4

Dependency Contracts

In this chapter we describe SPARK's features for describing data dependencies and information flow dependencies in our programs. This analysis offers two major services. First, it verifies that no uninitialized data is ever used. Second, it verifies that all results computed by the program participate in some way in the program's eventual output – that is, all computations are *effective*.

The value of the first service is fairly obvious. Uninitialized data has an indeterminate value. If it is used, the effect will likely be a runtime exception or, worse, the program may simply compute the wrong output. The value of the second service is less clear. A program that produces results that are not used is at best needlessly inefficient. However, ineffective computations may also be a symptom of a larger problem. Perhaps the programmer forgot to implement or incompletely implemented some necessary logic. The flow analysis done by the SPARK tools helps prevent the programmer from shipping a program that is in reality only partially complete.

It is important to realize, however, that flow analysis by itself will not show your programs to be free from the possibility of runtime errors. Flow analysis is only the first step toward building robust software. It can reveal a significant number of faults, but to create highly robust systems, it is necessary to use proof techniques as described in Chapter 6.

As described in Chapter 1, there are three layers of analysis to consider in increasing order of rigor:

1. Show that the program is legal Ada that abides by the restrictions of SPARK where appropriate. The most straightforward way to verify this is by compiling the code with a SPARK-enabled compiler such as GNAT.
2. Show that the program has no data dependency or flow dependency errors. Verify this by running the SPARK tools to "examine" each source file.

3. Show that the program is free from runtime errors and that it honors all its contracts, invariants, and other assertions. Verify this by running the SPARK tools to "prove" each source file.

We recommend making these three steps explicit in your work. Move on to the next step only when all errors from the previous step have been remedied. This chapter discusses the second step.

4.1 Data Dependency Contracts

The *data dependency contract* describes what global data a subprogram depends on and whether that data is read, written, or both. The data dependency contract appears as a Global aspect on a subprogram's declaration.

The use of global data is normally discouraged. This is primarily because it is difficult for programmers to reason about the behavior of a program when it makes frequent, undisciplined use of such data. Normally, one expects a subprogram to only read its **in** parameters and modify its **out** parameters. Reads and writes the subprogram makes to global data is not obvious from the call site and is easy to ignore. For example, if a call Process(X, Y) reads a global variable Z, it is easy to forget to initialize Z before the call is made.

SPARK's data dependency contracts allow the programmer to explicitly specify what global data each subprogram uses in a manner that is similar to the way subprogram parameters are specified. Thus, global data read and written is clearly defined as part of each subprogram's declaration. The SPARK tools use this information to ensure all global data is initialized before it is needed and all results written to global data are used. The flow analysis done by the SPARK tools verifies that the body of a subprogram manipulates both parameters and global data as described by the parameter list and the data dependency contract.

As an example, consider the following abbreviated package specification for a package that performs raster graphics drawing:

```
pragma SPARK_Mode (On);
package Raster_Graphics is

   Workspace_Size : constant := 100;
   type Coordinate_Type is new Integer range 1 .. Workspace_Size;
   type Point is
      record
         X, Y : Coordinate_Type;
      end record;
```

```
   type Line_Algorithm_Type is (Bresenham, Xiaolin_Wu);
   type Status_Type is (Success, Line_Too_Short, Algorithm_Not_Implemented);

   Status         : Status_Type;
   Line_Algorithm : Line_Algorithm_Type;
   Line_Count     : Natural;

   procedure Draw_Line (A, B : in Point)
     with
       Global => (Input => Line_Algorithm,
                  Output => Status,
                  In_Out => Line_Count);

end Raster_Graphics;
```

The first line of this package specification, **pragma** SPARK_Mode (On), informs the SPARK tools that this unit follows all the rules of the SPARK language. We say that this unit is "in SPARK." We can write programs in which certain units are written in SPARK while other units are written in full Ada or another programming language. We discuss mixing SPARK, full Ada, and C in Chapter 7. We can also use an aspect to state that a unit is in SPARK as follows:

```
package Raster_Graphics
   with Spark_Mode => On
is
```

In this book we use both the pragma and the aspect to state that a unit is in SPARK. We talk more about SPARK mode in Section 7.1.1.

Now let us look at the details of the package specification Raster_Graphics. The package draws on a square workspace of size 100 pixels. It first introduces Coordinate_Type to distinguish other integer values from values used to represent coordinates on the drawing space. The package then defines a record type Point for representing specific two-dimensional positions.

The procedure Draw_Line draws a line between two points A and B. The data dependency contract, as given by the Global aspect, specifies three global variables used by the procedure. Three modes can be used in data dependency contracts to indicate the direction of data flow just as subprogram parameters can use three modes for that purpose. In this case the desired line drawing algorithm is read, a status value representing the success or failure of the procedure is written, and a counter of the total number of lines ever drawn is updated. Each mode can only appear once but can be associated with an arbitrary list of variables enclosed in parentheses.

Global variables used by a subprogram behave similarly to parameters where the actual argument is fixed. However, it is not necessary in the data dependency contract to specify the types of the global variables. To be used in the contract, the variables must be visible at that point in the program. Thus, their types are available from their declarations elsewhere. In the preceding example, the global variables are declared inside the specification of package Raster_Graphics.

One important rule is that the variables mentioned in the data dependency contract have to be entire objects. For example, it is not permitted to use a single array element or record component. The variable must be the entire array or record even if the subprogram accesses only one component of the composite entity.

SPARK also requires that functions only read from global data. Thus, the modes Output and In_Out are illegal for functions. This is consistent with SPARK's restriction that functions have only **in** parameters. It is necessary to ensure that the unspecified evaluation order of subexpressions remains deterministic.

Consider, for example, an assignment statement such as

```
X := F(A) + F(B);
```

If F were allowed to output to a global variable, the final result stored in that variable would depend on the order in which the operands to + are evaluated. However, that order is unspecified by the language. To ensure the program always produces well-defined, predictable results, outputs cannot be allowed to depend on such unspecified ordering. The concern is eliminated by forbidding functions from having side effects such as writing to global data. This problem does not arise in the case of procedures because each procedure is called in its own statement, and the order in which statements execute is specified.

Procedure Draw_Line described earlier has two ordinary parameters and makes use of three global variables. One might wonder why some or all of those global variables were not declared as parameters instead. Doing so would be considered better style in the eyes of many developers.

There are, however, at least two cases in which the use of global data is reasonable. Perhaps the most important use of global data is to hold the internal state of a variable package. We used this approach with the Bingo basket package in Section 3.4. We discuss the SPARK aspects for managing internal state in Section 4.3.

Global data is also useful when a nested subprogram accesses the local variables or parameters of an enclosing subprogram. To demonstrate this, consider the following abbreviated body of package Raster_Graphics. For now we will

continue to let Draw_Line access three global variables, although, in a more reasonable application, some or all of those global variables might be passed into Draw_Line as parameters.

```
pragma SPARK_Mode(On);
package body Raster_Graphics is

   procedure Draw_Line(A, B : in Point) is

      Min_Distance : constant Coordinate_Type := 2;

      -- Verify that A, B are far enough apart.
      -- Write error code to Status if not.
      procedure Check_Distance
        with
          Global => (Input => (A, B),
                     Output => Status)
      is
         Delta_X : Coordinate_Type := abs (A.X - B.X);
         Delta_Y : Coordinate_Type := abs (A.Y - B.Y);
      begin
         if Delta_X**2 + Delta_Y**2 < Min_Distance**2 then
            Status := Line_Too_Short;
         else
            Status := Success;
         end if;
      end Check_Distance;

   begin
      Check_Distance;
      if Status = Success then
         case Line_Algorithm is
            when Bresenham =>
               -- Algorithm implementation not shown...
               Line_Count := Line_Count + 1;

            when Xiaolin_Wu =>
               Status := Algorithm_Not_Implemented;
         end case;
      end if;
   end Draw_Line;

end Raster_Graphics;
```

We assume, for purposes of illustration, that Draw_Line requires the end points of the line to be sufficiently far apart. Perhaps it is required for very short lines to be represented as large dots, or perhaps the underlying drawing hardware does not work reliably for short lines. In any case Draw_Line needs to check the distance between the points it is given and uses a helper subprogram Check_Distance to do so.

However, from Check_Distance's point of view the parameters of the enclosing procedure, as well as the local variables of the enclosing procedure that are declared above Check_Distance (if any), are global variables. The flow analysis done by the SPARK tools thus requires any global variables that are used to be mentioned in Check_Distance's data dependency contract.

Notice that Check_Distance also writes to the global variable Status. It must specify this in its data dependency contract despite the fact that the enclosing procedure has already done so. The body of each subprogram, even nested subprograms, is analyzed on its own so all information moving into and out of each subprogram must be declared. Flow analysis will ensure that the modes are all consistent. For example, if Check_Distance's data dependency contract was

```
procedure Check_Distance
  with
    Global => (In_Out => (A, B),
               Output => Status) ...
```

flow analysis would object because A and B are **in** parameters of the enclosing subprogram and thus may not be modified. The constant Min_Distance in the body of package Raster_Graphics is handled differently. Because it is a constant that is initialized with a static expression,[1] it can be used by any of the subprograms for which it is visible without being mentioned in the data dependency contract of those subprograms.

Finally, notice that Draw_Line increments Line_Count. Flow analysis verifies that the mode on that global variable is consistent with this usage. Also, the SPARK tools will understand that, because the global variable has an input mode, it must be initialized in some way before Draw_Line can be called.

4.2 Flow Dependency Contracts

An important part of the flow analysis done by the SPARK tools is to track which values are used in the computation of which results. Techniques for doing flow analysis inside a subprogram are well known and described in detail in, for example, textbooks on compiler design (Aho et al., 2007). However, to

accomplish the larger goal of ensuring no information is misused in the overall program, it is necessary to extend flow analysis across subprograms, including across subprograms in different packages.

Ada promotes the construction of large software systems as collections of loosely coupled packages. These packages are developed independently, often in parallel, and only integrated into the final program after they have been separately tested and analyzed. SPARK would be useless for realistic programs if it did not support this style of development.

SPARK allows programwide flow analysis to be carried out on packages independently by requiring the programmer to declare the way information moves into and out of a subprogram as part of that subprogram's declaration. These *flow dependency contracts* are used when analyzing code that calls the subprogram and checked when analyzing the implementation of the subprogram.

As an example, consider a procedure that searches a string for the first occurrence of a given character starting at a given position. It returns a status value of true if the character is found along with its location in the string. The declaration of this procedure might look like the following:

```
procedure Search (Text     : in  String;
                 Letter   : in  Character;
                 Start    : in  Positive;
                 Found    : out Boolean;
                 Position : out Positive)
   with
      Global  => null,
      Depends => (Found    => (Text, Letter, Start),
                  Position => (Text, Letter, Start));
```

The flow dependency contract, expressed using the Depends aspect, describes how each output of the procedure depends on the inputs. In this case the value produced for Found depends on the text being searched, the letter being searched for, and the starting position of the search. The value produced for Position depends on the same three inputs.

Based on the intended behavior, or *semantics*, of Search, it is intuitively clear that the dependencies expressed above are correct. For example, Found clearly depends on Start. If Letter appears only in the first position of the text, the search will succeed if Start is the beginning of the string but fail if Start is in the middle of the string. Similarly, Found clearly depends on the letter being searched for and on the text being searched.

Notice that we can meaningfully declare the flow dependency contract for procedure Search without having to implement it – or even look at its

implementation. Thus, the flow dependency contracts can be written in a package specification before the body is written. They represent a formal way of expressing a certain aspect of each subprogram's semantic behavior.

As an aside, we note that Search does not make use of any global data. This can be stated explicitly by adding

```
Global => null
```

to the aspect specification on Search. However, not specifying data (or flow) dependency contracts explicitly is not necessarily an error. We describe in more detail the effect of leaving off the contracts in Section 4.5.

Let us look at an example that illustrates flow analysis through multiple procedures. Consider another procedure that is intended to take strings in the form "NAME=NUMBER" and return the specified number if the name matches a given name. The declaration of this procedure, with its flow dependency contract, is as follows:

```
procedure Get_Value (Text  : in     String;
                     Name  : in     String;
                     Value : in out Integer)
  with
    Global  => null,
    Depends => (Value => (Text, Name, Value));
```

Here, the intention is for Value to be initialized to a default before Get_Value is called. If the given string is malformed or if it does not include Name as a prefix, Value is unchanged.

The body of Get_Value is as follows:

```
procedure Get_Value (Text  : in     String;
                     Name  : in     String;
                     Value : in out Integer) is

   Equals_Found    : Boolean;
   Equals_Position : Positive;
begin
   if Text'Length > 0 then
      Search (Text     => Text,
              Letter   => '=',
              Start    => Text'First,
              Found    => Equals_Found,
              Position => Equals_Position);
```

```
         if Equals_Found then
            if Name = Text (Text'First .. Equals_Position - 1) then
               Value := Integer'Value (Text (Equals_Position + 1 .. Text'Last));
            end if;
         end if;
      end if;
   end Get_Value;
```

The assignment to the parameter Value requires the conversion of a string of digits, such as "4932", to an integer. We use the 'Value attribute to accomplish this conversion.

Notice that Get_Value makes use of the procedure Search specified on page 105. When doing the flow analysis of Get_Value, the SPARK tools will know, from the flow dependency contract on the declaration of Search, that Equals_Found depends on Text. Furthermore, because Equals_Found is used in the controlling expression of an **if** statement, any values written inside that statement also depend, indirectly, on Text. Finally, because Name is used in the controlling expression of the **if** statement enclosing the assignment to Value, it follows that Value depends on Name. Notice also that Value depends on its own input because, if Equals_Found is false, the procedure returns with Value unchanged.

Our simple "by inspection" analysis has shown that the flow dependency contract is obeyed. However, if the procedure was incomplete, it is possible that some part of the flow dependency contract would be violated and flow analysis would produce diagnostics as appropriate. Flow analysis helps the programmer avoid shipping an incomplete program.

Continuing this example, suppose that later a change is made to Search that changes the way information flows through that procedure. The flow dependency contract on Search would have to be updated to reflect this change. Thus, Get_Value would need to be reanalyzed, and it is possible it might then contain flow errors as a result of the change to Search. In this way, changing the body of Search has the potential of causing a cascade of changes to the callers of Search, to the callers of those callers, and so forth.

This cascade might sound unappealing, but it is no different than what might happen if a parameter was removed from Search's parameter list or if the type of a parameter was changed. In that case, the callers of Search would have to be edited appropriately along with, potentially, the callers of those callers, and so forth. The flow dependency contract is part of the subprogram's interface just as is the parameter list. It serves to expose more information about the subprogram's behavior than can be done by the parameter list alone. The flow analysis done by the SPARK tools checks the consistency of this additional

information just as a traditional Ada compiler checks the consistency of each call's arguments.

There are many implementations of Search that will satisfy the parameter types and modes. A large number of those implementations have nothing to do with searching a string for a particular character, despite whatever suggestive names are used. The flow dependency contract rules out some of those implementations as flow errors leaving behind a smaller set of implementations that conform to both the parameter types and modes and the flow dependency contract.

However, it is still easy to see that many incorrect implementations of Search exist that satisfy the flow dependency contract. To further tighten the possibilities, the programmer can use pre- and postconditions to describe more precisely the relationship between the subprogram's outputs and its inputs. We introduced the Pre and Post aspects with the bounded queue package in Section 3.3. We cover these aspects in more detail in Section 6.2.

One important point, however, is that flow dependency contracts must be complete. If you use the Depends aspect at all, you must fully describe all flows into and out of the subprogram, including flows involving global data described by the data dependency contract. In contrast, as you will see, pre- and postconditions are often incomplete. They may not describe every aspect of the relationship between inputs and outputs. In this respect effective use of pre- and postconditions depends significantly on your skill with them and on the proof technology being used. However, flow contracts are more reliable in the sense that your program will simply not pass SPARK examination until it is completely free of flow errors.

4.2.1 *Flow Dependency Contract Abbreviations*

In general, a flow dependency contract consists of an association between each output of a subprogram and a list of the inputs on which that output depends. Writing these associations can be tedious and repetitive so SPARK provides two important abbreviations that make writing flow dependency contracts easier.

First, it is common for multiple outputs to depend on the same inputs. An especially common case, although not universal, is when each output depends on all inputs. The procedure Search shown previously is like this. The syntax of flow dependency contracts allows the programmer to associate a list of outputs with a list of inputs. Thus, a contract such as

```
with
  Depends => (Found => (Text, Letter,  Start ),
              Position  => (Text, Letter,  Start ))
```

can be abbreviated as

```
with
  Depends => ((Found, Position) => (Text, Letter, Start))
```

The understanding is that each output mentioned in an output list depends on all of the inputs mentioned in the associated input list.

For the case of **in out** parameters or global variables that are In_Out, it is common for the output value of the parameter or variable to depend on its own input value. The parameter Value in the previously shown procedure Get_Value is like this. In that case, the symbol =>+ can be used to indicate that each output on the left side depends on itself as well as on all inputs on the right side. For example, the contract

```
with
  Depends => (Value => (Text, Name, Value))
```

can be abbreviated as

```
with
  Depends => (Value =>+ (Text, Name))
```

In the important case where an **in out** parameter or In_Out global variable depends on only itself, a flow dependency contract such as Value => Value can be abbreviated to Value =>+ **null**. The abbreviated form can be read as, "Value depends on itself and nothing else."

Sometimes an output depends on no inputs whatsoever. This occurs when a subprogram writes a value from "out of the blue" to initialize an output. For example,

```
procedure Initialize (Value : out Integer)
  with
    Depends => (Value => null);
```

The flow dependency contract Value => **null** can be read as, "Value depends on nothing." It means the subprogram sets the value without reference to any input.

Finally, it is also possible to give a flow dependency contract that uses **null** on the output side of a dependency as shown in the following example:

```
procedure Update (Value  : in out Integer;
                  Adjust : in     Integer)
  with
    Depends => (Value =>+ null,
                null  => Adjust);
```

Here, Value depends only on itself, and nothing depends on Adjust. Thus, Adjust is not used in the computation of any results. Such a flow dependency contract

is unusual, but it explicitly documents that a particular value is not (yet) being used.

4.3 Managing State

For our purposes, the *state of a package* consists of the values of all global variables defined inside the package together with the state of any packages nested within the package. By placing these entities within the private part of the package specification or the package body, the state of a package is hidden from its clients. When a subprogram in a package is called, the caller is not directly aware of any global variables or nested packages the subprogram uses or modifies. Restricting the visibility of data as much as feasible is an important principle of software engineering. This hiding allows the representation of that data to be changed at a later time with minimal impact on the rest of the program. If clients of a package only interact with the package's internal state by way of a small set of public subprograms, the precise design of that state is not significant to the clients.

As an example consider a package that encapsulates a datebook. The package might provide subprograms for adding events to the datebook, removing events from the datebook, and enumerating the events currently stored in the datebook. The specification of such a package could be, in part as follows:

```
with Dates;
package Datebook is
   type Status_Type is (Success, Description_Too_Long, Insufficient_Space );

   procedure Add_Event (Description : in  String;
                        Date        : in  Dates.Datetime;
                        Status      : out Status_Type);
end Datebook;
```

Here we assume the package Dates provides a type Dates.Datetime that represents a combined date and time.

Each event that is entered into the datebook has a short description and an associated date and time when the event occurs. For purposes of this example, the duration of the event and other information about the event is ignored. The procedure Add_Event, one of several procedures the package might contain, adds information about an event to the datebook and returns a status indication, by way of an out parameter, to report the success or failure of that operation.

Everything seems fine, but where, exactly, is the datebook to which events are being added? In this case the datebook is presumably implemented in the form of global data inside the body of package Datebook. It is the package's state.

It might be implemented as a simple array of records where each record stores information about a single event, or it might be implemented in some more elaborate way. Perhaps the package maintains indexes to speed up datebook queries. Perhaps the package packs event descriptions into some auxiliary data structure to save space. None of this matters to the users of the package.

However, the fact that the package contains internal state that is accessed and modified by the public subprograms is important to flow analysis. For example, Add_Event updates the internal state by adding a new event record to the datebook. Because the new state depends on the existing state, Add_Event must effectively read the existing state. This implies that the internal state of the package must already be initialized in some way before Add_Event can be called.

To talk about the internal state in SPARK aspects, it is necessary to give that internal state a name. It is not necessary to declare the internal state fully. In fact, doing so would expose the information hidden by the package. The clients are not interested in the details of how the internal state is organized. They are only interested in the fact that it exists and in how it is read and updated by the public subprograms. Thus, all that is needed is for the internal state to be abstracted into a single name called a *state abstraction*.

The following example shows a SPARK version of the Datebook specification. Here a state abstraction named State is introduced to represent the internal state of the package:

```
with Dates;
package Datebook
  with
    Abstract_State => State
is
   type Status_Type is (Success, Description_Too_Long, Insufficient_Space);

   procedure Add_Event (Description : in  String;
                        Date        : in  Dates.Datetime;
                        Status      : out Status_Type)
      with
        Global  => (In_Out => State),
        Depends => (State  =>+ (Description, Date),
                    Status => (Description, State));
end Datebook;
```

The name State in this example is a name chosen by the programmer to identify the internal state of the package. Because we wish to keep the nature of this internal state as hidden as possible, we cannot easily give it a more

descriptive name. As you will see in Section 4.3.2, it is sometimes desirable to break the abstract state into two or more components. In that case, we would give the components names that distinguish them clearly.

We have enhanced the declaration of Add_Event with descriptions of the effects it has on the package's internal state. Notice that State is treated as a kind of global variable. In fact, that is exactly what it is. The state abstraction represents the global data inside the package. The details of this hidden data are not needed by the clients of the package. They need only be aware that the package has a state and that the subprograms they call use and/or change that state.

The data and flow dependency contracts on Add_Event indicate that the status depends on the state of the datebook and on the incoming description. Add_Event can fail if the datebook is already full or if the description is overly large.

If the only public subprogram in the package is Add_Event as shown in the preceding example, the flow analysis done by the SPARK tools will complain that the internal state of the package has no way of being initialized. Procedure Add_Event requires that State has a value before it is called. How does State get its initial value?

One approach would be to provide an initialization procedure. It might be declared as follows:

```
procedure Initialize
  with
    Global  => (Output => State),
    Depends => (State => null);
```

This procedure takes no parameters because it initializes global data inside the package. The data and flow dependency contracts say this by declaring that the procedure gives the state abstraction a value from "nothing." With this procedure it is now possible to call Add_Event correctly by first calling Initialize. Flow analysis will ensure that this happens.

In some cases the global data inside a package can be initialized without the help of an explicit procedure call. As we saw in Section 3.4.1, the internal state of a variable package can be initialized with suitable static expressions or with initialization code executed when the package body is elaborated. In such cases the package initializes itself. The aspect declaration on the package can be changed to reflect this behavior as the following example shows:

```
with Dates;
package Datebook
  with
    Abstract_State => State,
     Initializes   => State
```

```
is
   type Status_Type is (Success, Description_Too_Long, Insufficient_Space );

   -- No Initialize procedure needed
   procedure Add_Event (Description : in  String;
                        Date        : in  Dates.Datetime;
                        Status      : out Status_Type)
      with
        Global  => (In_Out => State),
        Depends => (State  =>+ (Description, Date),
                    Status => (Description, State) );
end Datebook;
```

If during program development the internal state becomes complicated enough to require a special initialization procedure, one can be added and the `Initializes` aspect removed. Flow analysis will ensure that the new procedure will get called as needed.

The package `Datebook` presented so far is a kind of variable package as discussed in Section 3.4. It allows a single datebook variable to be manipulated. Before looking at the body of package `Datebook`, it is useful to consider an alternative implementation as a type package, as described in Section 3.3. The following version, named `Datebooks` (note the plural), provides a `Datebook` private type and has no internal state. Instead, the components of the private type hold the state of each `Datebook` object.

```
with Dates;
package Datebooks is
   type Datebook is private;

   type Status_Type is (Success, Description_Too_Long, Insufficient_Space );

   procedure Add_Event (Book        : in out Datebook;
                        Description : in     String;
                        Date        : in     Dates.Datetime;
                        Status      :    out Status_Type)
      with
        Depends => (Book =>+ (Description, Date),
                    Status => (Description, Book) )
private

   -- Provide a full definition for the private type Datebook

   Maximum_Description_Length : constant := 128;
   subtype Description_Index_Type is
```

```
      Positive range 1 .. Maximum_Description_Length;
   subtype Description_Count_Type is
      Natural range 0 .. Maximum_Description_Length;
   subtype Description_Type is String(Description_Index_Type);

   -- Each Event_Record handles exactly one datebook entry.
   type Event_Record is
      record
         Description_Text : Description_Type;
         Description_Size : Description_Count_Type;
         Date             : Dates.Datetime;
         Is_Used          : Boolean;
      end record;

   subtype Event_Index_Type is
      Positive range 1 .. Maximum_Number_Of_Events;
   type Datebook is
      array (Event_Index_Type) of Event_Record;

end Datebooks;
```

In this version the caller must create a Datebook object and explicitly pass it to the various subprograms in the package. Accordingly, those subprograms, such as Add_Event, must now be given an additional parameter. The subprograms no longer have a data dependency contract, but the flow dependency contract appropriate for the new parameter mimics the flow dependency contract used with the state abstraction in the original version of the package.

This version has some advantages over the previous version. First, it allows clients to create and manipulate many different Datebook objects. Second, package Datebooks has no internal state, which presents advantages in multi-tasking environments. However, the original version was easier to use because clients did not need to create their own datebooks. The most appropriate approach depends on the application's needs.

The private section, shown in full earlier, seems complicated. However, the original Datebook package needs similar declarations in the package body to fully define the internal state of that package. This is described in more detail in the next section.

4.3.1 *Refinement*

When a state abstraction is used, the body of the package must explicitly declare which global variables and nested package state, if any, compose the abstract

state. Such entities are called the *constituents* of the state abstraction, and the process of breaking a state abstraction into its constituents is called *refinement*. In simple cases, a state abstraction might be refined to a single package global variable, thus, having a single constituent. In more complex cases, the state abstraction will be refined to multiple package global variables.

For example, suppose the Datebook package is implemented by way of an array of records in which each record contains information about one event, similar to the way the Datebooks package works. The array would thus be the single constituent of the state abstraction. The package body might look, in part, as follows:

```
package body Datebook
  with
    Refined_State => (State => Event_Array)
is
   -- Provide an appropriate type definition .
   Maximum_Description_Length : constant := 128;
   subtype Description_Index_Type is
      Positive range 1 .. Maximum_Description_Length;
   subtype Description_Count_Type is
      Natural range 0 .. Maximum_Description_Length;
   subtype Description_Type is String( Description_Index_Type );

   -- Each Event_Record handles exactly one datebook entry.
   type Event_Record is
      record
         Description_Text : Description_Type;
         Description_Size : Description_Count_Type;
         Date             : Dates.Datetime;
         Is_Used          : Boolean;
      end record;

   subtype Event_Index_Type is
      Positive range 1 .. Maximum_Number_Of_Events;
   type Event_Array_Type is
      array(Event_Index_Type) of Event_Record;

   Event_Array : Event_Array_Type;

   -- Body of add event operation
   procedure Add_Event . . .
end Datebook;
```

The Refined_State aspect on the package body specifies which global variables inside the package form the constituents of the abstract state previously declared. In our example, the abstract state State is refined to the single global variable Event_Array. It is an error to have an abstract state in a package specification without also refining it in the package body.

In addition, the data and flow dependency contracts in the package specification must be refined in the package body to explicitly specify the effects those subprograms have on the refined state. These refined contracts appear inside the package body as they reference information that is internal to the package. The body of procedure Add_Event might be

```
procedure Add_Event (Description : in  String;
                     Date        : in  Dates.Datetime;
                     Status      : out Status_Type)
   with
      Refined_Global  => (In_Out => Event_Array),
      Refined_Depends => (Event_Array =>+ (Description, Date),
                                Status    => (Description, Event_Array))
is
   Found     : Boolean;     -- Is there an available slot?
   Available : Event_Index_Type := Event_Index_Type'First; -- Location
begin
   -- If the given description won't fit there is no point continuing.
   if Description'Length > Maximum_Description_Length then
      Status := Description_Too_Long;
   else
      -- Search for a free slot in the event array.
      Found := False;
      for Index in Event_Index_Type loop
         if not Event_Array (Index).Is_Used then
            Available := Index;
            Found := True;
            exit; -- We found a available slot in the array
         end if;
      end loop;

      if not Found then
         -- If there is no free slot return an error.
         Status := Insufficient_Space ;
      else
         -- Otherwise fill in the free slot with the incoming information.
         -- Need to pad the description with blanks.
```

```
            Event_Array ( Available ). Description_Text := Description &
              (1 .. Maximum_Description_Length - Description'Length => ' ');
            Event_Array ( Available ). Description_Size := Description 'Length;
            Event_Array ( Available ). Date      := Date;
            Event_Array ( Available ). Is_Used := True;
            Status := Success;
         end if;
      end if;
   end Add_Event;
```

Notice that the Refined_Global and Refined_Depends aspects are expressed in terms of the actual package global variable rather than in terms of the state abstraction as was done in the package specification.

The example so far shows a one-to-one correspondence between a state abstraction and a single constituent variable. A more elaborate implementation of the Datebook package might include an index that allows events to be looked up quickly on the basis of a commonly used field. In that case the state abstraction might be refined to two package global variables as follows:

```
package body Datebook
  with
    Refined_State => (State => (Event_Array, Event_Index))
is
   type Event_Array_Type is . . .
   type Event_Index_Type is . . .

   Event_Array : Event_Array_Type;
   Event_Index : Event_Index_Type;
end Datebook;
```

Similarly, the refined data and flow dependency contracts on the subprograms must be updated to describe the effects those subprograms have on the individual constituents. It is likely that Add_Event will modify both the datebook itself and its index. For example,

```
package body Datebook
  with
    Refined_State => (State => (Event_Array, Event_Index))
is
  . . .

   procedure Add_Event (Description : in  String;
                        Date        : in  Dates.Datetime;
                        Status      : out Status_Type)
```

```
   with
     Refined_Global   => (In_Out => (Event_Array, Event_Index)),
     Refined_Depends => (Event_Array =>+ (Description, Date),
                         Event_Index =>+ (Description, Date),
                         Status      => (Description , Event_Array) )
   is
     -- Local variables of Add_Event omitted
   begin
     -- Body of Add_Event omitted
   end Add_Event;

end Datebook;
```

Notice that in this case Status does not depend on the Event_Index. Presumably, the success or failure of Add_Event is not affected by the index. In the package specification, Status depends on the overall state abstraction State as it depends on one of that abstraction's constituents, namely the Event_Array.

The important point is that the new version of the Datebook package has changed the internal organization of the datebook without changing the specification of the package. Not only do clients of this package not need to be recompiled, but flow analysis of those clients does not need to be redone. Of course, the analysis of package Datebook's body does need to be redone, causing the SPARK tools to consider the new refined contracts in light of the changes in the implementation.

4.3.2 *Multiple State Abstractions*

In the formulation of the Datebook package so far, a single state abstraction is used to abstract two constituents: the datebook itself and its index. This maximizes the amount of information hidden from the package's clients. Suppose, now, that one wanted to add a public subprogram to compact the index. The specification must be changed to include this subprogram and to declare how it interacts with the state abstraction provided by the package. We give this package the name Indexed_Datebook to inform its users of its different nature as compared to the previously described Datebook package. Because the index is now public information, it is appropriate to reflect that information in the package's name:

```
with Dates;
package Indexed_Datebook
  with
    SPARK_Mode   => On,
    Abstract_State => State
```

```
is
   type Status_Type is (Success, Description_Too_Long, Insufficient_Space );

   procedure Initialize
     with
       Global  => (Output => State),
       Depends => (State => null);

   procedure Compact_Index
     with
       Global  => (In_Out => State),
       Depends => (State =>+ null);

   procedure Add_Event (Description : in  String;
                        Date        : in  Dates.Datetime;
                        Status      : out Status_Type)
     with
       Global  => (In_Out => State),
       Depends => (State  =>+ (Description, Date),
                   Status => (Description, State) );

end Indexed_Datebook;
```

Because Compact_Index only updates the internal state, it need not take parameters. The flow dependency contract shows that the new state depends only on the old state. Because the data dependency contract declares that the state has mode In_Out, flow analysis will require that the initialization procedure for the package be called before Compact_Index can be used.

However, using a single state abstraction can sometimes cause unnecessary and undesirable coupling between the subprograms in a package. For example, suppose the event index can be initialized directly by the package and does not need the help of an initialization procedure. In that case, it should be possible to call Compact_Index without first calling Initialize. Perhaps some programs may find it convenient to do so.

These ideas can be expressed by using multiple state abstractions. The following version of the specification shows this. The name State has been dropped in favor of two more descriptive names.

```
with Dates;
package Indexed_Datebook_V2
  with
    SPARK_Mode   => On,
    Abstract_State => (Book, Index),
     Initializes   => Index
```

```
is
   type Status_Type is (Success, Description_Too_Long, Insufficient_Space );

   procedure Initialize
     with
       Global  => (Output => (Book, Index)),
       Depends => ((Book, Index) => null);

   procedure Compact_Index
     with
       Global  => (In_Out => Index),
       Depends => (Index =>+ null);

   procedure Add_Event (Description : in  String;
                        Date        : in  Dates.Datetime;
                        Status      : out Status_Type)
     with
       Global  => (In_Out => (Book, Index)),
       Depends => (Book  =>+ (Description, Date),
                   Index =>+ (Description, Date),
                   Status => (Description, Book));

end Indexed_Datebook_V2;
```

In addition to declaring two state abstractions, the package declares that one of those abstractions is automatically initialized. The data and flow dependency contracts on procedure Compact_Index indicate that it only needs the Index abstraction to be initialized before it can be used. Flow analysis will now allow Compact_Index to be called before Initialize is called.

The body of package Indexed_Datebook needs to also show how each state abstraction is refined. For example,

```
package body Datebook
  with
    Refined_State => (Book => Event_Array,
                      Index => Event_Index)
is
   . . .
end Datebook;
```

The refined aspects on the subprogram bodies remain as before because they were written in terms of the constituents anyway. Flow analysis will verify, of course, that the refined contracts on the bodies are consistent with the contracts in the declarations of the subprograms.

This approach reduces the *false coupling* between subprograms as a result of an overly abstracted view of the package's internal state. However, the price that is paid is more exposure of the package's internal structure to clients.

Suppose after making this change it was decided that the index was unnecessary and it was removed. The data and flow dependency contracts in the specification would need to be updated to reflect this change, necessitating a reanalysis of all clients along with possible modifications to the contracts of those clients. One might be tempted to replace the procedure Compact_Index with a version that does nothing so as to avoid removing the procedure outright (and breaking client code that calls it). However, the SPARK contracts for the new procedure would indicate that it has no effect and flow analysis would produce a warning because of this. Thus, the introduction of multiple state abstractions in a single package should be done cautiously as it is a form of information exposure; undoing such a change later might be more difficult than one desires.

4.3.3 *Hierarchical State Abstractions*

Complex state information may be simplified by the use of state hierarchies implemented by a hierarchy of packages. A general state at the top level may be refined into concrete state variables and abstract states contained in lower level packages. This refinement into abstract states can continue until all have been refined into concrete state variables. Let us look at an example:

```
package Hierarchical_State_Demo
   with SPARK_Mode   => On,
        Abstract_State => Top_State,
        Initializes    => Top_State
is
   procedure Do_Something (Value   : in out Natural;
                           Success :    out Boolean)
      with Global  => (In_Out => Top_State),
           Depends => (Value     =>+ Top_State,
                       Success   => (Value, Top_State),
                       Top_State =>+ Value);
end Hierarchical_State_Demo;
```

In the body of package Hierarchical_State_Demo, we refine the state Top_State into the concrete state variable Count and the abstract state State (defined in the nested package A_Pack):

```
package body Hierarchical_State_Demo -- Nested implementation
   with SPARK_Mode => On,
        Refined_State => (Top_State => (Count, A_Pack.State))
```

```
is
   package A_Pack
      with Abstract_State => State,
           Initializes    => State
   is
      procedure A_Proc (Test : in out Natural)
         with Global  => (In_Out => State),
              Depends => (Test  =>+ State,
                          State =>+ Test);
   end A_Pack;
   ------------------------------------------
   package body A_Pack
      with Refined_State => (State => Total)
   is
      Total : Natural := 0;

      procedure A_Proc (Test : in out Natural)
         with Refined_Global  => (In_Out => Total),
              Refined_Depends => ((Test =>+ Total,
                                   Total =>+ Test)) is
      begin
         . . . .
      end A_Proc;
   end A_Pack;
   ------------------------------------------
   Count : Natural := 0;

   procedure Do_Something (Value   : in out Natural;
                           Success :    out Boolean)
      with Refined_Global  => (In_Out => (Count, A_Pack.State)),
           Refined_Depends => (Value       =>+ (Count, A_Pack.State),
                               Success     =>
(Value, Count, A_Pack.State),
                               Count       =>+ null,
                               A_Pack.State =>+ (Count, Value)) is
   begin
         . . . .
   end Do_Something;
end Hierarchical_State_Demo;
```

The Refined_Globals and Refined_Depends of procedure Do_Something also refer to the abstract state State. Notice that the Initializes => Top_State in the top level package specification is fulfilled by assignment of an initial value to Count and the Initializes => State aspect in the specification of the nested package

A_Pack. The body of package A_Pack refines the abstract state State into the concrete variable Total, which is used in the Refined_Global and Refined_Depends aspects of procedure A_Proc.

We could easily extend the logic illustrated by this example with additional abstract states in packages nested within the body of Hierarchical_State_ Demo (to widen our hierarchy) or with packages nested within the body of package A_Pack (to deepen our hierarchy). The obvious problem with these extensions is that the package body Hierarchical_State_Demo grows larger with each additional state abstraction.

A seemingly obvious answer to this problem is to put the nested package into an independent library unit and use a **with** clause to gain access to it. Here is a version of Hierarchical_State_Demo that does just that:

```
with Hierarchical_State_Demo.A_Pack;
package body Hierarchical_State_Demo  -- Child package implementation
   with SPARK_Mode => On,
        Refined_State => (Top_State => (Count, A_Pack.State))
is
   Count : Natural := 0;

   procedure Do_Something (Value   : in out Natural;
                           Success :    out Boolean)
     with Refined_Global  => (In_Out => (Count, A_Pack.State)),
          Refined_Depends => (Value         =>+ (Count, A_Pack.State),
                              Success       => (Value, Count, A_Pack.State),
                              Count         =>+ null,
                              A_Pack.State  =>+ (Count, Value)) is
   begin
         . . . .
   end Do_Something;
end Hierarchical_State_Demo;
```

Removing the nested package A_Pack has made this body much shorter. We made A_Pack a child of Hierarchical_State_Demo to align our package hierarchy with our abstract state hierarchy.

However, we run into difficulty when we attempt to examine the independent child package containing our new abstract state. SPARK requires that all refined states be hidden from clients – in our example, the clients of Hierarchical_State_Demo. This was not a problem when the package with the abstract state used in the refinement was nested in the body. But as an ordinary child package, it is accessible to all.

The solution is to make A_Pack a private child package. That way its resources are only accessible to its parent, Hierarchical_State_Demo, and its parent's descendants. Here is the specification for the private child package A_Pack:

```
private package Hierarchical_State_Demo.A_Pack
   with SPARK_Mode => On,
        Abstract_State => (State with Part_Of => Top_State),
         Initializes    => State
is
   procedure A_Proc (Test : in out Natural)
      with Global  => (In_Out => State),
           Depends => (Test   =>+ State,
                       State  =>+ Test);
end Hierarchical_State_Demo.A_Pack;
```

This private child package specification contains the option Part_Of. When refining to an abstract state defined in a private child package, this option must be included with that abstract state. The Part_Of option denotes the encapsulating state abstraction of which the declaration is a constituent. In our example, State is a constituent of Top_State. The reasons for requiring the Part_Of option when the abstract state is not "local" are given in section 7.2.6 of the SPARK *2014 Reference Manual* (SPARK Team, 2014a).

The body of package A_Pack requires no modification from the original given on page 121. Complete code for both the nested and private child versions of Hierarchical_State_Demo are available on our http://www.cambridge.org/us/academic/subjects/computer-science/programming-languages-and-applied-logic/building-high-integrity-applications-spark.

4.4 Default Initialization

One of the purposes of flow analysis is to ensure that no ineffective computations take place. However, Ada allows you to define default initializers for types that require it, a feature that ensures all objects of that type are initialized. If you attempt to re-initialize such an object to a nondefault value, you might wonder if that causes the default initialization to be ineffective.

To illustrate, consider the following declarations that introduce a type for arrays of ten integers:

```
subtype Array_Index_Type is Positive range 1 .. 10;
type    Integer_Array    is array (Array_Index_Type) of Integer;
```

Now consider a simple function that adds the elements of such an array. In this example, we only consider flow issues, so concerns about overflow during the computations can be set aside for now.

```
function Add_Elements (A : Integer_Array) return Integer is
   Sum : Integer := 0;  -- Ineffective initialization
begin
   Sum := 0;
   for Index in A'Range loop
      Sum := Sum + A (Index);
   end loop;
   return Sum;
end Add_Elements;
```

This function contains a redundant initialization of Sum, first when the variable is declared and then again later at the top of the function's body. Flow analysis flags the initialization as a flow issue saying that it has "no effect" as the value initialized is overwritten without being used.

This is fine, but now consider the case when Ada's facilities for specifying a default initial value for a type are used. You might introduce a special kind of integer to use as accumulators.

```
type Accumulator is new Integer
  with Default_Value => 0;
```

The use of the Default_Value aspect means that each time an object of type Accumulator is declared, it will automatically be given the value zero. This ensures accumulators are always initialized to a reasonable value. Now consider a second, somewhat different, implementation of Add_Elements:

```
function Add_Elements2 (A : Integer_Array) return Integer is
   Sum : Accumulator;
begin
   -- Sum is automatically initialized .
   -- Yet assignment below does not cause a flow issue .
   Sum := 1; -- Start with a bias value.
   for Index in A'Range loop
      Sum := Sum + Accumulator (A (Index)); -- note type conversion
   end loop;
   return Integer (Sum);
end Add_Elements2;
```

In this version, Sum is automatically initialized to its default value of zero. Ordinarily this would be a nice convenience for a function like this. However, Add_Elements2 wishes to bias the accumulated sum by one and thus re-initializes Sum to 1 in the body of the function. Significantly, this does not cause flow

analysis to flag the default initialization as ineffective. The tools understand that it is reasonable to override the default initialization in some circumstances and, thus, does not consider doing so a flow problem.

Because Accumulator is a new type distinct from Integer, it is now necessary to do some type conversions in Add_Elements2. You might wonder why not define Accumulator as a subtype of Integer. Alas, Ada does not allow the Default_Value aspect to be applied to a subtype.

As another example, consider the following record definition that wraps a pair of integers. Here default values are specified for both components.

```
type Pair_With_Default is
   record
      X : Integer := 0;
      Y : Integer := 0;
   end record;
```

Now consider a function Adjust_Pair that takes a Pair_With_Default and moves it on the (x, y) plane:

```
function Adjust_Pair (P : Pair_With_Default) return Pair_With_Default is
   Offset : Pair_With_Default;
begin
   -- Offset is automatically initialized.
   -- Yet assignment below does not cause a flow issue.
   Offset.Y := 1;
   return (P.X + Offset.X, P.Y + Offset.Y);
end Adjust_Pair;
```

As with the previous example, Offset is default initialized. Yet the assignment to Offset.Y is not a flow issue because the re-initialization of a default initialized value is considered reasonable and correct. Furthermore, the use of Offset.X in the return expression is not a flow issue because, of course, Offset has a default initializer for its X component.

You might wonder what happens if the record had default initializers for only some of its components:

```
type Pair_With_YDefault is
   record
      X : Integer;
      Y : Integer := 0;
   end record;
```

Although such a type declaration is legal in full Ada, it is not allowed in SPARK. Either all components must have default initializers or none of them can.

Variable packages can also be, in effect, default initialized when they are elaborated. Consider again the Datebook example, shown here with a procedure

that clears the datebook of all events. For brevity, the other subprograms in the package are not shown.

```
with Dates;
package Datebook
  with
     Abstract_State => State,
      Initializes   => State
is
   procedure Clear
     with
       Global  => (Output => State),
       Depends => (State => null);
end Datebook;
```

The Datebook package models a single object represented by the state abstraction State. The Initializes aspect asserts that the package initializes the state when it is elaborated. However, the Clear procedure also initializes the state as evidenced from its flow dependency contract, State => **null**. If a nervous programmer calls Clear before using the package, the programmer would then be re-initializing a default initialized "variable." Following the principle described earlier, flow analysis will not flag this as a flow issue.

In general, re-initialization may bring the entity being re-initialized to a new starting state. In the preceding example of Adjust_Pair Offset.Y is set to a starting value of 1 as per the needs of the enclosing subprogram. Similarly, a complex package might have several "clearing" procedures that clear the internal state in different ways.

Of course re-initialization entails some overhead. When a value is re-initialized, the time spent doing the default initialization was wasted. This is an argument against defining default initializations for all entities everywhere. Instead, the programmer may wish to explicitly initialize an object when, if, and how the programmer desires. However, default initialization is useful for certain types and certain packages; the SPARK tools allow re-initialization in those cases when it makes sense.

4.5 Synthesis of Dependency Contracts

Conceptually, SPARK requires every subprogram to have both data and flow dependency contracts. However, it is not necessary to explicitly include these contracts in all cases; the SPARK tools can synthesize them using rules discussed in this section.

You can write SPARK code without tediously providing data and flow dependency contracts everywhere provided you are content with the synthesized contracts. This means you can "convert" existing Ada code to SPARK by just adding an appropriate SPARK_Mode aspect to an existing compilation unit as we describe in Section 7.1.1, and, of course, removing any non-SPARK constructs the unit might be using. It is not necessary to annotate all subprograms with dependency contracts before starting to work with the SPARK tools. Applying SPARK to existing Ada in this way is called *retrospective analysis*.

Earlier we emphasized that the data and flow dependency contracts are part of the specification of a subprogram. Like the parameter list, the dependency contracts should, ideally, be written as part of your design process before you have implemented the subprogram. Applying SPARK to your code base as you design and implement is called *constructive analysis*.

Yet even in the constructive case, relying on synthesized contracts is sometimes reasonable. This is particularly true for subprograms that are nested inside other subprograms or that are used only in the context of a particular package. The internal subprograms exist to service the enclosing subprogram or package and are really part of the implementation of the enclosing entity and not design elements in themselves.

The precise rules for how contracts are synthesized are detailed, but the effect is largely intuitive:

1. If a Global aspect exists but a Depends aspect does not the flow dependency contract is synthesized by assuming each output depends on all inputs. For small subprograms such as the Search procedure on page 105, this is often exactly correct. In any case the resulting flow dependency is conservative in the sense that it might define more dependencies than actually exist. This has the potential to increase the number of false positives in the callers but does not, for example, allow any errors such as the accidental use of uninitialized values[2]. If false positives prove to be problematic, you can always add more precise flow dependency contracts explicitly.
2. If a Depends aspect exists but a Global aspect does not the data dependency contract is synthesized by examining the flow dependency contract and constructing the data dependencies from the flows. For example, suppose a subprogram is declared as follows:

```
procedure Do_Something(X : in Integer; Y : out Integer)
  with
    Depends => (Y => (X, A, B), B => (X, A), C => X);
```

This contract mentions A, B, and C, which must be visible as global variables at this point in the program. The flows specified require that A

be an input, C be an output, and B be both an input and an output. The synthesized data dependency contract is thus

```
with
   Global => (Input => A, In_Out => B, Output => C)
```

3. If neither a Global aspect nor a Depends aspect exists, the SPARK tools analyze the body of the subprogram to synthesize the data dependency contract and then use that to synthesize the flow dependency contract as described earlier.

In some cases, however, the body of the subprogram is not available, not analyzed, or imported from another language. One important example of this is when processing the specification of a library package with no source code for the body. In that case the SPARK tools assume the subprogram makes use of no global data at all, but a warning is produced to alert the programmer of this (possibly incorrect) assumption. The synthesized flow dependency contract then only considers the subprogram's parameters and, as usual, assumes each output depends on all inputs.

Functions are handled in largely the same way as procedures. The synthesized data dependency contract takes the function result as the only output of the function. It is possible to specify flows to the output of a function F by using F'Result explicitly in the flow dependency contract. However, writing explicit flow dependency contracts for functions is uncommon as the generated contracts are often exactly correct.

There are two other important issues to keep in mind with respect to dependency contract use and synthesis. First, it is permitted to declare more dependencies than actually exist. The SPARK tools will detect the inconsistency, but it is not considered an error. The diagonstic message is justifiable using pragma Annotate,[3] allowing you to "pre-declare" dependencies that you anticipate future versions of your code may need. Callers will thus be forced to consider such dependencies even if they are not currently active. For example, callers might be required to initialize certain global variables before calling your subprogram if you have declared those variables as Input data dependencies, even if your subprogram does not currently use them.

It is also important to understand that synthesized dependency contracts are not checked in the body of the subprogram to which they apply. The synthesized contracts are too aggressive in many cases, for example, declaring more flows than actually exist, resulting in many false positives. However, the synthesized dependency contracts are used in the analysis of calling code. The aggressive contracts might increase the number of false positives in callers as well, yet practice shows the burden of these false positives is usually minimal. In cases

where it matters, false positives can be avoided by making the dependency contracts explicit.

In the future, the SPARK tools will also be able to synthesize Abstract_State, Refined_State, Refined_Global, and Refined_Depends aspects. However, at the time of this writing, support for synthesizing these aspects is largely missing. For example, if you include Abstract_State in a package, you must explicitly provide the corresponding refined dependency contracts. On the other hand, at the time of this writing, the SPARK tools are able to synthesize dependency contracts of a subprogram in terms of the abstract state of packages being referenced by that subprogram.

Summary

- The development of SPARK programs can be split into three stages: verifying that the program is in the SPARK subset, verifying that the program has no flow errors, and verifying that the program is free from runtime errors and that all executable contracts are obeyed.
- The data dependency contract using the Global aspect specifies what global variables a subprogram uses and the modes on those variables. The three modes for global variables are equivalent to those for parameters.
- The data dependency contract treats global variables like additional parameters to the subprogram where the argument used at every call site is the same.
- The global variables used in data dependency contracts must be entire variables; they cannot be components of some larger variable.
- Functions can only input from global variables.
- Using global variables should be avoided or at least minimized. They are reasonable, however, when accessing local variables and parameters of an enclosing subprogram or when accessing the global state of a package from inside that package.
- Global constants initialized with a static expression do not need to be mentioned in the data dependency contract.
- When analyzing a subprogram with a data dependency contract, the SPARK tools verify that the modes declared on the global variables are consistent with the way those variables are actually used.
- The flow dependency contract using the Depends aspect specifies for each output of a subprogram which inputs the output depends on.
- Flow dependency contracts on a subprogram can, and ideally should, be specified before that subprogram is written; they form part of the subprogram's interface.

- Flow analysis verifies that no uninitialized data is used and that all computations are effective.
- Flow dependency contracts constrain the set of legal implementations beyond that required by the parameters alone.
- Flow dependency contracts are complete in the sense that flow analysis will fail unless all data flow information is fully in place. In contrast, pre- and postconditions are only as strong as the programmer makes them.
- It is possible to declare that an output of a subprogram depends on no inputs (and is thus being initialized by the subprogram) using **null** on the input side of the dependency.
- SPARK allows compilation units to be separately analyzed. Dependency contracts on called subprograms participate in the analysis of the calling subprogram.
- A state abstraction is a name representing global state inside a package (either global variables or the state of internally nested packages).
- Once declared, a state abstraction can be used as a kind of global variable for the purposes of data dependency and flow dependency contracts on the public subprograms of the package.
- An Initializes aspect can be used to specify that a package initializes its internal state without the help of a specific initialization procedure.
- Each state abstraction must be refined in the package body into its constituent global variables or nested package state.
- When refining state abstraction in a package body, the data dependency and flow dependency contracts on the public subprograms must also be refined in the body. They must be refined in terms of the state abstraction's constituents.
- A package can define multiple state abstractions to reduce false couplings between subprograms at the cost of exposing more details of the package's internal structure.
- When an entity (variable or variable package) is default initialized, it is not a flow error to re-initialize it. The default initialization is not marked as ineffective.
- The SPARK tools can synthesize data and flow dependency contracts for subprograms that are not explicitly marked as having them.
- If a subprogram has a Global aspect but no Depends aspect, the synthesized flow dependency contract takes each output as depending on all inputs.
- If a subprogram has a Depends aspect but no Global aspect, the synthesized data dependency contract is generated from the inputs and outputs listed in the flow dependency contract.
- If a subprogram has neither a Global aspect nor a Depends aspect, the synthesized data dependency contract is deduced from the body of the subprogram,

if available, and the synthesized flow dependency contract is computed from the data dependency contract as usual.
- The synthesized contracts are conservative. They may include flows that do not exist causing false positives.
- Without access to a subprogram's body, if no contracts exist, the SPARK tools cannot synthesize a data dependency contract and instead assume the subprogram does not use global data. A warning is issued to this effect.
- Dependencies are not checked in the body of subprograms that have synthesized contracts. This is done to avoid excessive numbers of false positive messages.
- The ability to synthesize contracts allows retrospective analysis where SPARK is used on existing code written without SPARK in mind. It also allows you to not bother writing dependency contracts on, for example, internal subprograms.

Exercises

4.1 Which of the following data dependency contracts are definitely illegal and why?

a.
```
function F(X : Integer) return Integer
  with
    Global => (Input => (Y, Z));
```

b.
```
function F(X : Integer) return Integer
  with
    Global => (Input => Y, In_Out => Z);
```

c.
```
procedure P(X : in out Integer)
  with
    Global => Dates.Compilation_Date;
```

d.
```
procedure P(X : in out Integer)
  with
    Global => (Input => Y, In_Out => (X, Z));
```

4.2 What is an "ineffective" computation?

4.3 Which of the following flow dependency contracts are definitely illegal and why?

a.
```
procedure P(X : in Integer; Y : in out Integer)
  with
    Depends => (Y => X);
```

b.
```
procedure P(X : in Integer; Y : in out Integer)
  with
    Depends => (Y =>+ X);
```

c.
```
procedure P(X : in Integer; Y : in out Integer)
  with
    Global  => (Output => (Count, Timestamp)),
    Depends => (Y =>+ X, Count => (X, Y));
```

d.
```
procedure P(X : in Integer; Y : in out Integer)
  with
    Global  => (Output => (Count, Timestamp)),
    Depends => (Y =>+ X,
                Count => (X, Y),
                Timestamp => Count);
```

e.
```
procedure P(X : in Integer; Y : in out Integer)
  with
    Global  => (Output => Timestamp, In_Out => Count),
    Depends => (Y =>+ X,
                Count => (X, Y),
                Timestamp => X);
```

f.
```
procedure P(X : in Integer; Y : in out Integer)
  with
    Global  => (Output => Timestamp, In_Out => Count),
    Depends => (Y =>+ X,
                Count =>+ (X, Y),
                Timestamp => X);
```

4.4 Modify the specifications of versions 1 and 2 of the bounded queue package in Chapter 3, pages 73 and 78, to include appropriate data dependency and flow dependency contracts.

4.5 Modify the specification and body of the Bingo basket package in Chapter 3, page 84, to include appropriate data dependency and flow dependency contracts. This package encapsulates the state of a Bingo basket.

4.6 Modify the specification and body of the serial number package in Chapter 3, page 86, to include appropriate data dependency and flow dependency contracts.

4.7 State abstractions expose, to a certain degree, the internal structure of a package and thus seem to partially break information hiding. Is this a problem? Discuss.

4.8 What does it mean to say an object is default initialized? Default initialization is never considered ineffective even if the object is later re-initialized. Why not?

4.9 Consider the following simple package with a global variable and a swap procedure that exchanges its parameter with the global variable:

```
package Example is
   G : Integer;

   procedure Swap(X : in out Integer)
     with Global => (In_Out => G);
end Example;
```

What flow dependency contract is synthesized for Swap? Is it accurate?

4.10 When a contract is synthesized, it is not checked in the body of the subprogram to which the contract applies. Consider the following subprogram with a flow dependency contract:

```
procedure P(X : in out Integer)
  with Depends => (X =>+ A, A => B);
```

a. What data dependency contract is synthesized?
b. If P's body, in fact, makes use of another global variable C, will the inconsistency be detected? Be careful!

4.11 When a contract is synthesized, it is not checked in the body of the subprogram to which the contract applies. Consider the following subprogram with a data dependency contract:

```
procedure P(X : in out Integer)
  with Global => (Input => A, Output => B);
```

a. What flow dependency contract is synthesized?
b. If P's body does not actually have one of the flow dependencies in the synthesized contract, will the inconsistency be detected?

4.12 We recommend explicitly declaring data and flow dependency contracts on the public subprograms of a package. Why?

4.13 When is it reasonable to rely on synthesized data and flow dependency contracts?

4.14 What is the difference between *constructive analysis* and *retrospective analysis*?