The SPARK Way to Correctness is Via Abstraction

John Barnes

SIGAda, Laurel, November 2000

Our itinerary

A bit of background to SPARK.

Basic ideas of abstraction and refinement.

Enough proof to show that it works with refinement.

Back to basics for visibility and design.

Finishing remarks.

SPARK

Origins in research on program analysis done by Ministry of Defence in 1970s.

Extended by Southampton University in 1980s.

Now developed and supported by Praxis Critical Systems Ltd.

Has the form of a subset of Ada (83 and 95); subset chosen to enable rigorous analysis.

Additional information supplied as annotations which are Ada comments.

Programs are compiled by normal Ada compiler.

Programs are analysed by the SPARK Examiner.

Proof (optional) done by Simplifier and Proof Checker.

Used extensively for

Avionics, Railroads, Banking,

for systems where getting it correct matters.

Abstraction

Tell the whole truth - and nothing but the truth

(on a need to know basis).

Hide the irrelevant detail.

Reveal the relevant detail.

Ada interfaces do not tell us enough.

Simple example

What does this specification tell us?

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

Frankly - not much!

There is a procedure Add.

It has a parameter of type Integer.

But it doesn't say anything about what it does or to whom it does it.

It might print the date; It might subtract two fixed point numbers;

It might launch a missile; it might ...

Add the lowest SPARK annotation - visibility

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

Global annotation must mention all globals that Add accesses.

Mode information is stronger than in Ada.

In Ada, modes give permission to access.

In SPARK, modes state that values must be used or produced.

So now we know

Total will get a new value, nothing else will be changed.

Old value of Total will be used.

Value of parameter X will be used.

In summary, Add computes a new value of Total using its original value and X.

Add dependency annotation

The next level of annotation gives explicit dependency relations.

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

Adds no further information in this simple example

(because only one out parameter).

Think of globals as extra parameters where actual parameter is always the same.

Add proof annotation

Add the third level of annotation.

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

Postcondition explicitly says that the final value of Total is its initial value added to the value of X.

Note the tilde only used with mode in out

Total~ means initial value

Total means final value.

The specification is now complete - it tells the whole truth.

Specification is the Interface

Annotations are part of the subprogram specification.

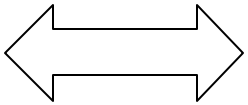
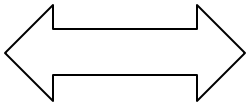
Since they are part of the interface.

Not generally repeated in body.

If no distinct spec then occur in body before "is".

Annotations separate interaction between caller and spec

from that between spec and implementation in body.

caller  **spec**  **body**

Examiner carries out two lots of checks.

Checks that all calls are consistent with spec.

Checks that body conforms to spec.

So if body of Add uses a global other than Total or misuses the modes, the Examiner will complain.

State

```
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
   Seed := 12345;
end Random_Numbers;
```

own annotation - announces Seed declared in body

used in annotations in spec.

initializes annotation - promises that it will be initialized at elaboration

could be in declaration of Seed rather than initialization part of package.

The existence of Seed (the state) is made known but full details are not revealed.

Note that no annotations in body.

Abstract State Machine

```
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;
```

Bad style.

The existence of S and Pointer are revealed.

Bad implications if implementation be changed.

Refinement

```
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

...-- as before

   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;

   ...-- Pop similarly

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

The abstract State is mapped onto S and Pointer.

Annotations on Push and Pop rewritten.

The existence of the State is now visible but the irrelevant details are kept hidden.

Analogy with records

Refinement is analogous to private types.

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

There are two views of the type Position. One shows the inner components.

There are two views of State. One reveals S and Pointer.

Transitivity and scalability

The consituents of refinement can also be in an embedded package or private child package.

Refinement can be repeated.

Key point

Annotations of subprograms external to a package which (indirectly) read or update its state (by executing subprograms of the package) must indicate that they import or export the state.

So

```
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;
```

Only the existence of the state (and its reading or updating) is significant in this context.

The details are hidden by refinement.

Location of state

Standard 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.

T is local and thus hidden. It does not occur in the annotations because it is hidden irrelevant detail.

Consider T as global

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

Note that derives annotation forces us to admit that we change T.

This leads us astray. Suppose we write

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

The Examiner then complains

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

Analysis of calls is done with external view. Internal use of T is hidden.

Avoid unnecessary global state. Annotations will cascade!

Proof

Consider Exchange again. Add postcondition.

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

The Examiner generates the following verification condition.

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

If the conclusions C1, C2, ... can be proved from Hypotheses H1, H2, ... then postcondition is satisfied.

Conclusions look OK!

Simplifier tool will simplify them and confirm

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

VCs created by a "hoisting process".

Loops

Loops have to be cut with an assertion (loop invariant).

```
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);    -- loop invariant
    exit when R < N;
    Q := Q + 1;
    R := R - N;
  end loop;
end Divide;
```

There are three paths - each has its own verification condition to be proved

from start to assert

from assert to assert around the loop

from assert to end.

Three VCs

H1: m >= 0 . **from start to assert**
H2: n > 0 .
->
C1: m = 0 * n + m .
C2: m >= 0 .

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

H1: m = q * n + r . **from assert to end**
H2: r >= 0 .
H3: r < n .
->
C1: m = q * n + r .
C2: r < n .
C3: r >= 0 .

All dead easy. Simplifier does them all automatically.

In practice we don't look at unsimplified VCs.

Other forms of conditions

```
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)];
```

Final value of A is the initial value with elements I and J interchanged.

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

Functions have return annotations rather than postconditions.

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

Returns true if at least one component of the array has the value X .

```
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));
```

Returns the index of first component of array with value X. Uses Value_Present in precondition.

Proof functions

A proof function is only used in annotations. It is not an Ada function.

Proof functions can recurse; Ada (SPARK) functions cannot.

```
--# 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;
```

Examiner produces VCs which use Fact without knowing what it means. From assert to end

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

Correct whatever Fact means.

Adding own rules

VC 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) .
```

To prove this we need a mathematical theorem for the Fact function

$fact(n) = n \times fact(n-1) \quad n > 0$

The other two paths need

$fact(0) = 1$

To prove the VCs using the Proof Checker, need to tell the Checker these rules.

```
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 .
```

The proof can then be 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.

Proof with refinement

```
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
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;
```

Note proof type Stack_Type.

For VCs, Examiner maps it into a record type with two components corresponding to S and Pointer.

Also proof functions Not_Full and Append (with parameters of the pro of type).

VCs for Push

Three verification conditions are generated for Push

one shows that the abstract precondition implies the refined precondition,

one shows that the refined precondition implies the refined postcondition,

one shows that the refined postcondition implies the abstract postcondition.

The first is

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

Need rules for proof functions in terms of 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.

Back to basics of abstraction and visibility

SPARK uses abstraction as a key ingredient in showing correctness.

Abstraction controls the level of visibility

- **familiar Ada private types,**
- **representation of state through refinement.**

Proof works with refinement.

Proof is not the prime goal of SPARK.

Real goal is developing correct programs more cheaply and

- **convincing the customer that they are correct within a given budget.**

Formal proof is sometimes useful but often overkill.

SPARK encourages good design by revealing the flow of information.

Information flow

Consider a package Stuff which contains a procedure Do_It.

Suppose Do_It 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;
```

Look at Main. What does it do? No idea at all!

Look at spec of Stuff. None the wiser.

Have to look at body of Stuff to discover that it messes about with The_Stack.

Not good.

Spec should say what it does.

Body should say how it does it.

SPARK version reveals all

Add 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;
```

Two more annotations.

Inherit clause required on package spec to access other packages.

Also main program annotation.

Global annotations reveal that The_Stack is being manipulated by Do_It and (transitively) by the main subprogram.

Fine details of what is being done to The_Stack are not revealed.

Side effect of manipulating state of the stack is revealed.

SPARK encourages good design

A bad design often has unexpected side effects which are revealed by annotations.

Changing the structure to reduce complexity of annotations simplifies design by increasing coherence and reducing unnecessary cross-coupling.

Design relates to specifications of components and their interrelationships
implementation relates to their bodies.

Most SPARK annotations apply to specifications and are about encouraging good design.

Use SPARK as early as possible. It weeds out poor design.

And then finds many implementation errors even without proof.

It statically detects many errors that the compiler cannot detect.

SPARK reaches parts of the programming process that other tools do not reach.

Summary

SPARK has several levels of operation

- **simple annotations detect flow errors with low effort**
- **derives annotations detect unexpected cross coupling**
- **proof annotations enable proof of key algorithms**
  - **Note: can prove absence of runtime errors without proof annotations.**

The various levels can be mixed in one program.

Overall goal is cost-effective reduction of risk.

See High Integrity Ada - The SPARK Approach, John Barnes with Praxis Critical Systems, Addison Wesley, 1997 (revised 2000). (Better still, buy one!)

The End