8

Software Engineering with SPARK

In the preceding chapters we have concentrated on the details of the SPARK language. In this chapter, we look at a broader picture of how SPARK might be used in the context of a software engineering process. The SPARK 2014 Toolset User's Guide (SPARK Team, 2014b) lists three common usage scenarios:

1. Conversion of existing software developed in SPARK 2005 to SPARK 2014
2. Analysis and/or conversion of legacy Ada software
3. Development of new SPARK 2014 code from scratch

We start by examining each of these scenarios in more detail, discussing the interplay between proof and testing, and then presenting a case study to illustrate some issues arising when developing new SPARK 2014 code from scratch.

8.1 Conversion of SPARK 2005

Converting a working SPARK 2005 program to SPARK 2014 makes sense when that program is still undergoing active maintenance for enhanced functionality. The larger language and the enhanced set of analysis tools provided by SPARK 2014 offer a potential savings in development time when adding functionality to an existing SPARK 2005 program.

As SPARK 2014 is a superset of SPARK 2005, the conversion is straight forward. Section 7.2 of the *SPARK 2014 Toolset User's Guide* (SPARK Team, 2014b) provides a short introduction to this conversion. Appendix A of the *SPARK 2014 Reference Manual* (SPARK Team, 2014a) has information and a wealth of examples for converting SPARK 2005 constructs to SPARK 2014. Explanations and examples are provided for converting subprograms, type (ADT) packages, variable (ASM) packages, external subsystems, proofs, and

more. Should you need to constrain your code to the SPARK 2005 constructs but wish to use the cleaner syntax of SPARK 2014, you may use **pragma** Restrictions (SPARK_05) to have the analysis tools flag SPARK 2014 constructs that are not available in SPARK 2005.

Dross et al. (2014) discuss their experiences with converting SPARK 2005 to SPARK 2014 in three different domains. AdaCore has a SPARK 2005 to SPARK 2014 translator to assist with the translation process. At the time of this writing, this tool is available only to those using the pro versions of their GNAT and SPARK products.

We illustrate a simple example of converting a SPARK 2005 package to SPARK 2014. The package encapsulates a circular buffer holding temperature data, for example, from an analog to digital converter. The SPARK 2005 specification is shown as follows:

```
with Data_Types;
--# inherit Data_Types;
package Temperature_Buffer05
  --# own Contents;
is
   type Temperature_Record is
      record
         Time_Stamp : Data_Types.Time_Type;
         Value      : Data_Types.Temperature_Type;
      end record;

   -- Adds a new item to the buffer.
   procedure Put (Item : in Temperature_Record);
   --# global in out Contents;
   --# derives Contents from Contents, Item;

   -- Returns True if buffer is not empty.
   function Has_More return Boolean;
   --# global in Contents;

   -- Retrieves the oldest item from the buffer.
   procedure Get (Item : out Temperature_Record);
   --# global in out Contents;
   --# derives Contents from Contents &
   --#         Item     from Contents;
   --# pre Has_More (Contents);

   -- Initializes the buffer.
```

```
   procedure Clear;
   --# global out Contents;
   --# derives Contents from ;

end Temperature_Buffer05;
```

The abstract state of this variable package is declared in SPARK 2005 by way of "own variables" in the specification. Annotations, in the form of Ada comments, provide the data and flow dependency contracts in a largely intuitive way. Of particular interest is the precondition on procedure Get that requires Has_More to return true. It is not permitted to get an item from the buffer if it is empty.

In SPARK 2005 the use of Has_More in the precondition is abstract because SPARK 2005 annotations are not executable. Any functions used in pre- and postconditions must be pure, therefore, SPARK 2005 requires that the abstract state read by the function be passed as an explicit parameter.

The body of the package starts by refining the abstract state and declaring the constituents of that state:

```
package body Temperature_Buffer05
  --#own Contents is Buffer, Count, Next_In, Next_Out;
is
   Buffer_Size  : constant := 8;
   type Buffer_Index_Type is mod Buffer_Size;
   type Buffer_Type is array (Buffer_Index_Type) of Temperature_Record;
   Buffer : Buffer_Type;

   type Buffer_Count_Type is range 0 .. Buffer_Size ;
   Count    : Buffer_Count_Type; -- Number of items.
   Next_In  : Buffer_Index_Type; -- Next available slot .
   Next_Out : Buffer_Index_Type; -- Next item to extract.
```

The use of a modular type for Buffer_Index_Type causes buffer indexes to wrap around automatically and simplifies the programming. In this case the buffer never fills; old data is pushed out as new data is entered.

The implementation of function Has_More and procedure Get follows:

```
   function Has_More return Boolean
   --# global in Count;
   is
   begin
      return Count > 0;
   end Has_More;
```

```
procedure Get (Item : out Temperature_Record)
--# global in Buffer; in out Count, Next_Out;
--# derives Item      from Next_Out, Buffer &
--#         Count     from Count            &
--#         Next_Out  from Next_Out;
is
begin
   Item := Buffer (Next_Out);
   Next_Out := Next_Out + 1;
   Count := Count - 1;
end Get;
```

As with SPARK 2014, the data and flow dependencies are refined in terms of their constituents. The code itself is straightforward. However, the SPARK 2005 tools have difficulty proving that the assignment statement Count := Count − 1 does not cause a runtime error. It does not because the precondition forbids Get from being called unless Count > 0. However, the SPARK 2005 tools do not understand the behavior of Has_More and need to be taught that behavior using techniques we do not detail here. This is necessary despite the fact that Has_More is implemented in the body of the package where, in principle, the tools can see it.

The following SPARK 2014 version of the package specification is a straightforward translation of the SPARK 2005 version:

```
with Data_Types;
package Temperature_Buffer14
  with
    SPARK_Mode => On,
    Abstract_State => Contents
is
   type Temperature_Record is
      record
         Time_Stamp : Data_Types.Time_Type;
         Value      : Data_Types.Temperature_Type;
      end record;

   -- Initializes the buffer.
   procedure Clear
     with
       Global  => (Output => Contents),
       Depends => (Contents => null);

   -- Adds a new item to the buffer.
```

```
   procedure Put (Item : in Temperature_Record)
     with
       Global  => (In_Out => Contents),
       Depends => (Contents =>+ Item);

   -- Returns True if buffer is not empty.
   function Has_More return Boolean
     with Global => (Input => Contents);

   -- Retrieves the oldest item from the buffer.
   procedure Get (Item : out Temperature_Record)
     with
       Global  => (In_Out => Contents),
       Depends => (Contents =>+ null, Item => Contents),
       Pre => Has_More;

end Temperature_Buffer14;
```

The body refines the abstract state using the Refined_State aspect. However, of particular interest is the implementation of Has_More and Get:

```
   function Has_More return Boolean is (Count > 0)
     with Refined_Global => (Input => Count);

   procedure Get (Item : out Temperature_Record)
     with
       Refined_Global => (Input => Buffer, In_Out => (Count, Next_Out)),
       Refined_Depends =>
         (Item      => (Next_Out, Buffer),
          Count     =>+ null,
          Next_Out =>+ null)
   is
   begin
      Item := Buffer (Next_Out);
      Next_Out := Next_Out + 1;
      Count := Count - 1;
   end Get;
```

The Has_More function is an expression function so its implementation is used to synthesize its postcondition. The SPARK 2014 tools do not need to have the meaning of Has_More explained separately. As a result, the SPARK 2014 tools prove procedure Get free of runtime errors without additional complications.

This example illustrates that the conversion of SPARK 2005 to SPARK 2014 may allow certain simplifications to be made in the code or in the proofs. It is not just a matter of translating SPARK 2005 annotations to SPARK 2014 aspects. In

addition, the SPARK 2014 tools use generally more powerful theorem provers; some verification conditions that were difficult with SPARK 2005 may be easier with SPARK 2014. As a result, it may be possible to simplify or eliminate certain loop invariants or write the code in the more natural way.

Finally, the richer language provided by SPARK 2014 offers the possibility of refactoring a SPARK 2005 program to take advantage of that richness. The Temperature_Buffer example presented earlier comes from an embedded system that also buffers other kinds of readings. It would be natural to make all of the buffers instances of a single generic unit. However, earlier versions of SPARK did not support generics. Of course, how much rewriting of a legacy system is appropriate will depend on the situation. However, if it is deemed worthwhile to update the software to SPARK 2014, it may also be worthwhile to reorganize and even redesign parts of the software to take advantage of SPARK 2014's larger set of features.

8.2 Legacy Ada Software

The preferred use of SPARK is in a *constructive analysis* style in which we create a program whose units contain a full set of contracts specified by the aspects discussed in Chapters 4 and 6. As we saw in Chapter 7, SPARK provides the means to mix SPARK code with non-SPARK code through the use of contracts on the specifications of non-SPARK code. Commonly, the non-SPARK code is written in Ada or C.

SPARK 2014 has the capability to analyze preexisting Ada code where no SPARK contracts are given. This *retrospective analysis* is made possible by the ability of the SPARK tools to synthesize a set of data dependency contracts and flow dependency contracts directly from the source code of a program unit (see Section 4.5). These synthesized contracts can be used to analyze legacy Ada code or during the early development of SPARK code in which contracts have not yet been included.

The synthesized contracts are safe over-approximations of the real contracts. For example, they assume that *all* outputs are dependent on *all* inputs. Because it is unlikely that the body of the subprogram meets all of the synthesized contracts, the SPARK tools do not generate warnings or checks when the body does not respect the synthesized contracts. The synthesized contracts are used to verify proper initialization and respect of any dependency contracts in the callers of the subprogram.

Section 6.8.1 of the *SPARK 2014 User's Guide* describes how contracts are synthesized for

- a non-SPARK subprogram – one with Spark_Mode => Off;
- a SPARK subprogram with no data or flow contracts;

- a SPARK subprogram with only data contracts; and
- a SPARK subprogram with only flow contracts.

As mentioned in the previous paragraph, the aspects that the SPARK tools synthesize are almost always more general than those we would write. However, when no data or flow contracts are given on a SPARK subprogram, the tools generate precise data and flow dependencies by using path-sensitive flow analysis to track data flows in the subprogram body. These synthesized contracts accurately describe the code whether or not that code is correct.

Section 7.3 of the SPARK*2014 Toolset User's Guide* (SPARK Team, 2014b) provides some guidance to using the SPARK tools to analyze legacy Ada code as a first step prior to performing a full or partial conversion to SPARK. Even without the intention of converting legacy Ada code to SPARK, the tools can help us find errors in our Ada code. Take, for example, the following contrived package specification and body:

```
package Unused_Parameter with SPARK_Mode => On is

   procedure Avg (A : in  Natural;
                  B : in  Natural;
                  C : out Natural);
end Unused_Parameter;

package body Unused_Parameter with SPARK_Mode => On is

   procedure Avg (A : in  Natural;
                  B : in  Natural;
                  C : out Natural) is
   begin
      C := (A + A) / 2;
   end Avg;

end Unused_Parameter;
```

Here we made a mistake in the body by typing (A + A) rather than (A + B). The SPARK Examine tool reports that B is not used. This analysis is not provided by the GNAT compiler. Here is another simple package:

```
package Overflow with SPARK_Mode => On is

   procedure Avg (A : in  Natural;
                  B : in  Natural;
                  C : out Natural);
end Overflow;
```

```
package body Overflow with SPARK_Mode => On is

   procedure Avg  (A : in  Natural;
                   B : in  Natural;
                   C : out Natural)  is
   begin
      C := (A + B) / 2;
   end Avg;

end Overflow;
```

This time the Examine tool reports no problems. However, the Prove tool reports that the line

```
C := (A + B) / 2;
```

may overflow. It is possible that the values of A and B are so large that their sum exceeds that of the largest value the accumulator can hold. This error is still present in implementations of the binary search and merge sort algorithms in many introductory programming books even after being pointed out by Pattis in 1988. While one probably would not think to test this case, our proof tool was quick to discover it. With a little algebra, we can reorganize the expression to

```
C := A + (B - A) / 2;
```

Now the Prove tool reports no possible runtime error.

Let us look at a larger example of using SPARK to analyze preexisting Ada code. We would like to verify that no runtime errors will be raised in the package Bingo_Basket that we presented in Section 3.4. Here is its specification with the addition of the aspect to make it in SPARK:

```
with Bingo_Numbers; use Bingo_Numbers;
package Bingo_Basket with SPARK_Mode => On is

   function Empty return Boolean;

   procedure Load   -- Load all the Bingo numbers into the basket
      with
         Post => not Empty;

   procedure Draw (Letter  : out Bingo_Letter;
                   Number : out Callable_Number)
```

```
   -- Draw a random number from the basket
      with
         Pre => not Empty;

end Bingo_Basket;
```

When we run the SPARK Examine tool we see the following messages concerning the package body (code on page 84):

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
bingo_basket.adb:34:07: warning: no Global contract available for "Reset"
bingo_basket.adb:34:07: warning: assuming "Reset" has no effect on global items
bingo_basket.adb:36:26: warning: no Global contract available for "Random"
bingo_basket.adb:36:26: warning: assuming "Random" has no effect on global items
```

It is obvious that the Examine tool is not happy with our use of the generic random value generator from the Ada library. Our solution here is to factor out the code involving operations from the library and put it into a private child package. We mark the specification as in SPARK and the body as not in SPARK:

```
private package Bingo_Basket.Random with Spark_Mode => On is
   function Random_Number return Callable_Number;
end Bingo_Basket.Random;

with Ada.Numerics.Discrete_Random;
package body Bingo_Basket.Random with SPARK_Mode => Off is

   package Random_Bingo is new Ada.Numerics.Discrete_Random
                                 (Result_Subtype => Callable_Number);

   -- The following object holds the state of a random Bingo number generator
   Bingo_Gen : Random_Bingo.Generator;

   function Random_Number return Callable_Number is
   begin
      return Random_Bingo.Random (Gen => Bingo_Gen);
   end Random_Number;

begin
   -- Initialize the random number generator from the system clock
   Random_Bingo.Reset (Gen => Bingo_Gen);
end Bingo_Basket.Random;
```

Here is the revised body of package Bingo_Basket with all Ada library usage done in calls to the child package:

```
with Bingo_Basket.Random;
package body Bingo_Basket with SPARK_Mode => On is

   type Number_Array is array (Callable_Number) of Callable_Number;
   The_Basket : Number_Array; -- A sequence of numbers in the basket
   The_Count  : Bingo_Number; -- The count of numbers in the basket

   procedure Swap (X : in out Callable_Number;
                   Y : in out Callable_Number) is
      Temp : Callable_Number;
   begin
      Temp := X; X := Y;  Y := Temp;
   end Swap;

   function Empty return Boolean is (The_Count = 0);

   procedure Load is
      Random_Index : Callable_Number;
   begin
      -- Put all numbers into the basket (in order)
      for Number in Callable_Number loop
         The_Basket (Number) := Number;
      end loop;
      -- Randomize the array of numbers
      for Index in Callable_Number loop
         Random_Index := Bingo_Basket.Random.Random_Number;
         Swap (X => The_Basket (Index),
               Y => The_Basket (Random_Index));
      end loop;
      The_Count := Callable_Number'Last;  -- all numbers now in the basket
   end Load;

   procedure Draw (Letter  : out Bingo_Letter;
                   Number : out Callable_Number) is
   begin
      Number    := The_Basket (The_Count);
      The_Count := The_Count - 1;

      -- Determine the letter using the subtypes in Bingo_Definitions
      case Number is
```

```
            when B_Range => Letter := B;
            when I_Range => Letter := I;
            when N_Range => Letter := N;
            when G_Range => Letter := G;
            when O_Range => Letter := O;
         end case;
      end Draw;
   end Bingo_Basket;
```

With all of the code that uses the Ada library moved out to a child package, the SPARK Examine tool finds no problems with this package. Nor does the proof tool report any problems. We are now confident that package Bingo_Basket has no runtime errors and that procedure Load's postcondition that the basket is not empty always holds.[1] We were tempted to add Abstract_State and Initializes aspects to the specification of Bingo_Basket.Random. However, our goal here is the analysis of legacy Ada code, not the conversion of Ada to SPARK.

When analyzing legacy code, we suggest beginning with the lowest level packages in your program – those that do not depend on other units. Add SPARK_Mode => On to each package specification and run the SPARK examine tool on it to see a listing of potential errors. Correct any errors reported by the examination. Then add SPARK_Mode => On to the body and examine it. As we did with package Bingo_Basket, you will see warnings for any standard Ada library operations you use. To obtain a deeper analysis, move such operations to their own package whose body is not in SPARK. You can then repeat the process for the next level of program units.

8.3 Creating New Software

SPARK is best applied to the development of new software. Starting from scratch provides the opportunity to make use of the SPARK tools from the development of the architectural design through the detailed implementation.

Test Driven Development (TDD) is a popular, evolutionary approach to software development that combines test-first development (in which you write a test before you write just enough code to fulfill that test) and refactoring (a disciplined restructuring of code). With SPARK's ability to combine proof with testing, we can extend TDD to an even more rigorous approach we call *Verification Driven Development* (VDD).[2] With VDD, we write contracts before we write the code to fulfill them. Data dependency and flow dependency contracts can be verified early on. Verification of freedom from runtime errors and of pre- and postconditions can be done once the code is written. Designing to meet your verification goals is a powerful approach to creating quality software.

Software design is all about the decisions a software engineer makes between the gathering and verification of requirements and the creation of code to implement those requirements. A design is the specification of a group of software artifacts or modules.

Many different design methodologies have been devised to guide us in making the many decisions involved. Section 7.1 of the *SPARK 2014 Toolset User's Guide* (SPARK Team, 2014b) gives the following overall view of developing SPARK programs from scratch:

1. Begin by creating a set of package specifications that describe the architectural design of the system. Include contracts with each specification that describe the abstract state encapsulated by each package. Use subprogram contracts to specify global dependencies on the abstract state and dependency contracts to specify information flow in each subprogram. Preconditions and postconditions may be added to these high-level packages to describe high-level properties such as safety and security.
2. Identify the SPARK packages with the SPARK_Mode aspect. At this stage the high-level package structure can be analyzed with the Examine tool before any executable code is implemented.
3. Implement the package bodies making use of top-down decomposition. Start with the top-level subprogram specifications and implement the bodies by breaking them down into lower-level subprograms, each with appropriate contracts. You can continuously run the Examine tool during each iteration of this process.
4. As each subprogram is implemented, you can verify it by proof or testing. Testing contracts with assertion checking enabled provides us with confidence that our contracts are written correctly. Proof then shows absence of runtime errors and that the contracts are met.
5. Once verification is complete, the executable can be compiled with assertion checks either enabled or disabled depending on the policy chosen by the project.

In the following sections we look at a more detailed approach to designing SPARK programs. The INFORMED^3 design method was developed especially for applying the strengths of SPARK during this crucial stage. This introduction to INFORMED is based on the technical report *INFORMED Design Method for SPARK* (SPARK Team, 2011). Chapter 13 *SPARK: The Proven Approach to High Integrity Software* (Barnes, 2012) provides another discussion of the INFORMED method. Although the examples in both the technical report (SPARK Team, 2011) and book (Barnes, 2012) are SPARK 2005, the principles apply equally well to SPARK 2014.

8.3.1 *Design Principles*

The properties of a good design are reasonably well known:

Abstraction is the separation of the essential features of an entity from the details of how those features actually work. It is our major tool for dealing with complexity.

Encapsulation is the inclusion of one entity within another entity so that the included entity is not apparent. The concept of class, in which data and operations are combined to form a single component, is a prime example of encapsulation in object-oriented design. Encapsulation provides a clear separation between specification and implementation – a necessary tenet of the contract model of programming. The package is SPARK's major construct for encapsulation.

Information hiding is the principle of segregation of the design decisions in a system that are most likely to change. It protects other parts of the program from extensive modification when the design decision is changed. Information hiding is closely related to abstraction as it is the details of how things work that we want to hide. Hiding unnecessary details allows us to focus on the essential properties of an entity.

Information hiding is related to but different than encapsulation. Encapsulation puts things into a box. Whether that box is opaque or clear determines whether the information is hidden or not. The private type is SPARK's major construct for information hiding.

Coupling is a measure of the connections between entities. Highly coupled objects interact in ways that make it difficult to modify one object without modifying the other. High levels of coupling may be a result of poor abstractions or inadequate encapsulation. Weak or loose coupling is desirable.

In SPARK, the appearance of a package name in a **with** clause represents loose coupling (use of a service). The appearance of a package name in a data dependency contract (`global` aspect) or a flow dependency contract (`depends` aspect) indicates stronger coupling.

Cohesion is a measure of focus or purpose within a single entity. It is the degree to which the elements of a module belong together. Having high cohesion in our modules makes it easier for us to understand them and make changes to them. Having unrelated abstract state variables in a SPARK package is an indicator of low cohesion.

Hiding unnecessary details allows us to focus on the essential properties of an entity. However, the state of an entity is an essential property that should not

be hidden – we cannot reason in the absence of state information. The state of an object is defined by its implementation. Because inspecting the implementation of an object to ascertain state information breaks the contract model, we must include this information in the object's specification. We accomplish this task by including Abstract_State aspects. These aspects provide the appropriate level of abstraction of state information that is needed to reason about the effect of operations on an object without having or needing the details of how the state is represented. Managing state information in this way is an important piece of the INFORMED design method.

8.3.2 *Design Elements*

INFORMED uses concepts from both object-oriented design (OOD) and functional design. OOD techniques are used to establish the architecture of the system. This architecture is expressed as a main program and framework of packages with contracts. Of particular importance is the assignment of state to these packages. With this information, we can use the SPARK tools to analyze the data dependency and flow dependency at an early stage.

We use classic functional decomposition to implement the operations within our objects. We can again make use of SPARK tools to check that the desired properties of our design are being maintained, our code is free of runtime errors, and any functional properties expressed by postconditions are met.

Main Programs

Main programs frequently have a form similar to the following program. The system is initialized and then a loop (often infinite) does all of the processing required.

```
with A, B, C;
procedure Main
   with Spark_Mode => On,
        Global     => -- references to states in packages A, B, and C,
        Depends    => -- references to states in packages A, B, and C
is
   procedure Initialize   is ...
   procedure Do_Something is ...
begin
    Initialize;
   loop
       Do_Something;
   end loop;
end Main;
```

The Initialize and Do_Something procedures are likely decomposed into several procedures, each responsible for some individual mode of the system's behavior. For example, one Do_Something procedure may be responsible for controlling the temperature in a vessel while another is responsible for controlling the pressure.

The most important parts of this generalized main program are the two aspects that specify the data dependencies and flow dependencies of the system. These aspects generally refer to the abstract state of the packages that are withed rather than to specific variables.

Packages

In Chapter 3 we organized packages into four groups: (1) definition packages, (2) utility packages, (3) type packages, and (4) variable packages. INFORMED adds a few more to this classification scheme.

Variable Packages. Contain states that should be revealed through an Abstract_State aspect. We use the name of the abstract state in the Global and Depends aspects of our main program and/or other units that use the package. Encapsulation and information hiding are maintained because no details of the internal state need be revealed. Refined_State allows us to specify a number of more detailed state items within the package body.

We may compose variable packages. Thus, a bicycle object could contain a frame object and two wheel objects. The abstract state of the bicycle may be refined into the abstract states of its components. SPARK private child packages allow the logical nesting or embedding of variable packages without the need to physically embed the packages that represent them.

Type Packages. Do not have state and therefore do not have Abstract_State aspects. As described in Section 3.3, a type package provides the name of a type that may be used in the declaration of objects in other units. This type may be a concrete type (as illustrated by the first version of our bounded queue on page 73) or a private type (as illustrated by the second version of our bounded queue on page 78).

Type packages are used to implement abstract data types. For private types, the package must also provide a set of operations that may be performed on objects of that type. Concrete types come with their own predefined set of operations, which may be extended by the declaration of additional operations with parameters of that type.[4]

Variables of the types declared in type packages are declared at the point of use and passed as parameters to the operations provided by the type package.

Because this localizes state at the point of variable declaration, type packages provide a mechanism for the reduction of information flow and, hence, coupling. The INFORMED report uses the phrase "instance of a type package" for a variable of a type declared in a type package.

INFORMED describes a more specialized type package called a *type package declaring concrete types*. This form of type package is equivalent to the definition package we described in Section 3.1.

Utility Packages. Provide shared services to other packages. These packages never contain state (otherwise they would be variable packages). Any types declared in a utility package should be simple concrete types rather than abstract data types (otherwise they would be type packages). The INFORMED report provides examples of where utility packages are appropriate and where they are not.

Boundary Variable Packages. Are a special kind of variable package that provide interfaces between our system and the elements outside of it. The entity to which our SPARK program communicates might be some kind of hardware sensor or actuator. Section 7.3 discusses the nature of external subsystems found commonly in embedded applications.

These external subsystems make use of external variables that represent streams of data arriving from or being sent to the external system so they are characterized by a **with** External in their abstract state aspects.

Boundary Variable Abstraction Packages. Are used to place an abstraction layer between the external variables of a system and their users. This approach eliminates direct exposure of any external variables, shielding the higher level units from the details of the external variables. Boundary variable abstraction packages use SPARK's state refinements to provide this indirect access to the external variables.

The abstraction may hide the fact that more than one external variable is involved in providing the inputs or may hide some other processing that is taking place. The abstraction may also hide other local state (such as previous values) that are not external variables. The INFORMED technical report provides an example of using a single boundary variable abstraction package to encapsulate two different buttons that may be pressed by a user. This package is refined into two boundary variable packages implemented as private child packages. Accompanying this example in the INFORMED report is a very important guideline concerning boundary variable abstraction packages – never mix input and output external variables in a single abstraction. This mixing leads to

confusing information flow results where inputs incorrectly appear to depend on values previously sent as outputs. Use a separate abstraction for each external variable as was done in package Serial_Port in Section 7.3.2.

Finally, we note that a variable abstraction package need not include any hidden external variables. The external entity to which our SPARK program communicates might be an API of some library, operating system, or cooperating software system.

8.3.3 *Principles of the INFORMED Design Approach*

The INFORMED technical report lists five guiding design principles.

1. Application-Oriented Aspects

SPARK aspects provide an expression of the behavior of the software independently of the actual code. This description is more useful if it is expressed in problem domain terms rather than in implementation terms. For example, we prefer to see the terms *fuel valve* rather than digital-to-analog converter #3.

2. Minimal Information Flow

To reason about the behavior of a system, we need to reason about the information that flows through it. Such reasoning is simplified if the information flows are minimized. Moving data from one part of the system to another increases the information flow complexity as measured by the Global and Depends aspects. Methods for minimizing information flow include

- Minimizing propagation of unnecessary details;
- Localizing and encapsulating of state information;
- Avoiding making copies of data; and
- Using an appropriate hierarchy of packages.

3. Clear Separation of the Essential from the Inessential

Software designers have to reconcile many, sometimes conflicting, constraints. When such conflicts arise, the designer should make the essential functionality of the system the highest priority.

For example, although it might ease the testing of a system by making certain data global, it is preferred that this inessential aspect of the design be located in the place that ensures minimal information flow. Additional code, clearly identified by comments and flow analysis as not being necessary for the essential functionality of the system, can provide access to the data at test time.

4. Careful Selection of the SPARK Boundary

Careful thought should be given to defining the boundary between what is in SPARK and what is not. This boundary is far more fluid with SPARK 2014 with its goal of supporting verification through a combination of proof and testing than it is with SPARK 2005.

5. Use Static Analysis Early

The packages making up the design should be examined as early as possible with the SPARK tools. Early use of these tools is facilitated by using abstractions, deferring implementation details, and making appropriate use of `Spark_Mode => Off`. The analysis should continue as the design evolves. This constant checking of design choices provides assurance that our aims are met.

8.3.4 *INFORMED Design Steps*

The INFORMED report includes a suggested sequence of six steps for constructing a SPARK application.

1. Identification of the system boundary, inputs, and outputs.
2. Identification of the SPARK boundary within the overall system boundary.
3. Identification and localization of system state.
4. Handling of the initialization of state.
5. Handling of secondary requirements.
6. Implementing the internal behavior of components.

Design is an iterative process so there is usually considerable looping, backtracking, and feedback among the steps within the following steps.

1. *Identification of the system boundary, inputs, and outputs*

The SPARK system being designed is typically part of a larger system that has interactions with the outside world. The first step is to delineate this boundary and identify the physical inputs and outputs. These are the environmental quantities that impact or are impacted by the system's behavior. They are described as *monitored* and *controlled* variables in the Four-Variable Model of Parnas and Madey (1995), which describes the interactions between a computer system and the environment. This model is illustrated in Figure 8.1.

2. *Identification of the SPARK boundary*

After identifying the boundary of our overall system, we give thought as to where to place the boundary of the SPARK portions of our application – that is, what parts are in SPARK and what parts are not. Selection of the boundary

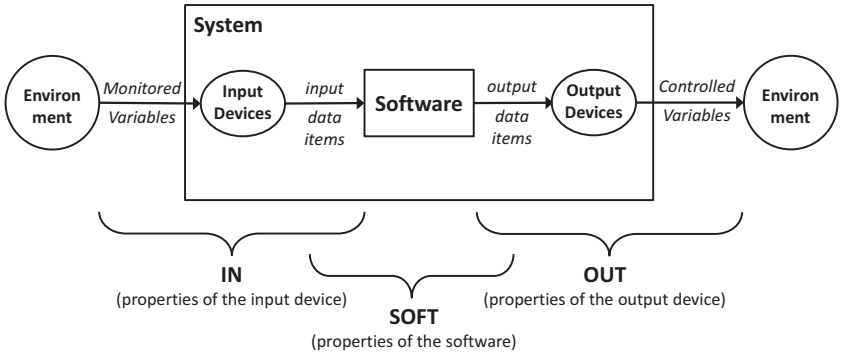


Figure 8.1. The four-variable model (adapted from Parnas and Madey, 1995).

variable packages and boundary variable abstract packages is one way to define the SPARK boundary. The input and output values provided by the abstract state of these packages are the input data items and output data items of the Parnas model illustrated in Figure 8.1. Figure 8.2 shows the flow of information through these packages to and from the non-SPARK software and environment defined by the Parnas four-variable model. The INFORMED report gives examples of systems that contain multiple SPARK systems within an overall system.

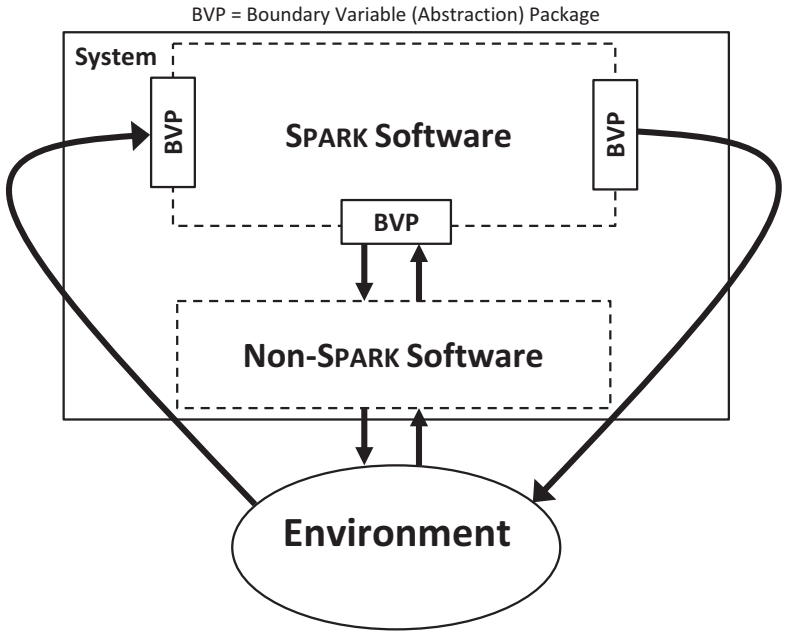


Figure 8.2. Boundary variable packages define the SPARK boundary.

The identification of the styles of verification that will be applied to each component identified at this point is a crucial part of VDD. Proof will certainly play a large role in the verification of SPARK units, and runtime assertion checking will help us verify Ada units. Units written in other languages will need to be tested. We must determine the depth of analysis required of each unit. The recommendations given in Section 8.4 provide guidance in planning the verification at the boundaries between SPARK and non-SPARK units.

3. *Identification and localization of state*

The identification of state that is required in our system and determining where to place that state information are central to meeting the INFORMED design method's goals. As such, the report categorizes the different ways that packages can be used for the management, concealment, and communication of state.

Most nontrivial systems store values in variables and therefore have "history" or state. In the absences of state, calling an operation with a particular set of values always returns the same answer. On the other hand, the result of calling an operation that uses state may also depend on some complex history of all previous calls of that procedure. The INFORMED report emphasizes that the selection of appropriate locations for this state is probably the single most important decision that influences the amount of information flow in the system. You must decide (1) what must be stored, (2) where should it be stored, and (3) how should it be stored.

What must be Stored? Some static data storage is almost always required, but the amount should be minimized as far as possible. You should avoid duplicating data by storing it in a single place and make it available to other units through "accessor functions."

Classification of state is a crucial part of INFORMED. States may be classified into groups such as essential, inessential, input, output, and so on. The *golden rule of refinement* tells us to only refine concrete states as constituents of the same abstract state if they are of the same classification and initialization mode.

Integrity levels for safety (level A, B, etc.) or security (classified, unclassified, top-secret, etc.) are good candidates for state classes. The golden rule of refinement is especially important here – never refine together states that are of different integrities. If you have separation properties, such as making sure that top-secret and unclassified material are kept separate, you need to design that separation into your state hierarchy.[5]

Where should it be Stored? The INFORMED reports gives us several guidelines. Data stored inside the main program does not appear in its data flow

or information flow dependency contracts. This has the unfortunate effect of removing all of the information describing the flow of data. Therefore, it is more appropriate to place this data within variable packages or boundary variable packages. The main program then includes the abstract states of these packages in its global and depends aspects. Of course, within the packages we will refine the abstract states into combinations of concrete state variables and abstract state variables of lower level packages with state.

How should it be Stored? The INFORMED report describes a number of ways to store state:

- In a variable package at library level (global state)
- In a variable package embedded within another variable package (hierarchical state refinement)
- In a variable package that is a private child of another variable package (the preferred approach to hierarchical state refinement)
- In a variable package embedded within the main program
- As an instance of type defined in a type package
- As a concrete Ada variable

State should be localized as much as possible. It should be avoided entirely where a local variable within a subprogram will suffice. Variables declared in subprograms (other than the main program) exist only for the life of a call to that subprogram. Type packages give us extra freedom in to locate items with complex state as locally as possible.

We advise that when you need only a single instance of an object with state that you use a variable package and use type packages when multiple copies are required. The use of variable packages can be extended to those situations with a small finite number of objects. For example, it might be better to encapsulate three buttons within a single control panel package rather than to define a button type package.

4. *Handling of the initialization of state*

After identifying and determining the location of states, our next consideration is on how the various states will be initialized. There are two approaches to initialize state:

a. Initializing during program elaboration. This initialization includes the assignment of initial values accompanying variable declarations and the execution of statements in a package's elaboration part (see Section 3.4.1).

   We discussed default initialization in Section 4.4. The use of the Initializes aspect provides an important piece of information to the SPARK

tools to ensure that states within a variable package are not used prior to their initialization.

b. Initializing during program execution. The main program can assign values directly to concrete Ada variables and call initialization procedures to initialize the state of variable packages or objects declared from types in type packages.

5. *Handling of secondary requirements*

The third principle of the INFORMED design approach is to have a clear separation of the essential and inessential requirements. INFORMED defines *secondary* requirements as those that are not derived from the core functional requirements. Secondary does not mean unimportant, but rather that they should be accommodated in ways that do not distort the "purity" of design through state and information flow. Read-only variables, data logging and test points, and caching are three examples of secondary requirements discussed in Apppendix B of the INFORMED report (Spark Team, 2011).

6. *Implementation of the internal behavior of components*

After identifying the components of our architecture such as variable packages, type packages, and boundary variable packages, we can create specifications with contracts. We can perform early static analysis on these specifications.

Next, we need to implement the desired behavior of each object. The first step, as usual, is to decompose this behavior into smaller INFORMED components. For example, a variable package for a control panel might be decomposed into a number of boundary variable packages – one for each component on the panel. When we have taken decomposition as far as we can, we can use a standard top-down refinement process to decomposed behaviors into appropriate subprograms. We can include contracts with each subprogram specification and continue to run the Examine tool before implementing the bodies. As the bodies of the subprograms within a particular package are completed, we can run the Proof tool to check for potential runtime errors or postcondition violations.

The INFORMED report (Spark Team, 2011) includes three case studies to illustrate the design method. Although these designs are in Spark 2005, the ideas translate easily to Spark 2014. We illustrate this approach with a security-related case study in Section 8.5.

8.4 Proof and Testing

As we discussed in Section 1.1.2, testing is the primary means for verifying and validating software. Testing can only show us that the program is "correct"

for the cases actually tested. A good tester picks the most appropriate test cases to discover faults. Industry standards such as DO-178 (RTCA, 2011a) and ISO/IEC/IEEE 29119 provide guidance on appropriate testing techniques.

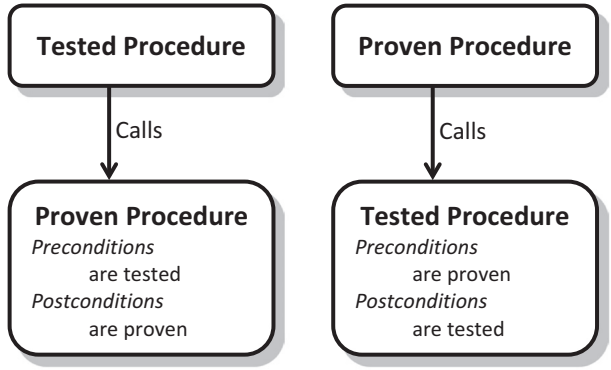
In Chapter 6 we looked at how to use SPARK to prove correctness properties of programs. If we can prove that our program is correct, then we have total confidence that it will work in all cases covered by the assertions. Formal verification gives stronger guarantees than testing. New verification standards provide guidance on using proof. For example, The DO-333 standard, a supplement on formal methods for the DO-178C standard, states, "Formal methods might be used in a very selective manner to partially address a small set of objectives, or might be the primary source of evidence for the satisfaction of many of the objectives concerned with development and verification" (RTCA, 2011b).

In addition to providing stronger guarantees, the cost of proof can be less than the cost of testing. This is particularly true for the most critical software for which coverage criteria require a lot of testing (Moy et al., 2013). A clear example of savings was seen in the use of SPARK for the mission computer of the C130J aircraft. Lockheed reported an 80 percent savings over their projected cost for testing and certification of more than 100,000 lines of code (Amey, 2002).

However, there are times when proof is not the best option:

- Our entire program is not amenable to proof. Parts of our program may need to use Ada constructs such as exceptions or access types that are not legal in SPARK. Parts of our programs may be written in another language such a C (see Section 7.2).
- It may be very expensive or even impossible to provide formal descriptions of all the desired properties of a package. Interactive packages are particularly difficult to formalize.
- Software validation[6] is not specifically addressed by formal contracts. Testing remains the best way to validate software.

By combining proof and testing, we get the best of both worlds. We can combine methods by dividing our application into pieces (packages or subprograms) and applying either proof or testing to each separate piece. As the verification chain is only as strong as its weakest link, our goal is to obtain verification at least as good as can be accomplished with testing alone, but at a lower cost. Comar, Kanig, and Moy (2012) provide an excellent discussion of combining formal verification and testing. This paper is available through AdaCore's GEM series.



a. When a tested subprogram calls a proven subprogram

b. When a proven subprogram calls a tested subprogram

Figure 8.3. Verifying preconditions and postconditions when combining proof and testing.

SPARK's contracts provides an ideal mechanism for combining proof with testing in such a divided program. How we verify the contracts depends on the relationship between the subprograms in our application:

- When a proven subprogram calls another proven subprogram, the preconditions and postconditions of the called subprogram are verified by proof.
- When a tested subprogram calls another tested subprogram, the preconditions and postconditions of the called subprogram need to be verified by test.
- When a tested subprogram calls a proven subprogram, the preconditions of the proven subprogram need to be verified by test and the postconditions of the proven subprogram are verified by proof (see Figure 8.3a). Calls from different tested subprograms to the same proven subprogram must have the proven subprogram's preconditions tested separately.
- When a proven subprogram calls a tested subprogram, the preconditions of the tested subprogram are verified by proof and the postconditions of the tested subprogram need to be verified by test (see Figure 8.3b).

The 80/20 ratio so common in software engineering seems to fit the model of combined proof and testing. About 80 percent of the subprograms in a typical high integrity program can be "easily" proven, and 80 percent of the remaining subprograms can be "easily" tested. That leaves 4 percent of the subprograms in the difficult-to-verify category.

By setting a switch in our Ada 2012 compiler, the preconditions and postconditions we have written in our subprograms will be checked at runtime.

This simplifies our testing work. We need only develop and run an appropriate set of test cases. An exception is raised whenever an assertion check fails.

The GNATtest tool in combination with the GNAT specific aspect Test_Case may be used to build and run a complete testing harness. This tool is based on AUnit, the Ada version of the xUNIT family of unit test frameworks.

Finally, it is worth saying that irrespective of how you intend to verify your operations, testing can help identify potential issues with proofs and proof can help identify potential issues with tests. If you cannot prove a postcondition, try testing it. You may find that the error is in the contract, not in the code.

8.5 Case Study: Time Stamp Server

In this section we outline a realistic case study to illustrate the use of SPARK in the construction of new software. The latest source for this case study, with additional documentation, can be found on GitHub (Chapin, 2014).

Our application is a secure time stamp server, which we call *Thumper*, implementing the time stamp protocol specified in RFC-3161 (Adams et al., 2001). For the sake of brevity, certain details of the protocol are not fully supported. However, the implementation is complete enough to demonstrate many features of SPARK. In this section we describe the protocol, sketch the architecture of Thumper, and discuss the role SPARK plays in its implementation.

8.5.1 *Time Stamp Protocol*

The time stamp protocol is simple in concept. It provides a way for some person, such as Alice, to obtain a small, cryptographic token that proves a particular document in her possession existed on or before a specific time. Alice can present this token to another person, Bob, for later verification.

More concretely, suppose Alice is a student at a university and Bob is an instructor. Bob requires that a certain assignment be completed by a certain date and time. Alice finishes the assignment and obtains the time stamp token. She then submits her work to Bob via e-mail. Unfortunately, because of some network problem, her message does not arrive in Bob's mailbox until after the due date. Alice can then use the time stamp to prove to Bob that she did, in fact, complete the assignment on time.

The time stamp protocol makes use of three cryptographic concepts that we briefly review here. For more information, consult any textbook on cryptography such as Mao (2004) or Stallings (2014).

A *cryptographic hash* is a relatively small value that can be used to represent a larger document. If a change is made to the document, the hash will, with very

high likelihood, be different. A cryptographic hash is similar to a checksum except that it is computationally infeasible for anyone to find a document that generates a particular hash value or modify a document in such a way as to generate the same hash value. When a high quality cryptographic hash algorithm is used, it is also computationally infeasible to find two different documents that generate the same hash value, a property sometimes called *strong collision resistance*.

A *public/private key pair* is a related pair of keys such that material encrypted by the public key can only be decrypted by the corresponding private key. Typically, a user keeps the private key secret but distributes the public key widely allowing people the user does not know to send encrypted messages that only the user can read.

A *digital signature* is a unit of data attached to a message obtained by processing a cryptographic hash of that message with the signer's private key. The signature can be verified by anyone using the corresponding public key. A successful verification shows both that the message was unmodified, because any modification would (with high likelihood) change the hash, and that the signer is authentic because only the owner of the private key can make a signature that is verifiable with the corresponding public key.

Here is how Alice obtains a time stamp for one of her documents using a time stamp server:

1. Alice computes a cryptographic hash H of her document.
2. Alice sends H to the server.
3. The server appends the current time to H and then digitally signs the combination. The result is the time stamp.
4. The server returns the time stamp to Alice.
5. Alice verifies the time stamp by checking the time it contains is reasonable, the hash it contains is still H, and the digital signature on the time stamp is valid.

Later Alice submits her document together with its time stamp to Bob. Bob computes the hash H of the document and verifies that it agrees with the hash in the time stamp. He also verifies the server's digital signature. If Bob believes the server has not been compromised and trusts the server to have an accurate time, he must agree that Alice's document existed at or before the time mentioned in the time stamp.[7]

Even though Alice holds the time stamp, she cannot defeat this protocol. Modifying her document after obtaining the time stamp will change its hash. If she tries to tamper with the time stamp itself to change either the hash or time stored in it, she will invalidate the server's signature. Notice that Alice does

not need to reveal her document to the server (only a hash of it), authenticate to the server, or trust the server in any way because she verifies the time stamp when she receives it.

8.5.2 *Architecture, Design, and Implementation*

Clearly, both Alice and Bob would make use of a specialized client program to simplify the handling of the time stamps and the required cryptographic operations. However, here we are only concerned with the time stamp server, Thumper.

During the design and implementation of Thumper we will make two important security-related assumptions. We assume all files stored on the host file system are private to Thumper. This includes especially Thumper's private key. Furthermore, we assume the time returned by the host operating system is correct. Ensuring these assumptions is a problem of system administration rather than of software design and implementation.

Thumper is conceptually simple. It waits for a time stamp request message from the network, computes the desired time stamp, and returns it. Thumper then waits for the next request message. Invalid request messages cause Thumper to generate an error response. However, RFC-3161 has many requirements for precisely how messages are to be formatted. In particular the Distinguished Encoding Rules (DER) of Abstract Syntax Notation 1 (ASN.1) (International Telecommunication Union, 2002) are used.

RFC-3161 is not prescriptive about how messages are to be transported. Thumper uses the User Datagram Protocol (UDP). This is reasonable because both the request and the response are small enough to fit into a single UDP datagram. Thus, there are no concerns about ordering multiple packets or about acknowledgments. The response serves as the acknowledgment to the request. Furthermore, Thumper is an iterative server that processes only one request at a time. This simplifies the implementation and is reasonable because the time required to process a request is small. Also, clients can in no way slow down the operation of the server by, for example, refusing to send required information after the initial request is made.

Thumper makes use of three major supporting services:

- The network
- A cryptography library
- A serial number generator

Thumper abstracts these services into their own packages. The network package provides subprograms for sending and receiving UDP datagrams. We

use a non-SPARK Ada library provided by the compiler vendor, GNAT.Sockets. The cryptography package provides a subprogram for making digital signatures. It is built on top of a C library provided by a third party, OpenSSL (OpenSSL Project, 2014a).

RFC-3161 requires that each time stamp created by the server be marked with an integer serial number. Furthermore, the serial number must be unique over the life of the server's deployment; even if the server is shut down and restarted it cannot use serial numbers from previous runs. We implement this requirement by generating serial numbers randomly using a pseudo-random number generator with a large period (2^{64}), seeded by the server's boot time. Although this admits the possibility of creating two time stamps with the same serial number, the probability of doing so is vanishingly small. The serial number generator package makes use of the Ada standard library.

In Section 8.3.4 we presented the six steps of the INFORMED design process. Here we walk through those steps showing how they can be applied to Thumper:

1. *Identification of the system boundary, inputs, and outputs*

The simple structure of Thumper makes this step relatively easy. The server as a whole reads messages from and writes messages to the network. It must also read its private key from the host file system. Finally, we identify a secondary requirement of writing a log file where messages about errors encountered can be stored.

It is possible that a future version of Thumper may have other inputs and outputs. For example, we considered providing remote management through a web interface or the ability to store generated time stamps in a database for auditing purposes. However, security sensitive applications such as Thumper benefit from being as simple as possible. Every input increases the attack surface of the system and makes maintaining security more difficult. Consequently, the current version of Thumper does not support these additional features. For similar reasons, all of Thumper's "configurable" parameters are hard coded; no configuration file is used.

2. *Identification of the SPARK boundary*

The INFORMED design method emphasizes the importance of clearly delineating the boundary between the SPARK and non-SPARK sections of the system. Deciding when to not use SPARK is as important as deciding when to use it. The first step in identifying this boundary is to identify your *verification goals*. These goals represent what you wish to accomplish by using SPARK, and they guide your decision about how to partition your program into SPARK and

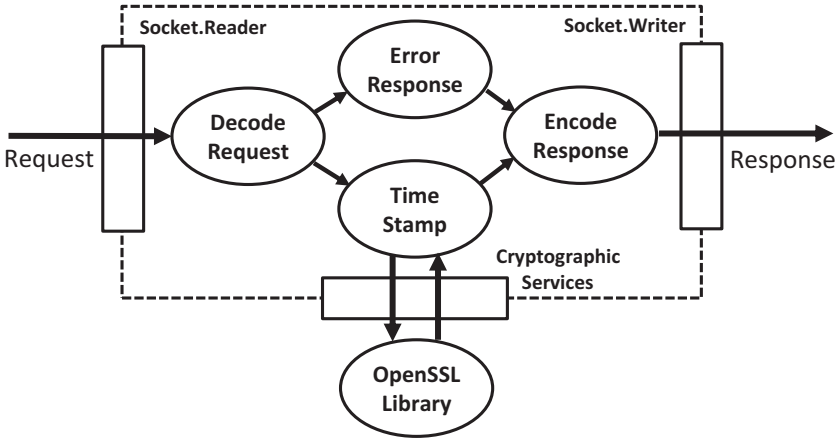


Figure 8.4. Architecture of the SPARK components of Thumper.

non-SPARK sections. In the case of Thumper we identify our verification goals as ensuring freedom from runtime error when processing client input along with ensuring that an invalid or inappropriate time stamp is never returned to the client.

From a security point of view, the critical path through Thumper is that traversed by the messages received from the client on their way to becoming responses sent back to the client. We assume input messages may be malicious and purposely malformed with the intent of crashing the server or coaxing the server to generate an invalid time stamp. In contrast, Thumper's initialization and secondary logging feature are not critical because they do not depend on client input and can thus be reasonably written in full Ada.

Figure 8.4 shows the architecture of the SPARK components of Thumper. The boundary between the SPARK and non-SPARK code is marked by several boundary variable packages that encapsulate the connection to the non-SPARK components. The packages Network.Socket.Reader and Network.Socket.Writer wrap the compiler vendor's network library and provide simplified interfaces. They also hide the underlying network library to prevent vendor specific names from leaking into the rest of the program. A future version of Thumper may use a different networking library or even a different transport mechanism for messages; the SPARK components do not need to know about such changes.

For pragmatic reasons we choose to use the OpenSSL library for Thumper's cryptographic needs. This is not ideal and is particularly worrisome in light of OpenSSL's history of security problems (OpenSSL Project, 2014b). Unfortunately, at the time of this writing no open source cryptographic library supporting digital signature algorithms and written in verified SPARK is available

to us. Because OpenSSL is written in C and lies outside of SPARK, we use a boundary variable package Cryptographic_Services that allows SPARK to access OpenSSL's services. It hides all the required C interfacing as well as Thumper's private key in its (non-SPARK) body.

SPARK is used in Thumper, as shown in Figure 8.4, to decode the incoming request messages, check their validity, and encode either an error response or a time stamp response as appropriate. Because of the complexity involved in processing DER encoded ASN.1 messages, there are plenty of opportunities for errors in this process, and we endeavor to show that no runtime exceptions can be raised by Thumper's code.

3. *Identification and localization of state*

The boundary variable packages Network.Socket.Reader and Network.Socket.Writer contain external state abstractions that represent the flow of messages from and to the network, respectively.

The specification of the Reader package is as follows:

```
pragma SPARK_Mode(On);

with Messages;
with Network.Addresses;

package Network.Socket.Reader
  with Abstract_State =>
    (Input_Message_Stream with External =>(Async_Writers   => True,
                                           Effective_Reads => False))
is
   type Status_Type is (Success, Failure );

   -- This procedure receives a datagram.
   -- It also returns the source address.
   procedure Receive (Message : out Messages.Network_Message;
                      From    : out Addresses.UDPv4;
                      Status  : out Status_Type)
     with
       Global  => (Input => Input_Message_Stream),
       Depends => ((Message, From, Status) => Input_Message_Stream);
end Network.Socket.Reader;
```

The Messages package is a utility package written in SPARK that provides basic data types and few helper subprograms for manipulating bounded messages of raw data. The package Network.Addresses is a type package written in SPARK

that provides an abstract type for UDP addresses. Notice that although the body of this package is implemented in terms of a vendor-provided library, no evidence of that fact is visible here.

The stream of messages coming from the network is modeled using an external state abstraction Input_Message_Stream. The external state abstraction has Async_Writers set to True because messages can appear at any time for reasons outside of Thumper's control. In particular, multiple calls to Receive in succession might return different results. On the other hand, Effective_Reads is set to False because reading a message has no influence on the external system sending them. This is particularly true in the case of the UDP protocol where there is no connection between the sender and receiver of a datagram.

The specification of the Writer package is as follows:

```
pragma SPARK_Mode(On);

with Messages;
with Network.Addresses;

package Network.Socket.Writer
  with Abstract_State =>
    (Output_Message_Stream with External =>(Async_Readers   => True,
                                            Effective_Writes => True))
is
   type Status_Type is (Success, Failure );

   -- This procedure sends a datagram to the given destination address.
   procedure Send (Message : in  Messages.Network_Message;
                   To      : in  Addresses.UDPv4;
                   Status  : out Status_Type)
     with
       Global  => (In_Out => Output_Message_Stream),
       Depends =>
         (Output_Message_Stream =>+ (Message, To),
          Status => (Output_Message_Stream, Message));
end Network.Socket.Writer;
```

Here the stream of outgoing messages is modeled using another external state abstraction Output_Message_Stream. In this case, Async_Readers is set to True indicating that some external system is reading the outgoing messages at a time unrelated to Thumper's activities. Also, Effective_Writes is set to True, indicating that every message potentially influences the external system reading them.

Some of the flow dependences on the Send procedure are not immediately obvious. For example, why should the output message stream depend on itself?

This arises because it is possible, in principle, for the output message stream to be temporarily unusable as a result of a filled network queue in the underlying operating system. In that case, sending may fail and the output message stream's state would not be changed. The output message stream returned by the procedure thus depends on the output message stream given to the procedure.

It is important to remember that the contracts used with these boundary variable packages are not verified by SPARK because the bodies of these packages are outside of SPARK and not analyzed by the tools. Careful review of these contracts is thus required if SPARK is to have a proper understanding of the real data and information flows involved.

Following the guideline in the INFORMED report (SPARK Team, 2011), we do not use a single state abstraction to model both input and output streams simultaneously. Using a single state abstraction would confuse analysis. For example, the SPARK tools might conclude that the messages read from the network are somehow related to the messages sent to the network.

There are two variable packages containing state to consider. The first is the pseudorandom number generator used to create serial numbers. Its specification is as follows:

```
pragma SPARK_Mode(On);

package Serial_Generator
  with
     Abstract_State => State,
     Initializes    => State
is
   type Serial_Number_Type is mod 2**64;

   procedure Next (Number : out Serial_Number_Type)
     with
        Global  => (In_Out => State),
        Depends => ((State, Number) => State);

end Serial_Generator ;
```

The second package containing state is the cryptographic services package that holds the server's private key used to make digital signatures. Its specification is as follows:

```
pragma SPARK_Mode(On);

with Hermes;
```

```
package Cryptographic_Services
   with
      Abstract_State => Key
is
   type Status_Type is (Success, Bad_Key);

   procedure Initialize (Status : out Status_Type)
      with
         Global  => (Output => Key),
         Depends => ((Key, Status) => null);

   function Make_Signature (Data : in Hermes.Octet_Array)
                            return Hermes.Octet_Array
      with Global => (Input => Key);

end Cryptographic_Services;
```

Here, Hermes is a package, entirely written in SPARK, containing ASN.1 encoding and decoding facilities. The type Octet_Array in that package holds raw data intended to be part of a DER encoded ASN.1 data stream.

The body of Cryptographic_Services contains all the necessary code to interface to the OpenSSL library, but the use of that library is not visible here. If at some future time a suitable SPARK cryptographic library becomes available, only the body of Cryptographic_Services would need to be updated to use it.

We note that the state held by the two packages described here have significantly different security levels. The key held by Cryptographic_Services is obviously security sensitive information. In contrast, the state of the pseudo-random number generator is much less sensitive because it is only used to derive serial numbers on response messages. It is thus appropriate to store these states in different variable packages rather than, for example, creating a single package to hold all the state in one place.

4. *Handling of the initialization of state*

The previously described state is initialized in different ways. The network packages are children of Network.Socket, a non-SPARK package that encapsulates a single UDP socket. The specification of Network.Socket is as follows:

```
with Network.Addresses;
private with GNAT.Sockets;
package Network.Socket is
```

```
   Network_Error : exception;

   procedure Create_And_Bind_Socket (Port : in Addresses.Port_Type);

private
   Socket : GNAT.Sockets.Socket_Type;
end Network.Socket;
```

The procedure Create_And_Bind_Socket initializes the two network streams. It reports failures by way of raising the Network_Error exception, a translation of the vendor specific exception GNAT.Socket_Error. The variable Socket declared in the private section of this package is visible to the (non-SPARK) bodies of the Reader and Writer packages. Here we use a parent package to provide private resources to the two child packages. In reality both message streams use the same underlying socket, yet this organization makes it possible to abstract the streams into different packages as we described earlier.

The Serial_Generator package is initialized at elaboration time, as evidenced by the Initializes aspect on its specification. It uses the system time at start-up to seed the pseudorandom number generator. The Cryptographic_Services package is initialized by the main program by way of a special initialization procedure. It reads the private key from the file system and stores it internally for future use.

5. *Handling of secondary requirements*

At this time Thumper has only one secondary requirement: to create a log of any external errors that arise during operation. We are not concerned with initialization errors. If Thumper fails to initialize, it terminates at once with an appropriate message; such errors do not need to be logged. However, there might be problems receiving or sending on the network and such problems should be recorded to alert the operator and assist with troubleshooting.

The logger is another boundary variable package, not shown in Figure 8.4, with a specification as follows:

```
pragma SPARK_Mode(On);
package Logger
   with
      Abstract_State =>
         (Log_Stream with External => (Async_Readers    => True,
                                       Effective_Writes => True)),
       Initializes   => Log_Stream
is
```

```
   procedure Write_Error (Message : in String)
      with Global => (Output => Log_Stream);

end Logger;
```

At elaboration time this package initializes itself by opening a suitable log file.

6. *Implementation of the internal behavior of components*

What remains, of course, is implementing the bodies of the packages mentioned so far along with the core logic of the program. Here we describe a few of highlights of the implementation.

Thumper's main program is not in SPARK. It is mostly concerned with initialization and dealing with any initialization errors. It then calls a procedure Service_Clients where the core functionality of the system resides. The main program, in its entirety, follows:

```
with Ada.Exceptions;
with Ada.Text_IO;

with Cryptographic_Services;
with Network.Socket;
with SPARK_Boundary;

use Ada.Exceptions;

procedure Thumper_Server is
   use type Cryptographic_Services.Status_Type;

   Crypto_Status : Cryptographic_Services.Status_Type;
begin
   -- Be sure the key is available.
   -- This initializes Cryptographic_Services.Key
   Cryptographic_Services.Initialize (Crypto_Status);
   if Crypto_Status /= Cryptographic_Services.Success then
      Ada.Text_IO.Put_Line
        ("*** Unable to initialize the cryptographic library: missing key?");
   else
      -- Set up the socket.
      -- This initializes the network streams (both input and output).
      Network.Socket.Create_And_Bind_Socket (318);

      SPARK_Boundary.Service_Clients; -- Infinite loop services requests
   end if;
```

```
exception
   when Ex : Network.Socket.Network_Error =>
      Ada.Text_IO.Put_Line
        ("*** Unable to  initialize  network: " & Exception_Message(Ex));
end Thumper_Server;
```

The package SPARK_Boundary corresponds to the dashed box in Figure 8.4. The Service_Clients procedure executes the material inside that box in an infinite loop, making use of the (boundary) variable packages we described earlier. The specification of SPARK_Boundary is shown as follows:

```
pragma SPARK_Mode(On);

with Cryptographic_Services;
with Logger;
with Network.Socket.Reader;
with Network.Socket.Writer;
with Serial_Generator;

use Network.Socket;

package SPARK_Boundary is

   procedure Service_Clients
     with
       Global =>
         (Input  =>
           (Reader.Input_Message_Stream, Cryptographic_Services.Key),
          In_Out =>
           (Logger.Log_Stream,
            Writer.Output_Message_Stream,
            Serial_Generator.State)),
       Depends =>
         (Logger.Log_Stream =>+
            (Reader.Input_Message_Stream, Writer.Output_Message_Stream,
             Cryptographic_Services.Key, Serial_Generator.State),
          Writer.Output_Message_Stream =>+
            (Reader.Input_Message_Stream,
             Cryptographic_Services.Key,
             Serial_Generator.State),
          Serial_Generator.State =>+ Reader.Input_Message_Stream);

end SPARK_Boundary;
```

Notice that the **with** statements exactly mention those packages that are either boundary variable packages or packages containing state. These **with** statements are necessary so that the various abstract states can be used in the data and flow dependency contracts on Service_Clients .

The presence of the logger complicates the dependency contracts. This is unfortunate because the logger is a secondary requirement, and one might prefer that it did not clutter the essential information. Unfortunately, the SPARK tools require that flows to (and, in general from) the logger's state be properly declared as for any other flows.

Some of the flows are surprising. In particular the dependency of the output message stream on itself. This arises because Service_Clients ultimately calls the Send procedure in the Writer package. As explained previously, this procedure might leave the output message stream unchanged. In addition if Service_Clients only gets errors when trying to receive a request, it will never even attempt to write to the output message stream. Thus the output message stream must be an In_Out global item. These details were not fully appreciated when the contracts on Service_Clients were first written. Some adjustments were made during the implementation of Service_Clients as these issues came to light.

It is interesting to note that the flow dependency contract helps with security review. Because attackers can manipulate the input message stream, any items that depend on that stream need to be carefully secured. In this case all outputs are potentially attackable. Of particular interest is the log stream. Is it possible for an attacker to send a malformed request in such a way as to cause a malformed log message to crash the server? This question is particularly interesting in light of the fact that the body of package Logger is not in SPARK. Similarly, could an attacker somehow manipulate the state of the serial number generator by sending appropriately malformed requests?

The Service_Clients procedure itself contains an infinite loop as shown:

```
procedure Service_Clients is
   use type Reader.Status_Type;
   use type Writer.Status_Type;

   Client_Address    : Network.Addresses.UDPv4;

   Network_Request   : Messages.Network_Message;
   Request_Message   : Messages.Message;
   Read_Status       : Reader.Status_Type;

   Response_Message  : Messages.Message;
   Network_Response  : Messages.Network_Message;
```

```
   Write_Status       : Writer.Status_Type;
begin
   loop
      Reader.Receive (Message => Network_Request,
                      From    => Client_Address,
                      Status  => Read_Status);

      if Read_Status /= Reader.Success then
         Logger.Write_Error("Failure reading request message!");
      else
         Request_Message := Messages.From_Network (Network_Request);
         Timestamp_Maker.Create_Timestamp (Request_Message, Response_Message);
         Network_Response := Messages.To_Network (Response_Message);

         Writer.Send (Message => Network_Response,
                      To      => Client_Address,
                      Status  => Write_Status);

         if Write_Status /= Writer.Success then
            Logger.Write_Error("Failure sending reply message!");
         end if;
      end if;
   end loop;
end Service_Clients;
```

The variables Network_Request and Network_Response hold raw octets of type Network.Octet received from and being sent to the network. They are records that carry a fixed size array along with a count of the number of elements in that array actually being used. The variables Request_Message and Response_Message hold raw octets of type Hermes.Octet that contain DER encoded ASN.1 data.

The functions From_Network and To_Network only do type conversions on the data from Network.Octet to Hermes.Octet and vice versa. Although it may seem pedantic to distinguish between different kinds of octets, it makes sense because data on the underlying network may have many forms. Hermes works only with DER encoded ASN.1. But in the future, time stamp messages may be encapsulated in some application protocol such as HTTP. So distinguishing between raw network data and the time stamp messages would be useful in a future version of Thumper.

The procedure Create_Timestamp converts request messages into response messages. Every request generates a response; invalid or malformed requests generate error responses. Create_Timestamp is the heart of the application and is almost purely functional. It accepts one array of octets and returns another.

However, because each generated time stamp is digitally signed and must include a unique serial number, Create_Timestamp needs to use the cryptographic key and the serial number generator. The SPARK declaration of Create_Timestamp is as follows:

```
procedure Create_Timestamp (Request_Message : in  Messages.Message;
                            Response_Message : out Messages.Message)
  with
    Global =>
      (Input  => Cryptographic_Services.Key,
       In_Out => Serial_Generator.State),
    Depends =>
      (Response_Message =>
        (Request_Message, Cryptographic_Services.Key, Serial_Generator.State),
       Serial_Generator.State =>+ null);
```

The bulk of the program is actually inside Create_Timestamp. However, the high degree of purity of the procedure – it rarely references any state – means that its implementation is, in principle, straightforward. We refer you to the Thumper site on GitHub (Chapin, 2014) for the details.

Alternate Designs

It is useful to consider some alternate designs to understand how the concepts we have developed so far apply to them. First, consider the network boundary variable packages Network.Socket.Reader and Network.Socket.Writer. These two packages encapsulate external state abstractions for input from and, respectively, output to the network. For the sake of accurate flow analysis, it is important that differen state abstractions be used for inputs and outputs. Yet creating two separate packages is not strictly necessary even if two state abstractions are to be used. It would be possible to define a single package with two state abstractions. Such an approach might seem more natural particularly in light of the fact that both state abstractions use a common underlying socket.

However, we anticipate that a future version of Thumper may wish to store generated time stamps for auditing. Such a change could be made by replacing the Network.Socket.Writer package body with a different body that also communicates with a database. This change would not affect the Network.Socket.Reader package at all. Two separate packages may seem like overkill, but it does clearly distinguish the inputs and outputs of the system, allowing one to be changed without interfering with the other.

The use of variable packages to hold the key and pseudorandom number generator state could also be changed. In particular, Cryptographic_Services and Serial_Generator could be converted to type packages that provided suitable

abstract data types to hold their state. Objects of those types could be declared in the main program and passed to Service_Clients as parameters. However, this does not simplify the dependency contracts on Service_Clients very much. Cryptographic_Services.Key and Serial_Generator.State would no longer be global items, but they would still appear, as parameter names, in the flow dependency contract. Also, Thumper needs neither multiple cryptographic keys nor multiple pseudorandom number generators so the main advantage of using type packages is not required.

Summary

- The conversion of a SPARK 2005 program to SPARK 2014 is straight forward as SPARK 2014 is a superset of SPARK 2005.
- The SPARK tools may be used to perform retrospective analyses of existing Ada code where no SPARK aspects are given.
- The SPARK tools will generate safe over-approximations of data dependency contracts and flow dependency contracts directly from the source code of a program unit.
- More in-depth analyses of legacy Ada code may require moving non-SPARK code segments into child or private child packages.
- Designing to meet your verification goals is a powerful approach to creating quality software. This approach to creating new SPARK programs is called verification driven development.
- Abstraction, encapsulation, information hiding, coupling, and cohesion provide measures of good software design.
- The INFORMED design method was developed especially for applying the strengths of SPARK during this crucial stage of software development.
- The elements of an INFORMED design include variable packages, type packages, utility packages, boundary variable packages, and boundary variable abstraction packages.
- We use Abstract_State aspects to make it known that a package encapsulates state without revealing the implementation details of that state.
- Localizing state and minimizing data flow are major tenets of the INFORMED design method.
- Combining proof and testing is a powerful mechanism for verifying code. As the verification chain is only as strong as its weakest link, our goal is to obtain verification at least as good as can be accomplished with testing alone, but at a lower cost.