characters, there is no way to distinguish between the case when the line is exactly the length of the buffer and when the user is trying to enter too much data. Fix these problems while maintaining SPARK's proof of freedom from runtime error.