The SPARK way to Correctness is Via Abstraction

John Barnes
11 Albert Road
Caversham
Reading RG4 7AN
United Kingdom
+44-118-9474125

1. ABSTRACT

This paper gives a short introduction to the SPARK language and illustrates how the use of abstraction leads towards correctness.

2. INTRODUCTION

Abstraction is a key concept in the design of many systems whether they be made of intangible software or real hard stuff such as an automobile. A good system will be such that the various components interact through well-defined interfaces in an appropriate manner. This should eliminate unwanted interactions which might occur if the interfaces are not properly defined. The brake pedal of your car should not change the volume of the radio and so on. This desirable state can be achieved by ensuring that interactions only occur via defined interfaces and moreover that the functionality of the components are completely and correctly specified by the interface definitions (the whole truth and nothing but the truth).

Ada provides interfaces through specifications – typically package specifications containing subprogram specifications. However, these subprogram specifications do not provide a full definition of the subprograms. All they provide is enough information to enable the compiler to construct calls of the subprograms but say little if anything about what the subprograms might actually do. Although the Ada approach enables information hiding to be achieved and good component specifications to be written, and indeed encourages these through its style, nevertheless it does not ensure correctness and completeness.

SPARK enables Ada specifications to be strengthened by providing more information about interfaces and the behaviour of components. This extra information can be provided at various levels. At the simplest level it ensures that a component can only interact with certain objects but need say nothing about what it does to them; at the highest level it provides a complete definition of what it does to the objects. At the simplest level it thus prevents unexpected side effects whereas at the highest level it can lead to complete proofs of correctness.

SPARK should be looked upon as a language in its own right. In practical terms, it is a subset of Ada with additional information provided through annotations which take the form of Ada comments. Programs are therefore compiled with a normal Ada compiler and in addition are examined with independent SPARK tools which also analyse the annotations.

It is often felt that formal tools are hard to use and require a great deal of effort. One of the advantages of SPARK is its flexibility. It can be used for formal proof but a great deal of benefit can be obtained by its use at the simplest level which requires little effort. This paper outlines some important features of SPARK using a number of examples.

3. ABSTRACTION

The first part of this paper introduces the basic ideas of abstraction and refinement.

3.1 A simple example

We start by considering a very simple example which shows how the SPARK annotations increase the level of information concerning abstraction. Consider the information given by the following Ada procedure specification

```
procedure Add(X: in Integer);
```

Frankly, it tells us very little. It just says that there is a procedure called Add and that it takes a single parameter of type Integer whose formal name is X. But it says nothing about what the procedure does. It might do anything at all. It certainly doesn't have to add anything nor does it have to use the value of X. It could for example subtract two unrelated global variables and print the result to some file. But now consider what happens when we add the lowest level of SPARK annotation. The specification might become

```
procedure Add(X: in Integer);
--# global in out Total;
```

This states that the only global variable that the procedure can access is that called Total. Moreover it has mode information similar to that of parameters; indeed a global variable can be looked upon as a parameter in which the actual is always the same. The SPARK rules also say more about the modes. Whereas in Ada the modes provide permission to read or update as appropriate, in SPARK such reading or updating is mandatory (SPARK generally abhors unused entities). So the specification tells us that the initial value of Total must be used (**in**) and that a new value will be produced (**out**) and also that the parameter X (**in**) must be used.

So now we know rather a lot. We know that a call of Add will produce a new value of Total and that it will use the initial value of Total and the value of X. We also know that Add cannot affect anything else. It certainly cannot print anything nor have any other malevolent side effect.

The next level of annotation gives the detailed dependency relations so that the specification becomes

```
procedure Add(X: in Integer);
--# global in out Total;
--# derives Total from Total, X;
```

In this particularly simple example, this adds no further information. We already knew that we had to use X and the initial value of Total and produce a new value of Total and this is precisely what this derives annotation says.

Finally we can add the third level of annotation which concerns proof and obtain

```
procedure Add(X: in Integer);
--# global in out Total;
--# derives Total from Total, X;
--# post Total = Total~ + X;
```

The postcondition explicitly says that the final value of Total is the result of adding its initial value (distinguished by ~) to the value of X. So now the specification is complete.

It is important to emphasize that these annotations are part of the procedure specification. (In the case of distinct specification and body, the annotations are not repeated in the body; if there is no distinct specification then they occur in the body before the reserved word **is**.) The annotations separate the interaction between the caller and the specification from that between the specification and the implementation. Hence the Examiner (the main SPARK tool) carries out two sets of checks; it checks that the annotations are consistent with the procedure body and it also checks that the annotations are consistent with each call of the procedure.

Thus when we come to implement Add, if we access a global other than Total or use Total or X in a way inconsistent with the mode information then the SPARK Examiner will produce appropriate error messages.

Generally, the higher levels of annotation enable the Examiner to carry out a more searching analysis.

3.2 State

The idea of state is vitally important. Programs do things by changing the state of objects in a general sense. In Ada, state is typically held in the form of variables in packages. A simple example is provided by a random number generator in which the state of the sequence is held in a variable hidden in a package body. Consider

```
package Random_Numbers
--# own Seed;
--# initializes Seed;
is
  procedure Random(X: out Float);
  --# global in out Seed;
  --# derives X, Seed from Seed;

end Random_Numbers;

package body Random_Numbers is
  Seed: Integer;
  Seed_Max: constant Integer := ... ;

  procedure Random(X: out Float) is
  begin
    Seed := ... ;
    X := Float(Seed) / Float(Seed_Max);
  end Random;

begin          -- initialization part
  Seed := 12345;
end Random_Numbers;
```

This example shows the package body containing the declaration of a variable Seed and the body of the subprogram Random. Each call of Random updates the value of Seed using some pseudo-random algorithm and then updates X by dividing by the constant Seed_Max. Each successive value of Seed depends upon the previous value and is preserved between calls of Random. The variable Seed is initialized in the initialization part of the package body.

This example also illustrates a number of other annotations. The variable Seed has to be mentioned in both an own annotation and an initialization annotation of the package specification. The own annotation makes it visible to other annotations and the initializes annotation indicates that it must be initialized by the elaboration of the package. The procedure Random contains a global annotation for Seed as well as a derives annotation.

The initializes annotation can also be satisfied by initializing Seed in its declaration. An alternative approach might be to declare some procedure Start in the package Random_Numbers (to be called from outside) whose purpose is to assign a first value to Seed. In this case an initializes annotation would not be required but the

Examiner will complain if flow analysis reveals that Random is being called before Start.

It is important to observe that from the Ada point of view the variable Seed is not declared until the body and is thus not known to the compiler at the point of the specification of the subprogram Random. However, Seed is a global variable of Random from the point of view of SPARK and thus must be mentioned in the annotation for Random so that flow through Random may be tracked; the own annotation ensures that Seed is known to the Examiner at the specification of Random.

The derives annotation shows explicitly that each call of Random produces a number X derived from Seed and also modifies Seed. As mentioned earlier this annotation is optional.

The variable Seed is protected from manipulation by users of the procedure Random by being declared within the body of the package although it is visible in the annotations in the specification. It could be argued that making the existence of Seed known to the user is a violation of abstraction. However, we certainly ought to know that the procedure Random does something to some state external to itself otherwise we could deduce that each call of Random would inevitably produce the same value each time it is called. On the other hand we don't need to know exactly what Seed is and indeed in this example the external view reveals no details.

3.3 Abstract state machines

The random number package is a very simple example of an abstract state machine. In general an abstract state machine is an entity, which has well defined states plus a set of operations, which cause state transitions; properties of the state can be observed by calling appropriate functions.

An abstract state machine is typically represented in Ada by a package, with variables which record its state declared in its body. Procedures that act on the machine and functions that observe its state are specified in the visible part of the package specification. All other details are hidden in the package body.

The following shows the full details of a single stack treated as an abstract state machine with the state initialized automatically on elaboration.

```
package The_Stack
--# own S, Pointer;
--# initializes Pointer;
is
  procedure Push(X: in Integer);
  --# global in out S, Pointer;
  --# derives S from S, Pointer, X &
  --#         Pointer from Pointer;
  procedure Pop(X: out Integer);
  --# global in S; in out Pointer;
  --# derives Pointer from Pointer &
  --#         X from S, Pointer;
end The_Stack;

package body The_Stack is
  Stack_Size: constant := 100;
  type Pointer_Range is range 0 .. Stack_Size;
  subtype Index_Range is
                Pointer_Range range 1 .. Stack_Size;
  type Vector is array (Index_Range) of Integer;
  S: Vector;
  Pointer: Pointer_Range;

  procedure Push(X: in Integer) is
  begin
    Pointer := Pointer + 1;
    S(Pointer) := X;
  end Push;

  procedure Pop(X: out Integer) is
  begin
    X := S(Pointer);
    Pointer := Pointer - 1;
  end Pop;
begin
  Pointer := 0;
end The_Stack;
```

The stack state variables S and Pointer are declared in the body of the package and Pointer is initialized. These internal variables are not directly accessible to users of the stack object. However, their existence and the existence of the initialization of Pointer are made visible to the Examiner for the purpose of analysis by the **own** and **initializes** annotations in the package specification just as the variable Seed of the package Random was made visible.

However, the above technique is not satisfactory since we have made visible considerable detail of the internal representation of the state of the machine, namely the existence of the individual variables S and Pointer. If at some later stage we need to change the implementation then there is a high risk that the specification will need to be changed because of the SPARK rules even though it would not need to be changed by the Ada rules. This would in turn give rise to tiresome dependencies since it would require all the calls to be reexamined and recompiled.

(A minor problem with the package as written is that when we come to use it we will get messages saying that S is being used before it is given a value. Of course we know that the dynamic behaviour is such that the initialization of S is unnecessary but the Examiner is not aware of this. Perhaps the best solution is simply to initialize S as well.)

3.4 Refinement

The problems of unnecessary dependencies can be overcome by using abstract own variables to provide what is known as refinement. An abstract own variable does not correspond to a concrete Ada variable at all but instead represents a set of variables used in the implementation.

As a consequence, an abstract own variable occurs in two annotations, the own variable clause in the package specification and then also in a refinement definition in the body giving the set onto which it is mapped.

The stack example could then be rewritten as

```
package The_Stack
--# own State;            -- abstract variable
--# initializes State;
is
  procedure Push(X: in Integer);
  --# global in out State;
  --# derives State from State, X;

  procedure Pop(X: out Integer);
  --# global in out State;
  --# derives State, X from State;
end The_Stack;

package body The_Stack
--# own State is S, Pointer;        -- refinement definition
is
  Stack_Size: constant := 100;
  type Pointer_Range is range 0 .. Stack_Size;
  subtype Index_Range is
                Pointer_Range range 1 .. Stack_Size;
  type Vector is array (Index_Range) of Integer;
  S: Vector;
  Pointer: Pointer_Range;

  procedure Push(X: in Integer)
  --# global in out S, Pointer;
  --# derives S from S, Pointer, X &
  --#             Pointer from Pointer;
  is
  begin
    Pointer := Pointer + 1;
    S(Pointer) := X;
  end Push;

  procedure Pop(X: out Integer)
  --# global in S; in out Pointer;
  --# derives Pointer from Pointer &
  --#             X from S, Pointer;
  is
  begin
    X := S(Pointer);
    Pointer := Pointer - 1;
  end Pop;

begin                       -- initialization
  Pointer := 0;
  S := Vector'(Index_Range => 0);
end The_Stack;
```

This enables the more abstract specification to be linked with the concrete body. The refinement acts as the link and says that the abstract own variable State is implemented by the two concrete variables S and Pointer.

Note moreover that the subprogram bodies have to have a refined version of their global and derives annotations (if provided) written in terms of the concrete variables.

One consequence of the refinement is that both Pointer and S have to be initialized because we have promised that the abstract variable State will be initialized. Of course, as mentioned earlier, we know that the dynamic behaviour is such that the initialization of S is unnecessary and we could omit it in practice and ignore the consequential message from the Examiner.

The various constituents of the refinement must either be variables declared immediately within the package body (such as S and Pointer) or they could be own variables of private child packages or of embedded packages declared immediately within the body. The process of refinement can be repeated since an own variable in the constituent list might itself be an abstract own variable of the child or embedded package.

It is worth summarizing some key points regarding the visibility of state variables of abstract state machines.

- The **own** annotation of an abstract state machine makes the existence of its state visible wherever the machine is visible.
- Annotations of subprograms external to a machine which (indirectly) read or update its state (by executing subprograms of the machine) must indicate that they import or export the machine state.
- Only the existence of the machine state (and its reading or updating) is significant in this context. The details can still be hidden by refinement.

The second point is important and states that annotations have to be explicitly transitive. Thus a procedure that calls Push and Pop also has to be annotated to indicate that it changes the state of the stack.

```
procedure Use_Stack
--# global in out The_Stack.State;
--# derives The_Stack.State from The_Stack.State;
is
begin
  The_Stack.Push( ... );
  ...
  The_Stack.Pop( ... );
  ...
end Use_Stack;
```

Finally note that one abstract state machine could be implemented using another abstract state machine embedded within it. Thus if a machine B is to be embedded in a machine A, this can be done by embedding the package representing B in the body of the package representing A. The state of B can then be represented as an item in the refinement. Alternatively the package representing B could be a private child of the package representing A.

Refinement of course relates to top-down design and provides a natural way of implementing such a design. It is especially important that refinement can be cascaded; this avoids a combinatorial explosion of visible data items which might otherwise occur especially in large programs. The key point is that it makes the existence of state known without giving away the details - the irrelevant detail is kept hidden.

3.5 The location of state

It is very important to ensure that state is located sensibly. In order to illustrate this first consider the following simple example

```
procedure Exchange(X, Y: in out Float)
--# derives X from Y &
--#         Y from X;
is
  T: Float;
begin
  T := X;  X := Y;  Y := T;
end Exchange;
```

The parameters X and Y have mode **in out**. This requires them to be both read and updated. The (optional) derives annotation in addition states that the final value of X depends upon the initial value of Y and vice versa. Note that the final value of X does not depend upon the initial value of X.

The scope of program objects should always be as restricted as possible. The rules of SPARK discourage the use of a global variable simply as a 'temporary store'. For example we might try to redefine the procedure Exchange so that the temporary T is global by writing

```
procedure Exchange(X, Y: in out Float)
--# global out T;
--# derives X from Y &
--#         Y from X;
is
begin
  T := X;  X := Y;  Y := T;
end Exchange;
```

But this is illegal because it violates one of several rules of completeness. The one that is violated here is that every variable mentioned in a global definition must be used somewhere in the dependency relation. We have to add T to the derives annotation thus

```
--# derives X from Y &
--#         Y, T from X;
```

and this forces us to admit that we actually change T. Moreover, flow analysis of a call of Exchange will reveal the use of T. Thus a succession of calls such as

```
Exchange(A, B);
Exchange(P, Q);
```

results in the following message from the Examiner

```
                Exchange(A, B);
                ^1
!!! (  1)     Flow Error    : Assignment to T is
              ineffective.
```

This is because the value of T produced by the first call of Exchange is overwritten by the second call without being used. Remember that analysis of the calls is done using only the abstract view presented by the specification and so the internal use of the value of T in the body is not relevant. Note further that this message will be produced even if the optional derives annotation is omitted.

Unnecessary state should thus be avoided. Indeed, the use of unnecessary state as in this example requires annotations for T on the subprogram calling Exchange and so on transitively. The annotations therefore cascade and so the use of unnecessary state is very painful and thereby discouraged.

But some state is necessary and we have seen how refinement may be used to ensure that although the existence of state in an abstract state machine must be made visible, nevertheless the fine details are properly hidden. (We can have our abstraction cake and still eat it!)

There is an interesting analogy between abstraction through refinement and the composition of records out of components. Consider a private type defining a position where the full type reveals the details in terms of *x*- and *y*-coordinates

```
type Position is private;

...

type Position is
  record
    X_Coord, Y_Coord: Float;
  end record;
```

Such a record type is sensible because the two coordinates are logically related; we can then consider a value of the type Position as a single entity which can be manipulated as a whole without knowing the details of its inner construction.

Refinement allows an abstract own variable to provide an external view of a more detailed set of variables within the package. Using the analogy to records, we should only use refinement to group together naturally related items. Thus the refinement of the variable State of the package The_Stack into the variables Pointer and S is appropriate.

4. PROOF

For some applications formal proof is a valuable technique for showing correctness. SPARK has comprehensive facilities for proof including the ability to develop proofs with refinement when there are two views of a state. In order to illustrate this it is necessary to explain some of the basic techniques involved.

4.1 The proof process

The general idea is that we state certain hypotheses which we assert are always satisfied when a subprogram is called (the *preconditions*) and we also state the conditions which we want to be satisfied as a result of the call (the *postconditions*). These conditions are given as further annotations in the subprogram specification. We then have to show that the postconditions always follow from the preconditions.

The Examiner processes the text and generates one or more theorems (conjectures really since they might not turn out to be true) which then have to be proved in order to show that the postconditions do indeed always follow from the preconditions. These theorems which are called *verification conditions* are often trivially obvious. If they are not then there are two tools which can be used. These are the Simplifier which carries out routine simplification and the Proof Checker which is an interactive assistant that enables the user to explore the problem and hopefully construct a valid proof.

In order for the proof tools to function correctly, they need to be aware of the various rules which can be used. For the predefined types these are built into the system but other rules can be provided as we shall see in a moment.

As a first example consider once more the procedure Exchange. There is no precondition since it is designed to work no matter what the values of the parameters happen to be. But there is of course a postcondition and so the procedure becomes

```
procedure Exchange(X, Y: in out Float)
--# derives X from Y &
--#         Y from X;
--# post X = Y~ and Y = X~ ;
is
  T: Float;
begin
  T := X;  X := Y;  Y := T;
end Exchange;
```

Note again the use of the tilde character with in out parameters; the decorated form indicates the initial imported value of the parameter whereas the undecorated form indicates the final exported value.

The verification condition generated by the Examiner for the procedure Exchange is

```
H1:   true .
       ->
C1:   y = y .
C2:   x = x .
```

The notation used is that there are a number of hypotheses (H1, H2, ...) followed by a number of conclusions (C1, C2, ...) which have to be verified using the hypotheses. Note that the conditions are written in a language known as FDL (Functional Definition Language) which has a strong mathematical flavour.

In this example there is no precondition and so effectively no hypotheses (this is represented as the single hypothesis H1 which is true). The two conclusions to be proved are that y = y and x = x which are reasonably self-evident and so it is pretty clear that the procedure Exchange is correct.

If we were stubborn and wanted to be completely confident then we could submit the above verification condition to the Simplifier which would reduce it to simply

```
*** true .        /* all conclusions proved */
```

Verification conditions often appear mysterious and not obviously related to the code; they are produced by a "hoisting process" whereby the postcondition is transformed backwards through the statements in order to arrive at the so-called weakest precondition; this is the condition that must hold at the start in order for the postcondition to hold. We then have to show that the weakest precondition follows from the given precondition. In the verification condition, the hypotheses correspond to the given precondition and the conclusions to be proved correspond to the weakest precondition. However, the details of the hoisting transformations need not concern us in this paper.

4.2 Loops

Significant computations usually have loops and these cause complexity in proving correctness. The problems arise because the code of a loop is usually traversed a number of times with different conditions.

The approach taken is to cut a loop so that the various parts can be treated separately. The cut is made by inserting an assert statement which gives conditions that are to be true at that point. The conditions can be thought of as postconditions for the sequence of code arriving at the cutpoint and as preconditions for the sequence going on from the cutpoint.

A simple example is provided by the following integer division algorithm which might be used on a processor without a hardware divide instruction.

```
procedure Divide(M, N: in Integer; Q, R: out Integer)
--# derives Q, R from M, N;
--# pre (M >= 0) and (N > 0);
--# post (M = Q * N + R) and (R < N) and (R >= 0);
is
begin
  Q := 0;
  R := M;
  loop
    --# assert (M = Q * N + R) and (R >= 0);
    exit when R < N;
    Q := Q + 1;
    R := R - N;
  end loop;
end Divide;
```

Each transversal of the loop adds one to the trial quotient and subtracts the divisor N from the corresponding trial remainder until the remainder first becomes less than the divisor. Clearly it only works if both M and N are not negative and also the divisor must not be 0; hence the precondition.

The postcondition has two parts. First the output parameters must have the appropriate mathematical relation implied by the division process and secondly the remainder must be less than the divisor and not negative, so we have

```
--# post (M = Q * N + R) and (R < N) and (R >= 0);
```

The choice of assertion is fairly obvious. As noted above, the final postcondition has two parts, the division relation and the upper and lower bounds on the remainder. All the loop does is keep the division relation true and reduce the remainder until it satisfies the upper bound (as well as keeping the lower bound satisfied). The assertion is simply that the division relation is true and that the remainder satisfies the lower bound; the exit statement is taken when the upper bound is satisfied as well. The initial statements before the loop are designed to ensure that the assertion is true when the loop is first entered.

There are therefore three sections of code to be verified. They are from the start to the beginning of the loop, around the loop, and from the loop to the end. The assert statement acts as the postcondition for the first section and as the precondition for the last section. It also acts as both precondition and postcondition for the loop itself; since it is unchanged by the loop it is often referred to as a loop invariant.

When the Examiner is applied to this subprogram, it produces verification conditions corresponding to the three sections. From the start to the assertion the verification condition is

```
H1:   m >= 0 .
H2:   n > 0 .
       ->
C1:   m = 0 * n + m .
C2:   m >= 0 .
```

Conclusion C2 is trivially obvious since it is just the hypothesis H1. Conclusion C1 is pretty obvious as well.

The verification condition for going around the loop from assertion to assertion is

```
H1:   m = q * n + r .
H2:   r >= 0 .
H3:   not (r < n) .
       ->
C1:   m = (q + 1) * n + (r - n) .
C2:   r - n >= 0 .
```

and that from the assertion to the final end is

```
H1:   m = q * n + r .
H2:   r >= 0 .
H3:   r < n .
       ->
C1:   m = q * n + r .
C2:   r < n .
C3:   r >= 0 .
```

In all cases the Simplifier reduces all the conclusions to true. It is also quite straightforward to show that they are true by hand – although perhaps a little tedious in the case of the loop itself which requires some manipulation. However, such trivial manipulation is prone to error if done by hand and the great advantage of the Simplifier is that it does not make careless mistakes.

Having shown that the verification conditions for the three separate sections of code are true it then follows that the procedure is correct. (To be honest we have only proved that it is partially correct; this means that it is correct provided that it terminates.)

In practice one does not bother to look at the unsimplified conditions and so the process is quite straightforward.

4.3 Proof functions

Annotations such as postconditions can be very expressive. Not only can we use the variables of the program but various other notations are also available. We have already noted the use of the tilde character to distinguish initial and final values of in out parameters. The following examples illustrate other possibilities.

```
type Atype is array (Index) of T;

procedure Swap_Elements(I, J: in Index;
                        A: in out Atype);
--# derives A from A, I, J;
--# post A = A~[I => A~(J); J => A~(I)];
```

The postcondition means that the final value of A is the initial value with elements I and J interchanged. Note carefully that it is the initial value of A that is referred to on the right hand side and so there are three uses of the tilde character.

```
function Max(X, Y: Integer) return Integer;
--# return M => (X >= Y -> M = X) and
--#          (Y >= X -> M = Y);
```

This illustrates that functions have return annotations rather than postconditions. The annotation should be read as return M such that if X >= Y then M is X and if Y >= X then M is Y.

```
function Value_Present(A: Atype; X: T) return Boolean;
--# return for some M in Index => (A(M) = X);
```

This function returns true if at least one component of the array has the value X. Remember that Index is the index type of the array type Atype.

```
function Find(A: Atype; X: T) return Index;
--# pre Value_Present(A, X);
--# return Z => (A(Z)) = X) and
--#      (for all M in Index range Index'First .. Z-1 =>
--#          (A(M) /= X));
```

This function returns the index of the first component of the array with the value X. Note the precondition which uses the previous function to ensure that such a value does exist. All Ada functions can be used in annotations in this way with any global variables being added as explicit additional parameters (remember the earlier remark that global variables can be looked upon as parameters that are always the same).

Sometimes, however, the functional nature of the annotation language is not rich enough in which case we can add our own so-called proof functions which do not exist as Ada functions at all.

As an elementary example consider the following implementation of the factorial function

```
--# function Fact(N: Natural) return Natural;

function Factorial(N: Natural) return Natural
--# pre N >= 0;
--# return Fact(N);
is
  Result: Natural := 1;
begin
  for Term in Integer range 1 .. N loop
    Result := Result * Term;
    --# assert Term > 0 and Result = Fact(Term);
  end loop;
  return Result;
end Factorial;
```

The approach we take is to introduce a proof function Fact which we can use in the annotations even though it is not defined in the Ada program text. An interesting observation is that although recursion is not permitted in SPARK because dynamic storage is forbidden, nevertheless proof rules can use recursion in their definition because proof is done offline independently of program execution.

The Examiner is now able to produce verification conditions; it does this without needing to know what the proof function Fact actually means because the process of producing verification conditions simply involves formal substitution.

There are four paths including one from start to finish which bypasses the loop in the case of N being zero. We will look at the verification conditions for just two of them. That from the assertion to the finish is

```
H1:   term > 0 .
H2:   result = fact(term) .
H3:   term = n .
       ->
C1:   result = fact(n) .
```

This is clearly correct by simply substituting from H3 into H2 irrespective of what Fact actually means. That from assertion to assertion is more interesting

```
H1:   term > 0 .
H2:   result = fact(term) .
H3:   not (term = n) .
       ->
C1:   term + 1 > 0 .
C2:   result * (term + 1) = fact(term + 1) .
```

In order to prove this we need a mathematical theorem for the Fact function namely

$$\text{fact}(n) = n \times \text{fact}(n\text{-}1) \qquad n > 0$$

The other two paths need the other obvious mathematical theorem

$$\text{fact}(0) = 1$$

In order to prove the verification conditions using the Proof Checker, it is necessary to give the Checker the rules corresponding to the above theorems. These can be expressed in the following form

```
rule_family fact:
  fact(X) requires [X : i] .

fact(1): fact(N) may_be_replaced_by
                                N * fact(N-1) if [N > 0] .
fact(2): fact(0) may_be_replaced_by 1 .
```

Given such rules the proofs can be entirely mechanized.

The reader might feel that this is all a bit of a cheat. However, the approach is typical of many safety-related mechanisms. Two routes to the solution are provided using entirely different technologies; one uses the Ada program and the other uses the annotations and proof rules. Since they agree we have a high degree of confidence in their correctness.

4.4 Proof and refinement

We are now in a position to return to the theme of abstraction and consider how we might add annotations for proof to the stack example.

We saw how we could have two views of the state of the package The_Stack – an external abstract view provided by the abstract variable State and an internal concrete view provided by the two variables S and Pointer. In order to develop proofs we need to map abstract conditions for the external view onto concrete conditions for the internal view. The package might become

```
package The_Stack
--# own State: Stack_Type;         -- abstract variable
--# initializes State;
is
  --# type Stack_Type is abstract;          -- proof type

  --# function Not_Full(S: Stack_Type) return Boolean;
  --# function Not_Empty(S: Stack_Type)
                                          return Boolean;
  --# function Append(S: Stack_Type; X: Integer)
                                      return Stack_Type;

  procedure Push(X: in Integer);
  --# global in out State;
  --# pre Not_Full(State);
  --# post State = Append(State~, X);

  ...   -- similarly Pop

end The_Stack;

package body The_Stack
--# own State is S, Pointer;         -- refinement definition
is
  ... -- etc as before

  procedure Push(X: in Integer)
  --# global in out S, Pointer;
  --# pre Pointer < Stack_Size;
  --# post Pointer = Pointer~ + 1 and
  --#          S = S~[Pointer => X];
  is
  begin
    Pointer := Pointer + 1;
    S(Pointer) := X;
  end Push;

  ...   -- similarly Pop plus initialization

end The_Stack;
```

The above omits the derives annotation partly for simplicity but also to emphasize that derives annotations are not necessary in order to develop proofs although we have shown them in earlier examples for completeness.

The abstract own variable State now includes a type announcement for the proof type Stack_Type. In developing the verification conditions, the Examiner converts this proof type into an FDL record type having two components corresponding to the variables S and Pointer. (Note again the strong analogy between refinement and record composition.)

There are also proof functions Not_Full and Append (with parameters of the proof type) which are used to give the pre- and postconditions for Push. The proof function Not_Empty is required for Pop.

Three verification conditions are generated for Push – one shows that the refined precondition follows from the abstract precondition, one shows that the abstract postcondition follows from the refined postcondition and the other (the usual one) shows that the refined postcondition follows from the refined precondition. The first is

```
H1:   not_full(state) .
H2:   s = fld_s(state) .
H3:   pointer = fld_pointer(state) .
        ->
C1:   pointer < stack_size .
```

The notation should be self-evident, H2 means that the refined variable S corresponds to the field s of the abstract State.

To complete the proofs we need proof rules for the proof functions in terms of the concrete variables such as

```
not_full(S) may_be_replaced_by
                          fld_pointer(S) < stack_size .
```

Given such rules the verification conditions can all be proved.

The stack package might be used by external procedures which themselves have proof annotations in terms of the proof functions. Of course they can only see the external view of the stack and so rules need to be developed in terms of that view. But the rules can themselves be proved using the concrete view.

5. DESIGN AND IMPLEMENTATION

One of the goals of this paper is to show that SPARK uses abstraction as a key ingredient in showing correctness. The important thing about abstraction is controlling the level of visibility. We are familiar in Ada with the idea of having more than one view of a type, for example the full view and the partial view of a private type. SPARK allows private types of course but as we have seen extends this idea of views to the representation of state through refinement. We have also seen how proofs may be developed around the two representations.

But it must not be thought that proof is the major goal of SPARK. The real goal is developing correct programs more cheaply and also of course convincing the customer that they are correct within a given budget. Sometimes formal proof is the appropriate tool to being convinced that the program is correct – but for most purposes it would be overkill.

But perhaps the real strength of SPARK is that it encourages good design by revealing the flow of information. For example, suppose we have a package Stuff which contains a procedure Do_It which in turn calls the procedures Push and Pop and thereby manipulates The_Stack. The Ada structure might be

```
package Stuff is
  procedure Do_It;
end Stuff;

with The_Stack;
package body Stuff is
  procedure Do_It is
  begin
    ...
    The_Stack.Push( ... );
    ...
    The_Stack.Pop( ... );
    ...
  end Do_It;
end Stuff;

with Stuff;
procedure Main is
begin
  Stuff.Do_It;
end Main;
```

By just looking at the procedure Main we have absolutely no idea what it does. Even if we look at the specification of Stuff we are none the wiser. We have to look at the body of Stuff to see that it has access to The_Stack. Clearly this is against the spirit of separation of specification and body. The specification ought to tell us what something does whereas the body should simply tell us how it does it. Of course the very fine detail is not always relevant but at least we ought to be clear about what is affected by looking at the specification.

Now consider the same example with the minimal SPARK annotations.

```
--# inherit The_Stack;
package Stuff is
  procedure Do_It;
  --# global in out The_Stack.State;
end Stuff;

with The_Stack;
package body Stuff is
  procedure Do_It is
  begin
    ...
    The_Stack.Push( ... );
    ...
    The_Stack.Pop( ... );
    ...
  end Do_It;
end Stuff;

with Stuff;
--# inherit The_Stack, Stuff;
--# main_program;
procedure Main
--# global in out The_Stack.State;
is
begin
  Stuff.Do_It;
end Main;
```

This introduces two more annotations. One is the inherit clause which is required on the specification of a package in order to give access to other packages. The other is the main program annotation. The global annotations now reveal that the state of the package The_Stack is being manipulated by the procedure Do_It and (transitively) by the main subprogram. The fine details of just what is being done to The_Stack are not revealed and indeed it is probably not necessary to know at this structural level.

But the key point is that the side effect of manipulating the state of the stack is revealed. The annotations encourage good design because a bad design will often have a lot of curious unexpected side effects which are embarrassingly revealed by the annotations. Changing the structure in order to reduce the complexity of annotations will simplify the design by increasing coherence and reducing unnecessary cross-coupling.

Design relates to the specifications of components and their interrelationships whereas implementation relates to their bodies. It is interesting to note that most SPARK annotations apply to specifications and this emphasizes that SPARK is primarily about encouraging good design which then in turn leads to correctness of implementation.

An important issue is scalability, that is the ability to cope with large programs as well as small ones. In this context it is important that refinement can be cascaded. Thus if a component C uses a subcomponent S such as the stack as implementation detail then this fact need not be revealed at the top level. The subcomponent S can be embedded in C or (equivalently) be a private child of C. The state of C can then be refined to include the state of S so that S becomes just an implementation detail.

Note carefully that the most benefit will be obtained from SPARK if it is used as early as possible in the design process. It can weed out poor design before energy is spent on implementation. Of course, SPARK is valuable at the implementation stage as well because it will statically detect many errors that the compiler cannot detect. Indeed, SPARK reaches parts of the program that other tools do not reach.

6. LEVELS OF USE

One of the beauties of SPARK is that it can be used at different levels according to the requirements of the project. The simplest level just requires visibility annotations such as global and own annotations. These alone enable the Examiner to detect a great many errors that cannot be found by the compiler and thus have to be found by the tedious process known as testing often at a later stage in the development process and thus both more expensive to find and to fix.

We know that a key strength of Ada is its strong typing which reveals errors that in a pathetic language such as C have to be found by testing. SPARK extends this capability of Ada by finding even more errors without testing.

At the lowest level of annotation, flow analysis detects many typical errors such as uninitialized variables (those read before being given a value), ineffective parameters (whose value has no effect on the outcome), overwritten values (values that are overwritten before being used), nonterminating loops, aliasing, and so on. In addition many of the errors that can be made in Ada (such as inadvertently using the wrong variable because a later declaration hides it) cannot occur in SPARK because of stricter naming rules.

The introduction of the derives annotation will give more detail of the interactions between components and analysis will then often reveal surprising cross-coupling indicative of poor design or coding errors.

Proof may be appropriate for algorithmic applications. Proof can be applied at several levels as well. This paper has described proof whereby the user is required to add proof annotations. Another option is to check for the absence of runtime errors such as those that arise from violating a bound of an array. Since the Examiner knows about the type model it can generate verification conditions which show the absence of such runtime errors without the user having to supply any annotations at all. Proof can be performed with or without the derives annotations so in fact there are really many levels at which SPARK can be used.

These different levels can be mixed up within a single program. The computational leaves of a system might be subject to proof, the derives annotation might be useful for intermediate subcomponents whereas the outermost part of the system might well have the lowest level of annotation. This is a big strength of SPARK; it can be seen as several tools rolled into one each appropriate to a different part of a project.

7. CONCLUSION

Abstraction has been the main theme of this paper. Good abstraction is about revealing relevant detail and hiding irrelevant detail. Plain Ada programs typically do not reveal all the relevant detail. But SPARK with its refinement capability can be used to reveal the detail that matters while keeping the irrelevant detail hidden.

The reader should be aware that this paper has only surveyed some of the capabilities of SPARK. Much has been omitted such as how to interface to external parts of a system. Further details will be found in [1] from which many of the examples given here have been taken and which includes a CD containing demonstration versions of the SPARK tools plus full documentation.

Finally it should be noted that SPARK is well-established and has been successfully used on many projects in a variety of application areas; see for example [2, 3, 4].

8. REFERENCES

[1] Barnes, J. G. P., High Integrity Ada - The SPARK Approach, Addison-Wesley, 1997.

[2] Chapman, R. C., Industrial Experience with SPARK, Proceedings of SIGAda 2000.

[3] Chapman, R. C., Hammond, J. A. R., King, S., and Pryor, A. A., Proof More Cost-Effective Than Testing? IEEE Transactions on Software Engineering, 2000.

[4] Croxford, M., and Sutton, J., Breaking Through the V and V Bottleneck, Proceedings of Ada in Europe Conference 1995, Lecture Notes in Computer Science 1031, Springer-Verlag, 1996.