9

Advanced Techniques

In this chapter we examine some advanced techniques for proving properties of SPARK programs. Although the approaches we describe here will not be needed for the development of many programs, you may find them useful or even necessary for handling larger, realistic applications.

9.1 Ghost Entities

Ghost entities make it easier to express assertions about a program. The essential property of ghost entities is that they have no effect on the execution behavior of a valid program. Thus, a valid program that includes ghost entities will execute the same with or without them.

9.1.1 *Ghost Functions*

In applications where you are trying to prove strong statements about the correctness of your programs, the expressions you need to write in assertions become very complex. To help manage that complexity, it is desirable to factor certain subexpressions into separate functions, both to document them and to facilitate reuse of them in multiple assertions.

Functions that you create for verification purposes only are called *ghost functions*. The essential property of ghost functions is that they do not normally play any role in the execution of your program.[1] Ghost functions may only be called from assertions such as preconditions, postconditions, and loop invariants. They may not be called from the ordinary, non-assertive portions of your program.

As an example consider the specification of a package Sorted_Arrays that contains subprograms for creating and processing sorted arrays of integers:

```
package Sorted_Arrays is
   type Integer_Array is array( Positive range <>) of Integer;

   -- Sorts the given array.
   procedure Sort (Data : in out Integer_Array )
     with Global => null,
          Post   => (for all J in Data'First .. Data'Last - 1 =>
                       (Data(J) <= Data(J + 1)));

   -- Return index of specified Value if it exists; zero if Value not in Data.
   function Binary_Search (Data : in Integer_Array;
                           Value : in Integer ) return Natural
     with Pre => (for all J in Data'First .. Data'Last - 1 =>
                    (Data(J) <= Data(J + 1)));
end Sorted_Arrays;
```

Notice that the postcondition of Sort and the precondition of Binary_Search both use the same quantified expression to assert that the array being processed is sorted. Although the expression is not exceptionally unwieldy in this case, it is still somewhat obscure and hard to read. Having it duplicated on two subprograms also hurts the package's maintainability.

Our second version of this specification introduces a ghost function to abstract and simplify the pre- and postconditions on the other subprograms. We use the Boolean aspect Ghost to indicate that the function Is_Sorted is included only for verification purposes.

```
package Sorted_Arrays2 is
   type Integer_Array is array( Positive range <>) of Integer;

   function Is_Sorted (Data : in Integer_Array ) return Boolean
     with Ghost => True,
          Post  => (Is_Sorted'Result =
                      (for all J in Data'First .. Data'Last - 1 =>
                         (Data(J) <= Data(J + 1))));

   -- Sorts the given array.
   procedure Sort (Data : in out Integer_Array )
     with Global => null,
          Post   => Is_Sorted (Data);
```

```
   -- Return index of specified Value if it exists; zero if Value not in Data.
   function Binary_Search (Data  : in Integer_Array;
                           Value : in Integer) return Natural
     with Pre => Is_Sorted (Data);
end Sorted_Arrays2;
```

The postcondition on Sort and the precondition on Binary_Search use the Is_Sorted function rather than the lengthier quantified predicates. They are now clearer and easier to maintain.

The function Is_Sorted is decorated with a postcondition that explains its effect. One might be tempted to use a conditional expression in the postcondition as follows:

```
function Is_Sorted (Data : Integer_Array) return Boolean
  with Ghost => True,
       Post  => (if Is_Sorted'Result then
                   (for all J in Data'First .. Data'Last - 1 =>
                      (Data(J) <= Data(J + 1))));
```

However, this postcondition is not strong enough. It says nothing about the array if Is_Sorted returns False. In particular, an implementation of Is_Sorted that returns False would always satisfy this postcondition (see the implication entry in Table 5.1). What is required is for Is_Sorted to return true "if and only if" the array is sorted. This equivalence is expressed in our first version with the equality of Is_Sorted'Result and the qualified expression. To satisfy the postcondition, it is necessary for the truth value of both those things to be the same.

It is not strictly necessary to include a postcondition on Is_Sorted at all. The SPARK tools will understand that the array coming out of Sort passes Is_Sorted, and this is all that is required to send that array to Binary_Search. Exactly what Is_Sorted does is of no immediate concern.

However, exposing the details of what Is_Sorted does might allow the SPARK tools to complete other proofs elsewhere in the program more easily. For example, with the postcondition on Is_Sorted visible, the tools will "know" that the array coming out of Sort has, for example, the property

```
Data(Data'First) <= Data(Data'Last)
```

as a result of the transitivity of <=. In general it is good practice to expose as much information as feasible to the SPARK tools.

Here is a body for the ghost function Is_Sorted:

```
function Is_Sorted (Data : in Integer_Array) return Boolean is
begin
   return (for all J in Data'First .. Data'Last - 1 =>
              (Data(J) <= Data(J + 1)));
end Is_Sorted;
```

In this case, the body is essentially a duplication of the postcondition. This is not unusual for ghost functions although it is not required or even universal. It does mean if the assertion policy is Check the test done by Is_Sorted is essentially done twice. The test is done in the body to compute Is_Sorted'Result and then again in the postcondition. However, if the assertion policy is Ignore, assertions are not executed and the entire Is_Sorted function could be removed from the program by the compiler because it can never be used outside assertion expressions anyway.

A better alternative in this case is to write the Is_Sorted function as an expression function as follows:

```
package Sorted_Arrays3 is
   type Integer_Array is array(Positive range <>) of Integer;

   function Is_Sorted (Data : in Integer_Array) return Boolean
   is (for all J in Data'First .. Data'Last - 1 =>
         (Data(J) <= Data(J + 1)))
     with Ghost => True;

   -- Sorts the given array.
   procedure Sort (Data : in out Integer_Array)
     with Global => null,
          Post   => Is_Sorted (Data);

   -- Return index of specified Value if it exists; zero if Value not in Data.
   function Binary_Search (Data  : in Integer_Array;
                           Value : in Integer) return Natural
     with Pre => Is_Sorted (Data);

end Sorted_Arrays3;
```

This version has the advantage that no separate body is needed for Is_Sorted because the entire function is implemented in the package specification. Furthermore, the body of an expression function is used to automatically generate a postcondition for the function as explained in Section 6.2.3.

9.1.2 *Ghost Variables*

The aspect Ghost may be applied to variables and constants as well as to functions. Such variables may only be used in verification assertions. They are commonly used in loop invariants. As with ghost functions, the runtime behavior of a valid program is the same with or without the ghost variables. We look at an example with ghost variables and constants in the next section.

9.2 Proof of Transitive Properties

Many properties we need to prove are transitive relations. As an example of such a property and the use of ghost entities, we return to the selection sort procedure. We gave its specification in Section 1.4 and repeat it here:

```
pragma Spark_Mode (On);
package Sorters is

   type Array_Type is array (Positive range <>) of Integer;

   function Perm (A : in Array_Type;
                  B : in Array_Type) return Boolean
   -- Returns True if A is a permutation of B
     with Global => null,
          Ghost  => True,
          Import => True;

   procedure Selection_Sort (Values : in out Array_Type)
   -- Sorts the elements in the array Values in ascending order
     with Depends => (Values => Values),
          Pre     => Values'Length >= 1 and then
                     Values'Last    <= Positive'Last,
          Post    => (for all J in Values'First .. Values'Last - 1 =>
                        Values (J) <= Values (J + 1)) and then
                     Perm (Values'Old, Values);
end Sorters;
```

Procedure Selection_Sort has two postconditions. The first postcondition states that the values returned are in ascending order. The second postcondition uses the ghost function Perm to state that the array returned, Values, is a permutation of the original array, Values'Old. That is, the new array is a reordering of the values in the original array. Without this second postcondition, a sort procedure that was given the integers (5, 7, 3, 3, 8) and returned (5, 5, 5, 5, 5) would be valid.

A common approach to determining whether two arrays are permutations is to sort each array and compare the sorted arrays. As we are using the permutation property to verify our sort procedure, we cannot use the sort to verify the permutation. Another approach is to count the number of each element in each array and compare the counts. As our array components are integers, this approach could require 2^{32} counters for 32-bit integer representations.

We take another approach to show that the result is a permutation of the original. Function Perm is a ghost function. The aspect Import states that the body of the function is external to the SPARK program. Rather than write a body, we will give the proof tool the mathematical rules it needs to verify our permutation postcondition. Here is the complete package body that implements our selection sort:

```
pragma Spark_Mode (On);
package body Sorters is

   function Perm_Transitive (A, B, C : Array_Type) return Boolean
     with Global => null,
          Post   => (if Perm_Transitive'Result
                        and then Perm (A, B)
                        and then Perm (B, C)
                     then Perm (A, C)),
          Ghost  => True,
          Import => True;


   procedure Swap (Values : in out Array_Type;
                   X      : in     Positive;
                   Y      : in     Positive)
     with Depends => (Values => (Values, X, Y)),
          Pre     => (X in Values'Range and then
                      Y in Values'Range and then
                      X /= Y),
          Post => Perm (Values'Old, Values)    and then
                  (Values (X) = Values'Old (Y) and then
                   Values (Y) = Values'Old (X) and then
                   (for all J in Values'Range =>
                     (if J /= X and J /= Y then Values (J) = Values'Old (J))))
   is
      Values_Old : constant Array_Type := Values
        with Ghost => True;
      Temp : Integer;
```

```
begin
   Temp      := Values (X);
   Values (X) := Values (Y);
   Values (Y) := Temp;
   pragma Assume (Perm (Values_Old, Values));
end Swap;


-- Finds the index of the smallest element in the array
function Index_Of_Minimum (Unsorted : in Array_Type) return Positive
  with Pre  => Unsorted'First <= Unsorted'Last,
       Post => Index_Of_Minimum'Result in Unsorted'Range and then
               (for all J in Unsorted'Range =>
                Unsorted (Index_Of_Minimum'Result) <= Unsorted (J))
is
   Min : Positive ;
begin
   Min := Unsorted'First ;
   for Index in Unsorted'First .. Unsorted'Last loop
      pragma Loop_Invariant
        (Min in Unsorted'Range and then
        (for all J in Unsorted'First .. Index - 1 =>
           Unsorted (Min) <= Unsorted (J)));

      if Unsorted (Index) < Unsorted (Min) then
         Min := Index;
      end if ;
   end loop;
   return Min;
end Index_Of_Minimum;


procedure Selection_Sort (Values : in out Array_Type) is
   Values_Last : Array_Type (Values'Range)
     with Ghost => True;
   Smallest : Positive ;  -- Index of the smallest value in the unsorted part
begin -- Selection_Sort
   pragma Assume (Perm (Values, Values));
   for Current in Values'First .. Values'Last - 1 loop
      Values_Last := Values;
      Smallest := Index_Of_Minimum (Values (Current .. Values'Last));

      if Smallest /= Current then
```

```
            Swap (Values => Values,
                  X      => Current,
                  Y      => Smallest);
         end if;

         pragma Assume (Perm_Transitive (Values'Loop_Entry, Values_Last, Values));

         pragma Loop_Invariant (Perm (Values'Loop_Entry, Values));
         pragma Loop_Invariant   -- Simple partition property
           ((for all J in Current .. Values'Last =>
                 Values (Current) <= Values (J)));
         pragma Loop_Invariant   -- order property
           ((for all J in Values'First .. Current =>
                Values (J) <= Values (J + 1)));
      end loop;
   end Selection_Sort;
end Sorters;
```

The heart of the permutation verification is in procedure Swap starting on line 14. Its postcondition states that (1) the resulting array is a permutation of the original array; (2) the values at index locations X and Y were swapped; and (3) the rest of the values in the array are unchanged.

We know that swapping two elements in an array creates a permutation of the original array. However, the SPARK tools do not know this fact. We use **pragma** Assume on line 34 with the ghost function Perm to state this fact. After swapping two elements, the SPARK tools will assume that the result is a permutation of the original. We used the ghost constant Values_Old to hold a copy of the original array. We could not use Values'Old as the 'Old attribute can only be used in postconditions.[2]

Next we have to tell the SPARK tools that after multiple element swaps the final array is a permutation of the original array. This is a transitivity property: if A and B are permutations and B and C are permutations, then A and C are permutations. We use another ghost function, Perm_Transitive defined on lines 4–11, to express this transitivity. The postcondition of this function states the transitivity property. A function postcondition should always mention the result of the function so we have also included it. As the postcondition is all we need for our proof, we included the aspect Import so we do not have to write a body for this function.

The **pragma** Assume on line 78 tells the proof tools that after each swap, the result is still a permutation of the original. The three parameters of the call to Perm_Transitive in this pragma are the original array (its value prior to the start of

the loop), the array before the last swapping, and the current array. We use the ghost variable Values_Last to hold a copy of the array prior to swapping elements.

There is still one more fact we need to give to the proof tools: an array is a permutation of itself. This fact is given in the **pragma** Assume on line 67.

As usual, we need to help the proof tools prove our postconditions by writing invariants for our loops. While you may only write one invariant for each loop, you may use multiple Loop_Invariant pragmas as long as they are consecutive. The three pragmas on lines 80–86 could be written as a single pragma using the conjunctive operator **and then** as was done in the function Index_Of_Minimum. Notice how the loop invariants in both Selection_Sort and Index_Of_Minimum support their postconditions.

The approach taken in this example may be applied to any proof requiring a transitivity property. Let us look at a more general example. In the specification,

- declare the property for which there is transitivity as a ghost function without a body and
- use that property in the postcondition of the change operation you wish to prove maintains that property.

See if you can pick out these two steps in our sorting package specification (page 330).

In the package body,

- declare the transitivity relation as a ghost function without a body (see lines 4–11 of the sorting package body),
- add a **pragma** Assume to the beginning of the change operation body to state that the property applies to itself (see line 67 of the sorting package body),
- declare a ghost variable (see lines 63–64 of the sorting package body) and assign the current value to it prior to applying a change (see line 69), and
- add a **pragma** Assume with the transitive relation ghost function after carrying out a change to indicate that the property applies to the current value (see line 78 of the sorting package body).

Let us look at a simpler example that illustrates this approach.[3] Procedure Change makes some modification to its parameter X with the postcondition that the transitive property, given by procedure Property, holds.

```
package Transitive
  with SPARK_Mode => On
is
   function Property (X, Y : in Natural) return Boolean
     with Global => null,
```

```
         Ghost  => True,
         Import => True;

   procedure Change (X : in out Natural)
     with Post => Property (X'Old, X);

end Transitive ;
```

Here is the body of this package. Notice that we have no idea about the meaning of Property, just that it is transitive.

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

   function Prop_Transitive (A, B, C : in Natural) return Boolean
   -- Define the transitivity of Property
     with Global => null,
          Post   => (if Prop_Transitive'Result
                        and then Property (A, B)
                        and then Property (B, C)
                     then Property (A, C)),
           Ghost  => True,
           Import => True;

   procedure Shift (X : in out Natural)
     with Post => Property (X'Old, X)
   is
   begin
      -- Tell the proof tool that Property holds after this operation
      pragma Assume (Property (X, X / 2));
      X := X / 2;
   end Shift ;

   procedure Change (X : in out Natural) is
      X_Last : Natural
        with Ghost => True;
   begin
      -- Tell the proof tool that the Property applies to an unchanged value
      pragma Assume (Property (X, X));
      for I in 1 .. 10 loop
         pragma Loop_Invariant (Property (X'Loop_Entry, X));
         X_Last := X;
         Shift (X);
```

```
            -- Tell the proof tool that
            --    because the Property holds for X'Loop_Entry and X_Last and
            --    the Property holds for X'Last and X then the property
            --    must hold for X'Loop_Entry and X
            pragma Assume (Prop_Transitive (X'Loop_Entry, X_Last, X));
         end loop;
      end Change;

end Transitive ;
```

Until recently, the default theorem provers were not able to complete the proof of our selection sort. And they took a while to prove our simpler transitive property example. However, the Z3 theorem prover was able to prove both of these packages in a very short amount of time. See Section 9.3.2 for a brief discussion of alternative theorem provers.

9.3 Proof Debugging

As a programmer you have no doubt learned a variety of techniques for finding and fixing bugs in your programs. You have probably also learned that program debugging can be difficult and requires significant practice. Special debugging tools exist to simplify finding and removing bugs, yet such tools also require practice to use well.

When you start to formally verify your programs using SPARK you need to develop skills for a new kind of debugging: proof debugging. If a proof is unsuccessful, how do you find out what is wrong and how do you fix it? Of course, if the proof is unsuccessful because the program has an error, fixing the error is a necessary first step. It is a surprisingly easy step to overlook –

- If a proof fails, check carefully that the code is right before trying to fix the proof.

Even before reviewing the code itself, it is worth stepping back to be sure the formal specification of that code is correct. For example, you may have implemented a subprogram properly, but if you stated the postcondition wrong, it may not prove. Here the problem is not in your implementation but in your specification of what the subprogram is intended to do. It is surprisingly easy to overlook this step as well –

- If a proof fails, check carefully that your assertions state what you intend.

However, a proof might fail even on a correctly specified and implemented code because of insufficient information in your program or because of limited

theorem prover technology. In this section we talk about various methods of proof debugging that help you diagnose such cases and fix them. However, be aware that like any debugging activity, proof debugging requires practice and experience. The best way to learn how to do it is to work through many troublesome proofs.

9.3.1 *Proof Workflow*

Imagine that you have just completed a package of SPARK code that successfully compiles as Ada and that successfully passes examination and so is free of flow issues. You are now ready to prove the code is free of runtime errors and that it obeys its pre- and postconditions and other assertions. Unfortunately when you run the SPARK tools, you find that some proofs fail. How should you proceed?

The automatic theorem prover that backs the SPARK tools requires a certain amount of time to complete the proof of each verification condition. Unfortunately it is not possible, in general, to set a specific limit on how much time is required. Verification conditions that are false will never prove no matter how long the theorem prover works (assuming the theorem prover is sound). But even verification conditions that are true may not be provable by any given theorem prover or, even if they are, may require a very long time for the proof to complete.

The problem with allowing the theorem prover to work for a long time is that many failing verification conditions will require you to wait an inordinate amount of time for the proving process to finish. For example, if your package has fifty verification conditions of which ten fail to prove and you set your theorem prover to work for a maximum of sixty seconds on each verification condition, you may have to wait ten minutes to get past the failing verification conditions – in addition to whatever time the prover spends on the other proofs. This makes the process of editing and reproving your code tedious because each time you try to prove your code you must wait for all the failing verification conditions to time out.

The SPARK tools mitigate this problem to some extent by caching proof results on a per-subprogram basis. If you edit only one subprogram in your package, it is only that subprogram that is reprocessed. The proofs (failed or otherwise) done in the unchanged subprograms are not recomputed. However, this is not as helpful if you are working in a complex subprogram with many failing verification conditions. Thus, your first line of defense when trying to prove complicated code is to avoid complex subprograms –

- Split complicated subprograms into several simpler ones.

In addition to taking better advantage of the SPARK tools' proof caching features, this also tends to reduce the complexity of the verification conditions by reducing the number of paths in each subprogram, thus, speeding up their processing. As an additional bonus, it can make your program easier for a human to understand as well.

We recommend at first using a relatively short timeout, say five seconds, on the theorem prover. This will abort the prover on failing verification conditions quickly and gives you a quicker turnaround time in your editing cycle. Unfortunately this also means some proofs will fail that might succeed if given more time. As you work with your code and fix "easy" failing proofs by, for example, adding necessary preconditions or loop invariants, you can gradually increase the prover's timeout. It is reasonable to spend twenty or even thirty seconds on each verification condition as the number of failing verification conditions declines, depending on your patience. The longer timeout by itself may allow other proofs to succeed with no additional work on your part –

- Start with a short timeout on the prover, but increase it as the number of failing verification conditions drops.

Sometimes you will encounter a verification condition that can be proved using an exceptionally long timeout of, perhaps, many minutes. It may be possible to prove such a verification condition more quickly by changing your code. Using extremely long timeout values should only be done as a last resort. However, if nothing else is working and you are confident that your code is actually correct, trying a timeout value of ten or twenty minutes or more may be successful.

Of course it also helps if your development system is fairly powerful. Ten seconds on one machine might get more work done than two minutes on another. In addition, the SPARK tools support multicore systems. You can set the number of processors used by GNATprove with a switch on the command line or in the GNAT project properties.

In the version of the SPARK tools available at the time of this writing, each verification condition is analyzed by a single thread. Running, for example, four threads at once allows the tools to process four verification conditions in parallel, but each verification condition by itself is not accelerated. A single verification condition that requires many minutes to prove will still require many minutes regardless of how many threads you use –

- Use your multi-core processor to make long timeouts more tolerable by allowing the SPARK tools to work on several difficult proofs simultaneously.

9.3.2 *Alternate Theorem Provers*

At the time of this writing, the SPARK tools actually use two theorem provers each time you launch a proof: CVC4 (New York University, 2014) and Alt-Ergo (OCamlPro, 2014). You can also download and install other provers. Different provers have different strengths and weaknesses. You may find some verification conditions provable by one prover but not others. In fact, this is the reason the SPARK tools ship with two provers that are used together. Yet even if two provers both discharge a verification condition, there might be considerable difference in the time required or memory consumed to do so. Furthermore, theorem proving is an evolving technology so the capabilities and performance of provers used in the future are likely to be better than that available today.

As an example, until recently we were not able to prove the selection sort given in Section 9.2 with these two provers, even with very long timeouts. We discovered that the Z3 theorem prover (Bjørner, 2012) could prove this procedure with a very short timeout. However, a new version of CVC4 was also able to prove this code –

- Try using alternate theorem provers to see how they fare against your difficult verification conditions.

Section 9.2 of the *SPARK 2014 Toolset User's Guide* (SPARK Team, 2014b) provides information on using alternative theorem provers. GNATprove can use any theorem prover that is supported by the Why3 platform.[4] To use another prover, it must be listed in your .why3.conf configuration file. The command `why3config --detect-provers` can be used to search your `PATH` for any supported provers and add them to your .why3.conf configuration file. Any prover name configured in your .why3.conf file can be used as an argument to switch `--prover` when running GNATprove from the command line or entered into the "Alternate prover" box when running GNATprove from GPS. You can specify that multiple provers be used by listing their names separated by commas.

Theorem provers all endeavor to be sound in the sense that they should never claim to have proved a false verification condition. Thus if any one prover claims to have discharged a verification condition, it is reasonable for you to say it has been proved. This does assume, of course, that all the provers you use are themselves error free. Using multiple provers increases the size of your *trusted computing base*, defined here as the body of software that needs to be correct for you to have faith in the validity of your proofs. Clearly that is a disadvantage of using multiple provers, but we feel that the advantages of doing so generally outweigh the disadvantages.

9.3.3 *General Techniques*

The previous discussion is mostly about different ways to run the tools. In this and the following subsections we talk about ways to modify your program to help difficult proofs go through. Here we make a few general observations that can be helpful to consider. In the later subsections we discuss more active techniques of proof debugging –

- Use types and subtypes that are as specific as possible.

Types such as Integer or Float are almost never the right choice. Instead you should define types that encode as much information about your problem as possible and then use those types. The built-in subtypes Natural and Positive are sometimes appropriate, but often you can do better than that. For example:

```
subtype Class_Size_Type   is Natural  range 0 .. 500;
subtype Header_Size_Type  is Positive range 20 .. 40;
subtype Student_ID_Type   is String(1 .. 9);
subtype Extended_Integer  is Very_Longs.Very_Long(20);
```

Many proofs are greatly simplified if the SPARK tools have knowledge about the constraints that are really being used. For example, computing the average (A + B)/2 of two Positive values may overflow, but computing the average of two Class_Size_Type values will not because the computation is done in Class_Size_Type's base type (likely Integer). Similarly, proving that X/Y does not entail division by zero is trivial if Y's subtype does not allow the zero value.

While tight constraints may make certain proofs more complicated, in the long run they simplify the overall proving process tremendously. A corollary of this rule is that unconstrained types, such as unconstrained array types, tend to make the proofs harder. Although they can be very useful for writing general purpose code, do not use them unless they are really necessary –

- Verification is typically easier if the code is less general.

Many subprograms are written with assumptions about the nature of their inputs. The SPARK tools cannot know these assumptions unless you make them explicit using preconditions. Suitable preconditions will often allow difficult proofs in the subprogram's body to go through because more information is now available to the tools when analyzing the body. Of course, strong preconditions do add proof obligations at the call site, but probably those are proofs that should have been done anyway –

- Add strong preconditions.

Preconditions and tightly specified subtypes are really two ways of saying the same thing. Both entail bringing more information into a subprogram body.

Preconditions can express relationships between inputs and are thus more general and have a heavier weight. It is usually better to use subtypes when it is possible to do so and resort to preconditions for the situations in which Ada's subtyping is insufficiently expressive.[5]

Many subprograms with loops will need loop invariants to be fully proved. Writing good loop invariants can be tricky. However, if you find difficult verification conditions inside or past a loop, it is likely that a loop invariant is what you need. Of course, loop invariants also add their own proof obligations, but that is often a necessary cost for getting the other proofs to work –

- Add loop invariants.

You can get any verification condition to prove by adding an appropriate Assume pragma. You might convince yourself that the assumption you want to make is always valid and so be tempted to do this –

- Avoid using Assume.

See Section 6.3.3 for a discussion about why the Assume pragma is dangerous. That said, Assume does have a role to play in the proof debugging process as we describe in the next section and in conveying to the SPARK tools information that would ordinarly be outside their knowledge such as when we proved transitive properties in Section 9.2.

9.3.4 *Containing Proof Problems*

Although uninhibited use of the Assume pragma can be dangerous, we now describe a powerful way to use it temporarily while debugging a failing proof. The procedure can be described with three steps:

1. As always, when trying to fix a failing proof, try to understand why the code looks correct to you. For example, imagine trying to explain to someone else why the code is correct.
2. Temporarly add an Assume pragma to the code that embodies your understanding of what makes the code correct. Adjust your assumption until the SPARK tools are able to complete the previously failing proof.
3. If the assumption still looks reasonable to you after any adjustments you made to it, try changing it to an Assert pragma. If the SPARK tools are able to prove the assertion, you are done. If not, try changing the assumption, or moving the assumption to a different place, or even repeating the process by using a second assumption to assist with proving the newly placed assertion.

In some cases you may find the only way to get a difficult proof to succeed is to make assumptions that are entirely unreasonable. This is a strong indication that there may, in fact, be something wrong with your code or with your assertions. The nature of the assumption you have to make, and where you have to make it, can help you find the fault.

We recommend using assumptions for this process rather than assertions directly because the SPARK tools will not attempt to prove an assumption and that speeds up the analysis of your program. If instead you used an assertion immediately and if the tools failed to prove both the assertion and the original verification condition, you end up waiting twice as long before you can try again. This is particularly an issue if you are using long timeouts. Just be sure you convert the assumption into an assertion before you are done.

This approach could be called *backward tracing* because it involves starting at the failing proof and moving backward through the code until you find the assertion(s) you need. For example, suppose the program contained an assignment statement such as

```
X := A/B;
```

Suppose the proof that `B /= 0` is failing. You can "fix" this proof by using an assumption such as

```
pragma Assume (B /= 0);
X := A/B;
```

Of course, as soon as you convert the assumption to an assertion that assertion will immediately fail to prove and you will have gained nothing. However, if you move backward through the program flow, you may be able to find a place where you can say something more meaningful. For example, showing a bit more context in our hypothetical program gives

```
B := F(X);
if B <= 0 then
   C := G(X) + 1;
   pragma Assume (C /= 1);
   B := C - 1;
end if;
X := A/B;
```

In this case if the **if** block is skipped, `B` must be greater than zero. Otherwise, if we assume `C` receives a value that is not one from the expression `G(X) + 1`, all is well. In this case converting the assumption to an assertion may still fail to prove, but the exercise perhaps reveals that `G`'s postcondition is inadequately specified. For example, you might find yourself saying, "That is not a problem

because G always returns a positive value and any positive value plus one cannot be one." The fix to the failing proof might be as simple as adding a postcondition to G such as

```
Post => G'Result > 0;
```

Of course the real problem might be that G's result type is not declared appropriately. Perhaps it was declared as Integer but instead should be Positive, or even better, some more constrained subtype of Positive.

This example is simple and highly contrived. However, it illustrates the general approach of exploring the paths leading to the troublesome proof by injecting suggestive assumptions into those paths until the proof can be made to work. The assumptions can then be converted to assertions or at least guide other changes that need to be made in the program. As hinted here, the method can often reveal problems in remote parts of the program or help you uncover faults you missed.

Another similar approach that could be called *forward tracing* entails starting at the beginning of the program flow, adding assertions (not assumptions) that are "obviously" true, and working toward the failing proof. Each assertion that you add can contain progressively more information and be readily proved using the assertions before it, until the amount of information available to the tools by the time the flow reaches the failing verification condition is sufficient to complete the difficult proof. This approach is similar to proving a difficult theorem in mathematics by first proving a series of increasily strong lemmas that ultimately make the final proof straightforward. Adding assumptions or assertions in this way is the proof debugging equivalent of adding debugging print statements to your program.

As with ordinary debugging, finding the problem is really only the first step toward solving it. Often the more difficult step is deciding on the most appropriate fix. Should more suitable types be declared and used? Should a runtime test be added using a conditional statement? Should pre- or postconditions be adjusted? All of these choices have the potential to affect significant parts of your program. For example, adding a stronger postcondition on a procedure will increase the burden of proof inside that procedure. If that, in turn, requires the procedure to have a stronger precondition, more proof obligations may be created throughout the code base.

9.3.5 *Partitioning Unproved Code*

Sometimes completing the proof of certain verification conditions is infeasible for one reason or another. Yet you may still have confidence about the code in

question as a result of rigorous testing, code review, or other reasons. Having program units with failing proofs is unsightly. It can also be hazardous because you become insensitive to the repeated ignorable messages from the tools. If the messages change or a new message appears that is not ignorable, you may not notice. Thus, it is highly desirable to have program units partitioned into those that prove cleanly and those that are not intended to be SPARK.

The SPARK tools do provide a means for suppressing warnings using **pragma** Warnings and for justifying failed proofs using **pragma** Annotate. See the *SPARK 2014 Toolset User's Guide* (SPARK Team, 2014b) for more information. Of course, suppressing messages should be done cautiously and only after careful review or concentrated testing provides evidence of correctness.

In cases where a larger block of code contains numerous unproved verification conditions, or is even outside of SPARK, it may be more appropriate to factor the unproved code into a separate unit. This can help draw attention to the difficult code as a subject for future review. The Ada feature of private child packages is particularly useful in this context. We illustrate the technique with a simple example.

In an earlier version the SPARK tools did not have a good understanding of the effect of the bit manipulation operations.[6] As a result, proving verification conditions related to those operations was often problematic.

Suppose one wanted to create a package implementing serveral network protocols. It would be reasonable to organize such a potentially large package as a collection of child packages working together. The top-level parent package might include declarations of interest to the entire system such as declarations of fundamental types representing the basic units of data flowing over the network. A partial specification of such a package Network follows:

```
package Network
   with SPARK_Mode => On
is
   type Octet        is mod 2**8;
   type Double_Octet is mod 2**16;
end Network;
```

Children of this package might implement various network protocols such as, for example, TCP, UDP, or application protocols like SMTP. Assume that various helper subprograms that are useful when implementing multiple protocols are factored out into a child package Network.Helpers. For sake of illustration, suppose that package contained a procedure Split16 that separated a 16-bit value

into an 8-bit most significant octet and an 8-bit least significant octet. One might attempt to write that procedure as follows:

```
package body Network.Helpers0
   with SPARK_Mode => On
is

   function Shift_Right (Value : in Double_Octet;
                         Count : in Natural) return Double_Octet
      with Import => True,
           Convention => Intrinsic;

   procedure Split16 (Value : in  Double_Octet;
                      MSB   : out Octet;
                      LSB   : out Octet) is
   begin
      MSB := Octet (Value and 16#00FF#);
      LSB := Octet (Shift_Right (Value, 8));
   end Split16;
end Network.Helpers0;
```

Of course, a realistic package Network.Helpers would likely contain many other subprograms of varying complexity. Function Shift_Right has two aspects. The Boolean aspect Import tells us and the tools that the body of this function is defined in a foreign language that is imported into our Ada program. The aspect Convention names the foreign language. In Section 7.2 we used Convention => C to say that the body of a subprogram was written in C. In this case, the Convention specified is Intrinsic . This convention means that the body of the subprogram is provided by the compiler itself, usually by means of an efficient code sequence (usually a single machine instruction), and the user does not supply an explicit body for it. Intrinsic operations available with the GNAT compiler are listed in Section 2.10.2 of the *GNAT User's Guide* (GNAT, 2015b).

The desire is to write as much of the network protocol implementations in SPARK as possible. However, the older version of the SPARK tools had difficulty proving, for example, that the conversion of Value **and** 16#00FF# to an Octet will not raise Constraint_Error because it did not understand the significance of the masking operation. All the tools saw was that the type being converted, Double_Octet, might have a value that would not fit into Octet. Similar comments apply to the use of Shift_Right to find the least significant byte. To get around this, we can move the problematic constructs into a separate package as follows:

```
pragma SPARK_Mode(On);
private package Network.Bit_Operations is
```

```
   -- Returns the least significant byte in a 16-bit value.
   function TakeLSB_From16 (Value : in Double_Octet) return Octet
     with
       Inline => True,
       Global => null;

   -- Returns the most significant byte in a 16-bit value.
   function TakeMSB_From16 (Value : Double_Octet) return Octet
     with
       Inline => True,
       Global => null;
end Network.Bit_Operations;
```

The aspect Inline applied to the two functions in this package specification requests that the compiler replace subprogram calls with copies of the subprogram. This substitution typically improves the execution time and stack memory usage but increases the size of the program. Here is the body of this package:

```
pragma SPARK_Mode(Off);
package body Network.Bit_Operations is

   function Shift_Right (Value : in Double_Octet;
                         Count : in Natural) return Double_Octet
     with
       Import => True,
       Convention => Intrinsic;

   function TakeLSB_From16 (Value : in Double_Octet) return Octet is
   begin
      return Octet (Value and 16#00FF#);
   end TakeLSB_From16;

   function TakeMSB_From16 (Value : in Double_Octet) return Octet is
   begin
      return Octet (Shift_Right (Value, 8));
   end TakeMSB_From16;

end Network.Bit_Operations;
```

This package has a specification with SPARK_Mode set to On so that it can be used from SPARK code. However, the body has SPARK_Mode set to Off as a

way of explicitly saying that its code is not subject to proof. The Network.Helpers package is now

```
with Network.Bit_Operations;
use  Network.Bit_Operations;
package body Network.Helpers
   with SPARK_Mode=> On
is
   procedure Split16 (Value : in  Double_Octet;
                      MSB   : out Octet;
                      LSB   : out Octet) is
   begin
      MSB := TakeMSB_From16 (Value);
      LSB := TakeLSB_From16 (Value);
   end Split16;
end Network.Helpers;
```

This new version proves easily as all the types match without conversions.

Notice that Network.Bit_Operations is a private child package. This is stated by the use of **private** in its specification. Private children are commonly used in Ada as a way of factoring out internal support facilities from a collection of related packages. See Section 3.5.1 for more information. Here we use a private child to encapsulate the unprovable code needed by some other package(s) allowing the bulk of the code base to be proved cleanly.

9.3.6 *When All Else Fails*

If you are completely unable to get a verification condition to prove after trying the techniques described in this section, you can instead resort to testing to explore the issue. We discuss this in Section 8.4. Here, the value of SPARK is in the focus it provides. The unproved verification conditions must be carefully covered by test cases, whereas tests to cover the proved aspects of the code are less critical and can be given a lower priority –

- Use testing to cover unproved verification conditions.

9.4 SPARK Internals

In this section we give an overview of how the SPARK tools work internally. Our intention is not to provide a detailed description of the theory and operation of the tools, but rather to give you a sense of what they are doing so you can be a

more effective tool user. We also hope to interest you in learning more about how to use the advanced techniques made possible by the SPARK architecture.

We illustrate the internal operation of the tools by way of a short example. Consider a package Workspaces that contains a number of utility subprograms for manipulating unconstrained arrays of natural numbers. An abbreviated specification for that package might be as follows:

```
pragma SPARK_Mode (On);
package Workspaces is
   type Workspace_Type is array ( Positive  range <>) of Natural;
   function Generate_Workspace (Size :  in  Positive ) return Workspace_Type;
end Workspaces;
```

The function Generate_Workspace returns an array of the specified size initialized in a specific way. In particular, all array elements in the first half of the array are initialized to their index values, and all array elements beyond the halfway point are initialized with the same values in descending order. For example, a call to Generate_Workspace(6) returns the array (1, 2, 3, 3, 2, 1).

To keep the example simple from a verification standpoint we do not provide a postcondition for the function. However, we will verify that our implementation is free of runtime errors. Our first attempt at an implementation follows:

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

   function Generate_Workspace (Size :  in  Positive ) return Workspace_Type is
      Workspace : Workspace_Type(1 .. Size) := (others => 0);
   begin
      for J in 1 .. Size / 2 loop
         Workspace (J) := J;
         Workspace ((Size + 1) - J) := J;
      end loop;
      return Workspace;
   end Generate_Workspace;

end Workspaces;
```

The SPARK tools are unable to prove that the computation of Size + 1 will not overflow. This attempted proof shows a real problem with the implementation. Because Natural'Last is the same as largest value of Natural's base type (using GNAT's default overflow options), adding one to Size might, in fact, cause an overflow. But what proof are the tools actually trying to complete and how are they trying to do it?

The SPARK tools make use of a program verification system called Why3 (Bobot et al., 2011). The Why3 tools accept as input a language called WhyML that provides many facilities similar to other ML family languages such as Standard ML or Objective Caml. WhyML programs have executable semantics and could, in principle, be run like any other program.

WhyML was developed as an intermediate language for software verification. WhyML augments the usual ML language elements with extensive features for expressing pre- and postconditions, loop invariants, and other assertions similar to the ones available in SPARK. WhyML also supports the ability to specify and use various *theories* about the properties of types and data structures. The Why3 system includes a predefined library of theories for commonly used entities.

The main SPARK tool, GNATprove, translates each unit it analyzes into a collection of WhyML modules, with one module for each subprogram analyzed. This module, along with extensive supporting material, is written into a file (`workspaces.mlw` for our example) in the proof folder for the project. By default the proof folder is the `gnatprove` folder in the project's build output folder.

As an example of some of the supporting material created for SPARK programs, consider the following WhyML module describing the Ada built-in type Natural:

```
module Standard__natural
  use import "_gnatprove_standard".Main
  use import "int".Int
  use        "_gnatprove_standard".Integer

  type natural "bounded_type"
  function first : int = 0
  function last : int = 2147483647
  predicate in_range(x : int) =
    ((first <= x) /\ (x <= last))

  clone export "ada__model".Static_Discrete with
  type t = natural,
  function first = first,
  function last = last,
  predicate in_range = in_range
end
```

Here, various library theories are imported, including some provided by the SPARK tools to supplement those available in the stock Why3 platform. A type

`natural` is introduced along with functions returning the lower and upper bound of that type. In addition, a predicate is defined that takes an integer and returns true if the integer is in the bounds of the type. We note that in WhyML integers have arbitrary precision and thus model true mathematical integers. Finally, the `Static_Discrete` theory is cloned with the type `natural` substituted for `Static_Discrete`'s type `t` and similarly for functions `first` and `last` and predicate `in_range`. The result is a full logical description of the Ada type Natural as a discrete type.

All standard Ada types are provided in this way in addition to types defined in the program itself. In this example, this includes the unconstrained array type Workspace_Type, which is described by cloning a library `Unconstr_Array` theory. Anonymous types used in the program are given tool-generated names and described as well. In the example, the dynamic subtype used to specify the bounds on the array Workspace is given the name `TTworkspaceSP1` and described by cloning a library `Dynamic_Discrete` theory.

The WhyML module created to check contracts and freedom from runtime errors of the SPARK function Generate_Workspace appears in `workspaces.mlw` after all the supporting definitions. The SPARK tools do not format the WhyML in a human-friendly way, making the code difficult to read. However, WhyML is intended to be producible by humans as some users of the Why3 system write WhyML directly. With more natural names and better formatting, it is clearer that the WhyML program is a translation of the SPARK subprogram with all checks made explicit.

The following fragment of WhyML is taken from the module created from Generate_Workspace. To promote readability, the names have been greatly simplified and the formatting arranged suggestively.

```
1  workspace :=
2    Array__1.set
3     (!workspace)
4
5     (let  workspaces_0 =
6       (("VC_OVERFLOW_CHECK"
7         (integer.range_check_((positive.to_int(size)) + 1))) - !j)
8      in
9       ((assert
10        { ("VC_INDEX_CHECK"
11            (((integer.to_int  workspace_first) <= workspaces_0) /\
12              (workspaces_0 <= (integer.to_int  workspace_last))))  };
13      workspaces_0)))
14
15    (natural.of_int (!j));
```

Understanding WhyML also requires a familiarity with the syntax and semantics of ML-like languages in general. The preceding expression was generated from the SPARK assignment statement

```
Workspace ((Size + 1) - J) := J;
```

The code creates a new version of the Workspace array by using a function `set` to set one of the elements of the existing array. In ML-like languages function arguments are separated by spaces, here shown as blank lines. The first argument on line 3 is the existing Workspace array. The name `workspace` is actually a kind of pointer, and the `!` operator is used to dereference that pointer.

The second argument to the function on lines 5–13 is the result of a "let expression" that creates a temporary variable `workspaces_0` initialized by (Size + 1) − J and ultimately just evaluates to that variable on line 13. GNATprove inserted an overflow check into the evaluation of Size + 1 using a range-checking helper function. Finally, before `workspaces_0` is returned, an assertion is used to check that its value is in the range of the array's index subtype.

The third argument to the `set` function on line 15 is just the value of J used on the right-hand side of the original assignment statement. Notice that no check is needed here because the type of J at the SPARK level guarantees its value must be in range of the array element.

The Why3 tools can convert WhyML directly into code in the Objective Caml programming language. Thus, GNATprove together with the Why3 tools constitute a kind of SPARK to OCaml compiler. However, of greater interest to us is the use of theorem provers. The Why3 tools extract from the WhyML "proof tasks" based on the requested checks and using other information in the program (not shown in the previous example). These proof tasks are then processed by Why3 into a form that can be read as input by a theorem prover. This processing is guided by a Why3 driver.

Different theorem provers have different requirements on the kinds of logics they can accept as input. It is the job of the Why3 driver to transform the proof tasks extracted from the WhyML program into a form suitable for the prover being driven.

In this case the overflow check fails to prove because overflow might, in fact, occur. The code is easily fixed by rearranging the order of operations in the expression (Size + 1) − J to (Size − J) + 1. This change results in correponding changes to the generated WhyML that allow the proof to succeed.

Why use an intermediate verification system such as Why3? Why not just generate proof tasks directly from the SPARK code in a format suitable for the theorem prover to process? In fact, that is exactly what the SPARK 2005

tools did. However, the architecture of SPARK 2014 is much more flexible and extensible. It can also capitalize better on advances in proof technology. For example, to support a new theorem prover one only needs to create a suitable Why3 driver for it. This could even be done by a third party; no changes to GNATprove are required. The SPARK tools ship with Why3 drivers for several popular theorem provers, and this is the basis of how the tools support multiple provers.

Also, it is possible to directly interact with the underlying tools at the Why3 level. Consider, for example, asking the SPARK tools to prove a mathematical property about trigonometry functions such as the Pythagorean identity

```
Sin(X)*Sin(X) + Cos(X)*Cos(X) = 1.0
```

Even if we set aside the difficulties associated with floating point arithmetic, it is not a simple matter to write suitable (provable) postconditions for Sin and Cos that would allow the above to be proved. The Pythagorean identity is a consequence of an underlying mathematical theory relating the two functions. To illustrate, suppose Cos was declared as follows:

```
function Cos(X : Float) return Float
  with Post => Sin(X)*Sin(X) + Cos'Result*Cos'Result = 1.0;
```

The body of Cos might be something like[7]

```
function Cos(X : Float) return Float is
begin
  return Sqrt(1.0 - Sin(X)*Sin(X));
end Cos;
```

It might be possible to prove the given postcondition of such an implementation, provided a suitable postcondition for Sqrt was available, but clearly Sin cannot be written in a similar way or else the two functions would be infinitely mutually recursive. Instead, some other implementation of Sin would be necessary, say based on the Taylor series expansion of $\sin(x)$. The relationship between such an implementation and the Pythagorean identity is not obvious; the connection requires an appeal to broader mathematical concepts.

Yet the Pyhagorean identity is an example of a basic relationship that might be quite useful in proving other properties of your program. How can you tell the SPARK tools about such relationships?

One way it can be done is to provide an *external axiomatization*. This is, in effect, a handwritten Why3 theory that describes the desired properties. The Why3 tools will use this theory like any other, allowing proofs to succeed that otherwise might not. However, care is needed when using this technique

because an error in the Why3 theory might introduce unsoundness and allow false conclusions to be proved as well as true ones. As we cautioned in Section 6.3.3, if even one false conclusion can be proved, then all conclusions can be proved. In this respect, careless use of external axiomatizations has much in common with careless use of the Assume pragma. See Section 9.5 of the *SPARK 2014 Toolset User's Guide* (SPARK Team, 2014b) for more information about external axiomatizations.

In addition to manually injecting human written WhyML into the proving process, it is also possible to conduct human proofs at this level as well. In particular, the SPARK tools ship with Why3 drivers supporting the Coq and Isabelle proof assistants. This may allow you to discharge some difficult proofs that elude fully automated theorem provers. However, the use of a proof assistant obviously requires you to be familiar with that tool as well. See Section 9.2.3 of the *SPARK 2014 Toolset User's Guide* (SPARK Team, 2014b) for more information about configuring Coq for use with SPARK.

Finally, there are other software verification systems that use Why3 as an intermediate verification system. One system of particular interest is Frama-C with the WP plug-in for C programs. Using a common intermediate verification language opens the possibility of seamless mixed language verification. In many embedded systems applications in which SPARK and C are often used together, the possibility of doing full verification across the SPARK-C boundary is very enticing.

The techniques mentioned here are advanced and, outside of this section, we have not discussed them in this book. However, they do illustrate that the architecture of SPARK 2014 sets the stage for more advances and developments in the future. This is an exciting time to be involved in the formal verification of software, and SPARK is at the forefront of the field.