The Spark Approach

to

High Integrity Software

Regensburg

18 June 2009

John Barnes

What is software for?

Does the software drive the hardware?

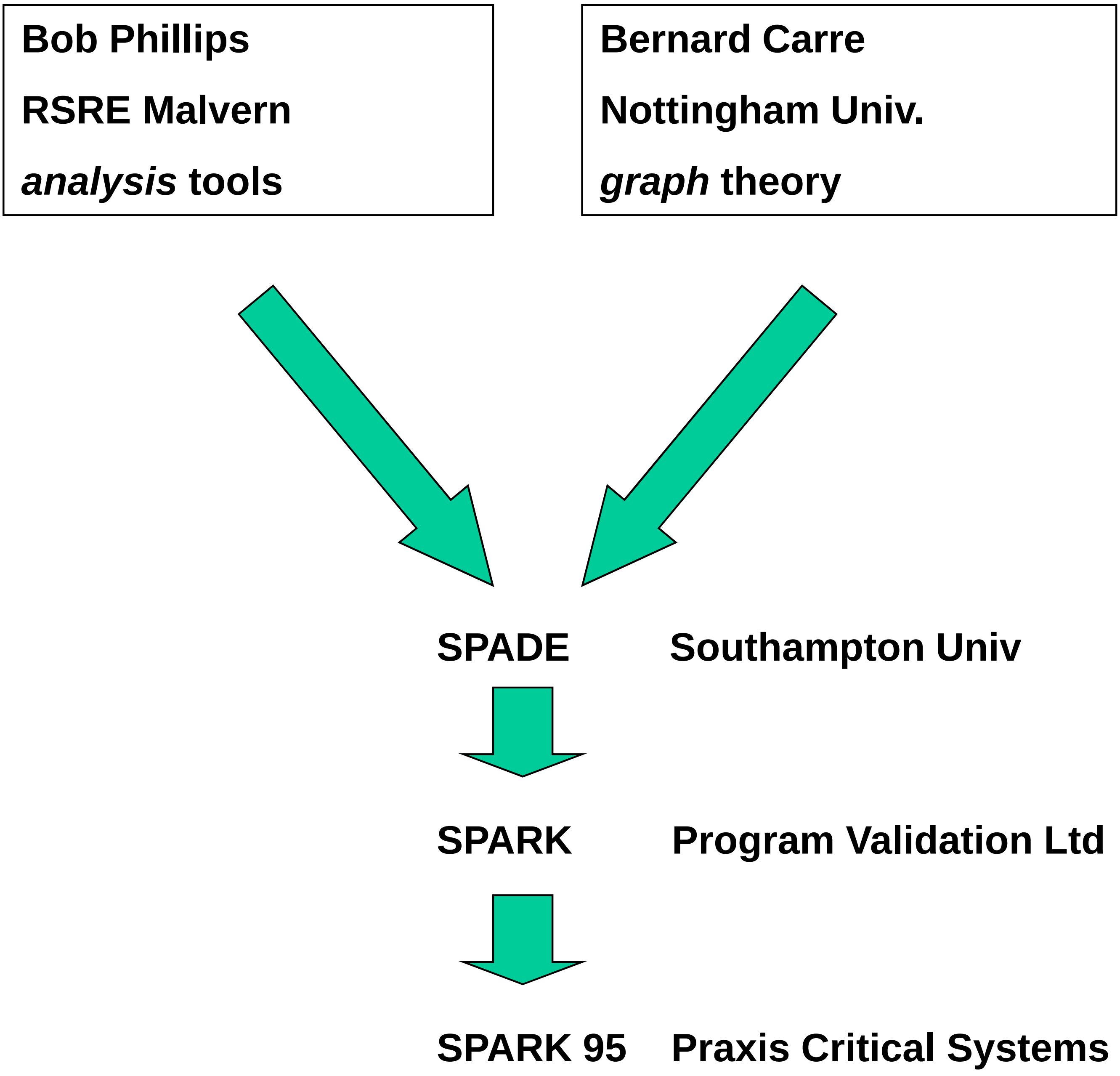
Or

Does the hardware implement the software?

We confuse these at our peril.

Must separate these concerns by suitable decomposition and well defined interfaces.

Evolution



Bob Phillips

RSRE Malvern

***analysis* tools**

Bernard Carre

Nottingham Univ.

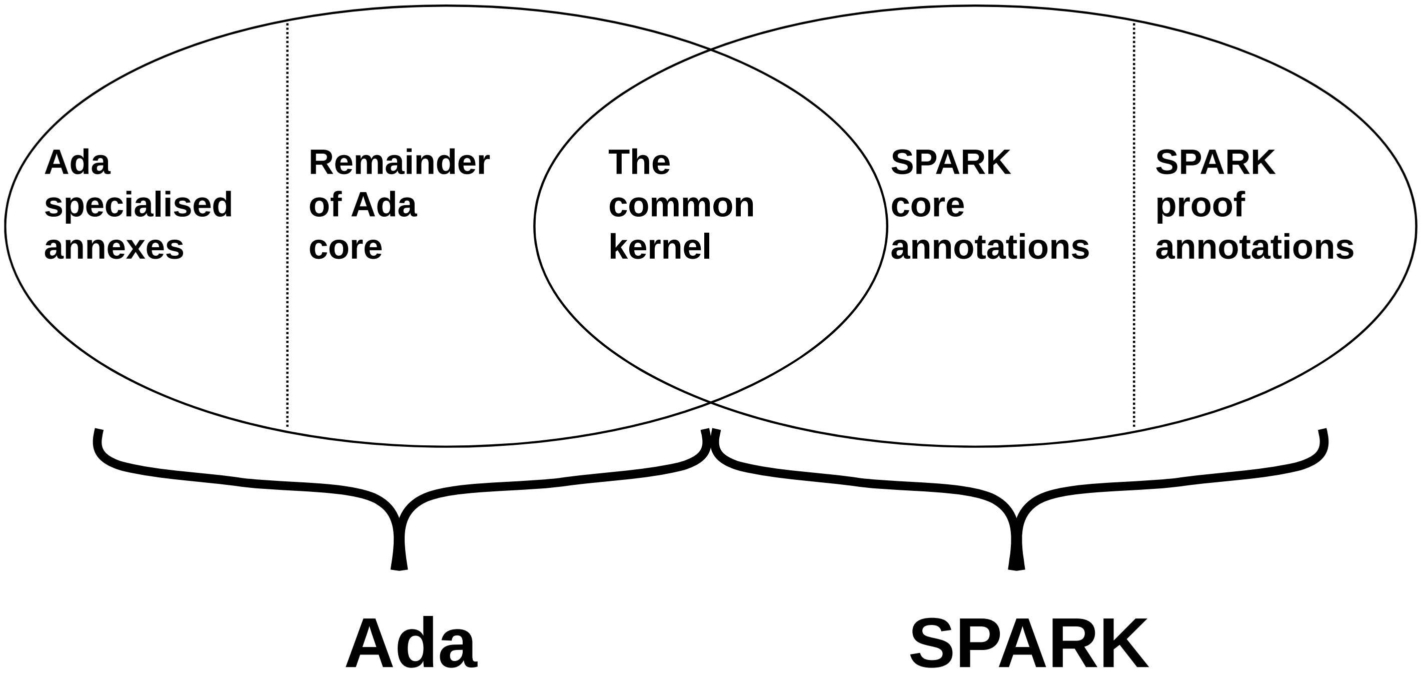
***graph* theory**

SPADE **Southampton Univ**

SPARK **Program Validation Ltd**

SPARK 95 **Praxis Critical Systems**

Spark and Ada



A subset of Ada plus annotations as comments;

subset chosen to enable rigorous analysis.

Programs usually compiled by normal Ada compiler.
Can also compile to C.

But Spark is really a language in its own right.

Programs are analysed by Spark Examiner.

Proofs (optional) done by Simplifier and Proof Checker.

Ada Example

An example of a simple stack of floating point numbers.

Implemented as an array of fixed length, say 100.

User can Push items on and Pop them off.

Also ask how many items on the stack.

Attempt to Push when full, or Pop when empty raises an exception.

There are three parts:

> Specification, view of the stack to the user,
>
> Body of stack, its implementation,
>
> User.

They can be compiled separately.

> But need spec in order to compile body and user.
>
> Body and user are independent.

Spec of Stack

```
package A_Stack is
   Stack_Error: exception;
   procedure Push(X: in Float);
   procedure Pop(X: out Float);
   function Number_On_Stack return Integer;
end A_Stack;
```

Gives enough information to be able to write code using the stack and for the compiler to compile calls of the procedures and functions.

Says nothing about how it is implemented.

Body of Stack

```
package body A_Stack is
   S: array(1 .. 100) of Float;
   Pointer: Integer := 0;

   procedure Push(X: in Float) is
   begin
      if Pointer = 100 then
         raise Stack_Error;
      end if;
      Pointer := Pointer + 1;
      S(Pointer) := X;
   end Push;

   procedure Pop(X: out Float) is
   begin
      if Pointer = 0 then
         raise Stack_Error;
      else
         X := S(Pointer);
         Pointer := Pointer - 1;
      end if;
   end Pop;

  function Number_On_Stack return Integer is
   begin
      return Pointer;
   end Number_On_Stack;
end A_Stack;
```

The User

```
with A_Stack;
procedure Main is
   A, B, C: Float;
begin
   A := ...;   B := ...;   C := ...;
   A_Stack.Push(A);
   A_Stack.Push(B);              -- mess about with the stack
   A_Stack.Pop(A);
   A_Stack.Pop(B);
   ...                                            -- and so on
exception
   when A_Stack.Stack_Error =       -- if stack goes wrong
      .....                   -- print some message perhaps
end Main;
```

More Briefly

If get fed up with writing A_Stack then more briefly

```
with A_Stack;  use A_Stack;
procedure Main is
   A, B, C: Float;
begin
   A := ...;   B := ...;   C := ...;
   Push(A);
   Push(B);                              -- mess about with the stack
   Pop(A);
   Pop(B);
    ...                                     -- and so on
exception
   when Stack_Error =>      -- if stack goes wrong
      .....                               -- print some message perhaps
end Main;
```

But this shorthand is not allowed in Spark because it violates an important principle.

Some Principles

Static storage:

no dynamic bounds,

no recursion

Unique names:

one name <-> one entity

no anonymous entities

No exceptions:

=> no run-time errors

Software Problems

Lots of kinds of software

Pocket calculators

Office stuff: word processors, spreadsheets

if they go wrong, tough luck, swear and start again

there is no specification, they do what they do

Serious programs: safety critical and security critical

if safety critical go wrong,
people die and/or environment is damaged

if security critical go wrong,
loss of national security, commercial reputation, theft

Serious programs have to be correct.

And seen to be correct.

Contracts

Serious programs have a specification

=> contract between ultimate client and software developer.

Contracts are not new.

Early programming libraries had specs (eg Algol 60 library in CACM).

Spec tells the user what parameters to supply, any constraints etc.

In essence there is a contract between user and writer of software.

> The user promises to provide suitable parameters and the subroutine promises to provide the right answer.
>
> If they both keep their promises then all is OK.

Contracts are in fashion but distinguish Static and Dynamic.

Program Decomposition

Break a program into parts, define what each part does.

This defines the interface to other parts.

We can then develop the parts independently.

IF each part obeys its side of the contract implied by the interface

AND

IF the interfaces are defined correctly

THEN

when we put the system together, it will work perfectly!

But

But it goes wrong - why?

Interface definitions are not usually complete:

holes in the contracts

Components are not correct or are not used correctly:

contracts are violated

Also, maybe, contracts are not for the right problem anyway!

Spark addresses these problems by providing techniques that help to ensure:

no holes in contracts
contracts are not violated.

General idea is

Correctness by Construction

That is, use techniques that prevent you from writing a wrong program in the first place.

Interface Definition

Interface definition should:

> **hide all irrelevant detail**
>
> **expose all relevant detail.**

Interface should be complete and correct.

Distinguish the interface from the implementation.

> **Details of implementation should not concern writer of interface.**

As we have seen:

> **Package specification defines the interface.**
>
> **Package body provides the implementation.**

But 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 says nothing 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 Minimal Spark Annotation - Visibility

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

Global annotation must mention all globals accessed by Add.

Mode information stronger than 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.

But we still are not assured that it will do addition.

Add Proof Annotation

Can add postcondition.

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

Postcondition explicitly says that 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 (strictly current value)

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

But ...

Should really add a precondition, such as

```
--# pre X + Total <= Integer'Last;
```

Don't need tilde because current value (at start) is initial value.

Proof is Static

These pre- and postconditions are checked before the program runs.

It's no good checking when the program runs.

Consider

```
procedure Touchdown( ... );
--# pre Undercarriage_Down;
```

Too late to be told as the plane lands.

Like bolting the door after the horse has bolted.

Note double use of "bolt".
Overloading is a problem in English.
Forbidden in Spark.

Eiffel has pre- and postconditions -
but only checked at execution time,
not much use.

Implementation

We have seen how to define the interface fully.

It is harder to ensure that implementation is OK. This leads into thoughts of how to ensure it is correct. (Debugging)

Four ways of finding errors:

1 by the compiler

2 at runtime by a language check

3 by deliberate testing

4 by program crashing

It is cheaper and more reliable to find errors earlier.

Ada is quite good at finding many errors at compile time.

It has a good syntax structure - another reason why Ada was a good base for Spark.

Consider

A piece of Ada:

```
type Signal is (Danger, Caution, Clear);
...
if The_Signal = Clear then
   Open_Gates;
   Start_Train;
end if;
```

The corresponding C might be

```
if (the_signal == clear)
{
   open_gates();
   start_train();
}
```

Consider typical typographical error
eg add semicolon at end of first line.

Ada program fails to compile - quite safe. Type 1 error.

C program compiles OK. But then program misbehaves.

It always opens the gates and sends train on its perilous journey.

Type 4 error (well, train crashes rather than program!)

Similarly replacing == by = also results in disaster.

Sooner the better

The sooner an error is found the better.

Cheaper to find and cheaper to put right.

Goal of Spark:

provide tools so that *all* errors can be found before the program executes.

Tools allow various levels of annotation and so various levels of analysis.

Also Derives Notation

Says for each output which inputs it depends upon.

```
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.

Three Levels of Annotation

Choose level according to analyses required.

- **visibility annotations (mandatory)**

  data flow analysis - direction of flow correct

- **derives annotations (optional)**

  information flow analysis - relationships correct

- **proof annotations (optional)**

  proof of correctness - values correct

Proof and derives annotations are independent:

proof can be done without adding the derives annotations.

Specification is the Interface

Annotations go with the specification.
Since they are part of the interface.

Not generally repeated in the body.

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

```
procedure Add(X: in Integer)
--# global in out Total;
--# derives Total from Total, X;
--# pre X + Total <= Integer'Last;
--# post Total = Total~ + X;
is
begin
   Total := Total + X;
end Add;
```

Annotations separate interaction between caller and spec from that between spec and implementation in body.

caller <=> spec <=> body

The Spark Examiner carries out two lots of checks:

Checks that all calls are consistent with spec.
Checks that body conforms to spec.

But never needs body in order to analyse calls.

Three Tools

The Examiner

checks conformance to Ada kernel

checks annotations

performs flow analysis

can generate Verification Conditions for proof

The Simplifier

simplifies verification conditions

Proof Checker

can be used to prove verification conditions

An Example

An odometer, records total distance and distance for trip

Trip distance can be reset to zero

Total distance cannot be reset.

```
package Odometer is
   procedure Zero_Trip;
   function Read_Trip return Integer;
   function Read_Total return Integer;
   procedure Inc;
end Odometer;
```

This describes the interface to a package called Odometer. The package encapsulates various procedures and functions (subprograms).

It just gives the interface to those subprograms but not the details of their code.

Odometer in Spark

Add Spark annotations to specification

```
package Odometer
--# own Trip, Total: Integer;
is

   procedure Zero_Trip;
   --# global out Trip;
   --# derives Trip from ;
   --# post Trip = 0;

   function Read_Trip return Integer;
   --# global in Trip;

   function Read_Total return Integer;
   --# global in Total;

   procedure Inc;
   --# global in out Trip, Total;
   --# derives Trip from Trip & Total from Total;
   --# post Trip = Trip~ + 1 and Total = Total~ + 1;

end Odometer;
```

The interface now describes the full details.

Spark is mostly about strengthening the definition of interfaces.

Odometer Body

```
package body Odometer is
   Trip, Total: Integer;
   procedure Zero_Trip is
   begin
      Trip := 0;
   end Zero_Trip;
   function Read_Trip return Integer is
   begin
      return Trip;
   end Read_Trip;
   function Read_Total return Integer is
   begin
      return Total;
   end Read_Total;
   procedure Inc is
   begin
      Trip := Trip + 1;  Total := Total + 1;
   end Inc;
begin
   Total := 0;  Trip := 0;
end Odometer;
```

No annotations in body.

The package body is quite unchanged.

Exchange

```
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 local variable T is not mentioned in the annotations.

It is hidden irrelevant detail.

Avoid global variables whenever possible.

Bad Exchange

Using unnecessary global T is sinful

```
procedure Exchange(X, Y: in out Float)
--# global ... T;              -- mode to be supplied by student
--# derives X from Y &
--#              Y from X;
is
begin
   T := X;  X := Y;  Y := T;
end Exchange;
```

This is illegal because T not used in derives annotation.

Have to write

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

Annotations regarding T will now permeate the program. Moreover now writing

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

results in

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

Remember Interfaces

Assignment to T is ineffective - for the external view presented by the abstraction.

Of course the assignment is effective internally but that is not the concern of the abstraction as presented by the interface.

Remember that the interface simply says

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

The analysis of the calls is done using just this information and never looks at the 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;
```

Note how details of internal state are revealed.

This is bad because the detail is irrelevant.

Promises that Pointer is initialized.

Body

```
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                                                 -- initialization
   Pointer := 0;
end The_Stack;
```

State is initialized in package initialization;

- this is before main subprogram is entered.

Refinement

```
package The_Stack
--# own State;
--# 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;
```

Indicates presence of hidden state

- but does not reveal irrelevant details.

State is an abstract variable - not an Ada variable.

Refined Body

```
package body The_Stack
--# own State is S, Pointer;                            -- refinement
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;
```

Annotations are rewritten in terms of concrete variables.

Both Pointer and S have to be initialized.

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.

Core Annotations

--# global permits access to global variables from within subprograms.

--# derives defines interdependencies between mports and exports of subprograms.

--# main_program indicates that a library subprogram is the main subprogram.

--# own announces variables declared within packages and thus having state.

--# initializes indicates that the given own variables are initialized before the main subprogram is entered.

--# inherit permits access to entities in other packages.

--# hide identifies a section of text that is not to be examined.

Correctness

General goal is to find bugs as early as possible.

Ada is better than C because more bugs are found at compile time.

Spark extends the concept by finding even more bugs statically and in some cases (using proof), finds all bugs statically.

A Spark program should not raise exceptions.

Considered harder to show correctness of exception handling than to show that exceptions are not raised.

Tasking_Error	**cannot arise in Spark**
Program_Error	**cannot arise in Spark**
Storage_Error	**storage requirements statically known**
Constraint_Error	**need to prove cannot be raised**

Use Examiner with Run Time Check option to show no Constraint_Error.

A Bug in Exchange

Consider error in last assignment

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

```
T := X;  X := Y;  Y := X;
 ^1
!!! (  1)  Flow Error        : Ineffective statement.
```

This tells us that the assignment to T is useless - the value of T is never used.

Other messages are

```
!!! (  2)  Flow Error        : Importation of the initial
          value of variable X is ineffective.
!!! (  3)  Flow Error        : The variable T is neither
          referenced nor exported.
!!! (  4)  Flow Error        : The imported value of X is
          not used in the derivation of Y.
??? (  5)  Warning           : The imported value of Y may
          be used in the derivation of Y.
```

Last two arise from mismatch with derives annotation. Are not produced if only data flow analysis is requested.

Proof Annotations

--# pre defines the preconditions for a procedure or function.

--# post defines the postcondition for a procedure.

--# return defines (explicitly or implicitly) the result of a function.

--# assert defines a predicate that is required to be true at that point; it forms the sole hypothesis for the following code.

--# check like assert but adds its conclusions to the existing hypotheses.

--# function declares a proof function whose meaning is given by distinct rules.

--# type declares a proof type to be used with own abstract variables.

Verification Conditions

The Examiner produces verification conditions. If these can be shown to be true then the postconditions are satisfied.

```
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;
```

VCs are

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

Have to show that Conclusions C1 and C2 follow from the Hypothesis H1.

Clearly OK. Simplifier reduces it to

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

Asserts and Loops

Division by subtraction

```
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;
```

All loops must be broken with an assert.

Three are three paths

1 from start to assert
2 from assert around loop to assert
3 from assert to end

Each path has its own VC.

NB assert provides sole hypothesis at the cutpoint.

Three Paths

From start to assert

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

Around the loop

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

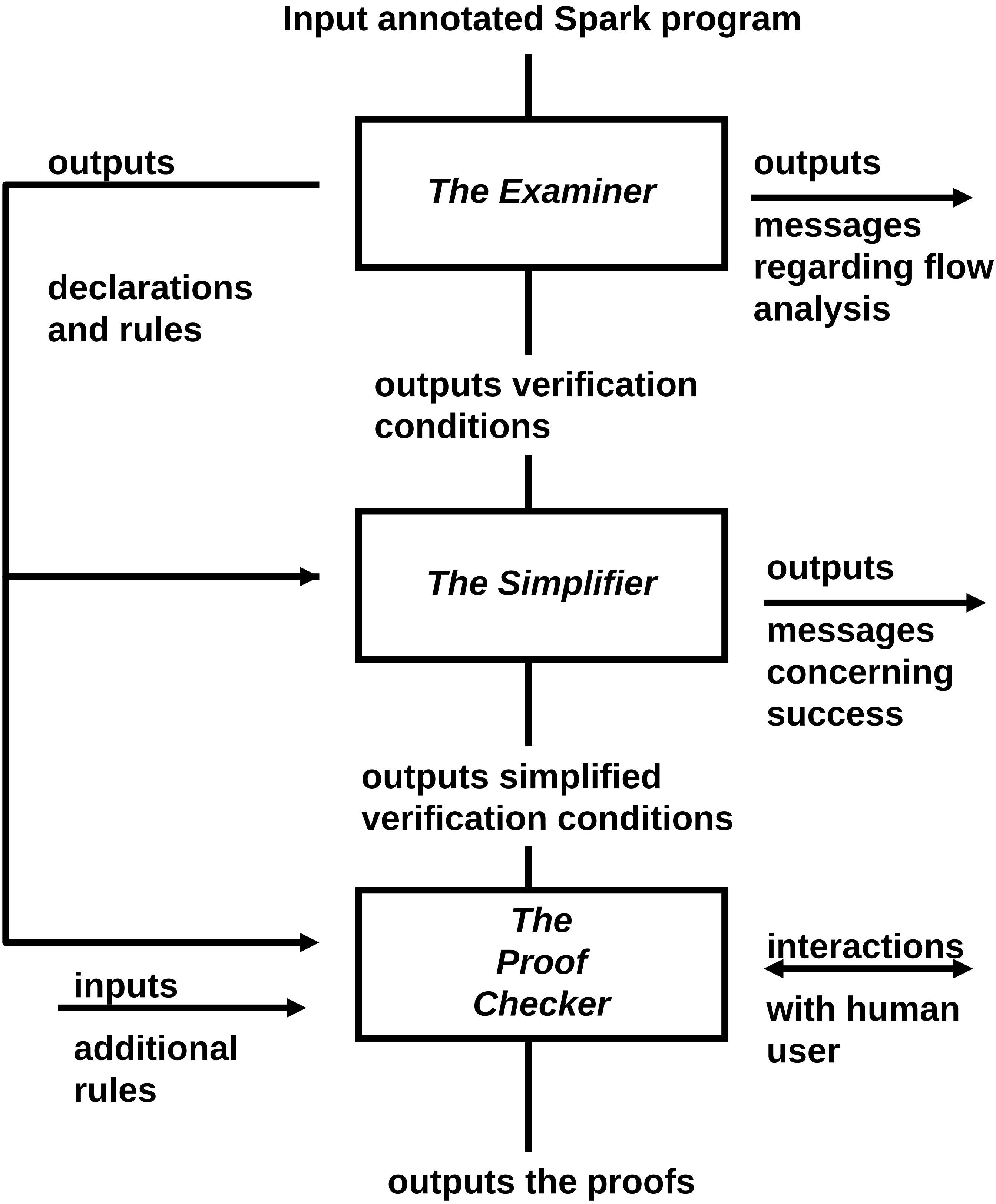
From assert to end

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

All easily reduced to true.

So procedure is correct - provided it terminates.

Proof Tools



Input annotated Spark program

The Examiner

outputs declarations and rules

outputs messages regarding flow analysis

outputs verification conditions

The Simplifier

outputs messages concerning success

outputs simplified verification conditions

The Proof Checker

inputs additional rules

interactions with human user

outputs the proofs

Rules

Proof tools use built-in rules such as

```
arith(3): X + 0 may_be_replaced_by X .

assoc(4): A * (B * C) may_be_replaced_by (A * B) * C .

distrib(1): A*(B+C) & A*B + A*C are_interchangeable .


transitivity(1): I <= K may_be_deduced_from
                                        [ I <= J, J <= K ] .

inference(2): X may_be_deduced_from [ Y -> X, Y ] .
```

We can add our own rules if we introduce our own proof functions.

Exercise

Consider

```
function Factorial(N: Natural) return Natural;
--# pre  N >= 0;
--# return Fact(N);
```

Write a suitable body.

Write some appropriate rules for the proof function.

Note: although recursion not permitted in Spark code, recursion is permitted in rules.

Answer

```
package body P is

   --# function Fact(N: Natural) retun 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;

end P;
```

rules are

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

Care

Test in loop must be precise

while Term < N loop is unfruitful

Verification conditions

There are four paths

For path(s) from start to assertion:

```
H1:   n >= 0 .
H2:   1 <= n .
       ->
C1:   1 > 0 .
C2:   1 * 1 = fact(1) .
```

For path(s) from assertion to assertion:

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

For path(s) from start to finish:

```
H1:   n >= 0 .
H2:   not (1 <= n) .
       ->
C1:   1 = fact(n) .
```

For path(s) from assertion to finish:

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

All easy to prove given

```
  fact(n) = n*fact(n-1)   n>0
  fact(0) = 1
```

What is wrong with

```
function Factorial(N: Natural) return Natural
--# pre N >= 0
--# return Fact(N);
is
   Result: Natuiral := 1;
   Term: Natural := 1;
begin
   loop
      Result := Result * Term;
      --# assert Term > 0 and Result = Fact(Term);
      exit when Term = N;
      Term := Term + 1;
   end loop;
   return Result;
end Factorial;
```

Verification conditions are all true.

Answer

Does not terminate when the parameter N is zero.

So only partially correct.

Must prove termination.

For loops always terminate.

Quantification

There are other forms allowed in conditions, for example

```
subtype Index is Integer range 1 .. 10;
type Atype is array (Index) of Integer;

procedure Zero(A: out Atype);
--# post for all M in Index => (A(M) = 0);
```

Sets every element of the array A to zero.

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

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

```
function Find(A: Atype; X: Integer) 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.

Ravenspark

Ravenscar - predicatable tasking subset

Ravenspark

- **allows multiple tasks at top level**
- **interaction via protected objects**
- **interrupt handling**

Enables programs to be written all in Spark without a cyclic executive.

Conclusions

Nearly done. And to finish.

Summary of the key issue.

Advert for my book.

Some applications.

Key Issue

Abstraction through well defined interfaces is the key.

Ada is perhaps unique in its separation of spec and body.

It distinguishes specification from implementation.

No other language seems to do that so well.

Specifications define interfaces.

Interfaces = Contracts

Proof is Static

Spark annotations (except some for proof) go on the specs and increase their detail.

Ada Hides Relevant Detail

Reconsider a package Stuff that contains a procedure Do_It. Suppose Do_It calls Push and Pop and so manipulates The_Stack. 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.

That is not good.

Spec should say what it does.
Body should say how it does it.

Spark Version Reveals All

Add minimal 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;
```

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

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

Side effect of manipulating state of the stack is revealed.

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.

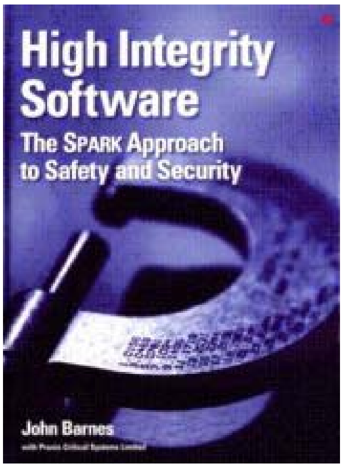
Overall goal is cost-effective reduction of risk.

- Use SPARK as early as possible. It weeds out poor design. And then finds many implementation errors without proof.
- It statically detects errors that a compiler cannot detect.

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

For further details of Spark see
High Integrity Software: The SPARK Approach to Safety and Security.
John Barnes and
Praxis Critical Systems Limited
Addison Wesley

ISBN 0-321-13616-0. (Better still, buy one!)



Applications – 1

In book

Lockheed Martin C130J

avionics control system (P)

Multos CA

secure system for credit cards (no P) (J)

Sholis

Ship/Helicopter Operational Limits Instrumentation System

SIL 4 (P), SIL 2 (no P) (J TA part)

Applications – 2

More recently and/or ongoing

BAE Australia & MBDA (Filton) - Australian ASRAAM Missile (fight control, auto-pilot etc)

SAAB Bofors & MBDA – Meteor Missile

Aermacchi – M346 Jet Trainer (primary flight control) Italian

BAE Systems UK

Hawk – new mission computers

Harrier – mission computers and stores management
0055 SIL 4 (P) (J)

Tornado – stores management (RTE P)

Eurofighter – all critical systems, including primary flight control (Spark 83 P?)

Tokaneer – security demonstrator (RTE & critical P) (J)

Applications – 3

New ones

QinetiQ Aberporth – range radar tracking and sensor data fusion

Rolls-Royce (all RTE P)

Trent 1000 FADEC

Trent 900 EMU (by Praxis)

Trent 1000 EMU (by Praxis)

Thales Wells – Watchkeeper mission planning and flight plan verification

Ansaldo Signal Australia – SIL4 railway interlocking

iFACTS – interim Future Area Control Tool System (by Praxis) (RTE+ P) (J)

Check it out

Check out

www.adacore.com

www.sparkada.com

Use and enjoy!