SPARK: An “Intensive Overview”

Rod Chapman

Praxis High Integrity Systems

Tutorial Outline

- The rationale of SPARK
- The core SPARK language
- Data and information flow analysis
- Lunch
- Design building blocks
- SPARK program design
- Formal verification
- Exception freedom
- Effective SPARK use

Tutorial Outline

• The rationale of SPARK	45 minutes
• The core SPARK language	90 " (with break)
• Data and information flow analysis	60 minutes
• Lunch	
• Design building blocks	60 "
• SPARK program design	45 "
• Formal verification	40 "
• Exception freedom	40 "
• Effective SPARK use	20 "

Tutorial Outline

- The rationale of SPARK
- The core SPARK language
- Data and information flow analysis
- Lunch
- Design building blocks
- SPARK program design
- Formal verification
- Exception freedom
- Effective SPARK use

Static Analysis Overview

- **Identifying properties of a program without execution**
  - style, coding standards, dubious construct detection
  - language subset conformance, wellformedness
  - control flow and complexity
  - data flow analysis
  - information flow analysis
  - proof (or formal verification)
- **An Ada compiler is a powerful static analyser**
- **Analysis:** shows that a program *should* work in *all* cases
- **Testing:** shows that it *does* work for certain *specific* cases

SPARK Goals

- **Precise static analysis**
- **Early use of static analysis**

- **Facilitated by:**
  - **an exact language**
  - **removal of ambiguous and erroneous constructs**
  - **annotations**

SPARK - Major Design Considerations

- **Logical soundness**
- **Simplicity of formal language definition**
- **Expressive power**
- **Security**
- **Verifiability**
- **Bounded space and time requirements**
- **Correspondence with ... Ada?**
- **Verifiability of compiled code**
- **Minimal run-time system requirements**

Principal Language Features

The kernel includes

- *packages,*
- *private types,*
- *library units,*
- *typed constants,*
- *unconstrained arrays,*
- *functions with structured results.*

The kernel excludes

- *unrestricted tasking,*
- *exceptions,*
- *generics,*
- *access types,*
- *goto statements,*
- *use clauses (except use type).*

Core Annotations

- *Global definitions* **declaring use of global variables by subprograms.**
- *Dependency relations* **of procedures, specifying information flow between their imports and exports.**
- *Inherit clauses* **to restrict penetration of packages.**
- *Own variable clauses* **to control access to package variables, and to define refinement.**
- *Initialisation specifications* **to indicate initialisations by packages of their "own" variables.**

These annotations are related to executable code by static-semantic rules, which are checked mechanically.

Why Annotations?

- **Annotations strengthen specifications**
  - **Ada separation of specifications/implementations too weak**
- **Allows analysis without access to implementations**
  - **which can be done early on during development**
  - **even before programs are complete or compilable**
- **Allows efficient detection of erroneous constructs**
- **Allows description of *system* behavior**

An example

```
procedure Inc (X : in out Integer);
--# global in out Callcount;
```

detection of function side-effect

```
function AddOne (X : Integer) return Integer is
  XLocal : Integer := X;
begin
  Inc (Xlocal);
  return XLocal;
end AddOne;
```

detection of aliasing

```
Inc (CallCount);
```

SPARK language features

In this language,

- **overloading is avoided as far as possible;**
- **scope and visibility rules are simplified;**
- **unconstrained array assignments are simplified;**
- **all constraints are statically determinable.**

SPARK texts can be rigorously analysed.

Static Program Analysis

- **This kind of analysis, performed on source code prior to program execution, is intended to check that**
  - **a program is “well-formed”, and**
  - **it performs its intended function.**

- **It can take several forms:**
  - **static-semantic checking (e.g. of strong typing, visibility),**
  - **flow analysis (of control, data and information flow),**
  - **semantic analysis (such as formal code verification).**

Control- and Data-Flow Analysis

- Control-flow analysis **reveals**
  - **unreachable or “dead” code,**
  - **code from which no exits are accessible,**
  - **“badly-structured code”, e.g. multiple-entry loops.**

- Data-flow analysis **reveals**
  - **conditional or unconditional use of undefined variables,**
  - **unused variable definitions,**
  - **loop-invariant definitions and redundant tests.**

Information Flow Analysis

- **Dependency relations of subprograms can be constructed easily in the course of software design.**
- **Information flow analysis of subprogram bodies checks that they have the required import/export dependencies.**
- **It also reveals program defects such as**
  - **data-flow errors (i.e. use of undefined variables),**
  - **ineffective imports and ineffective statements,**
  - **redundant tests, loop-invariant definitions, loop stability.**

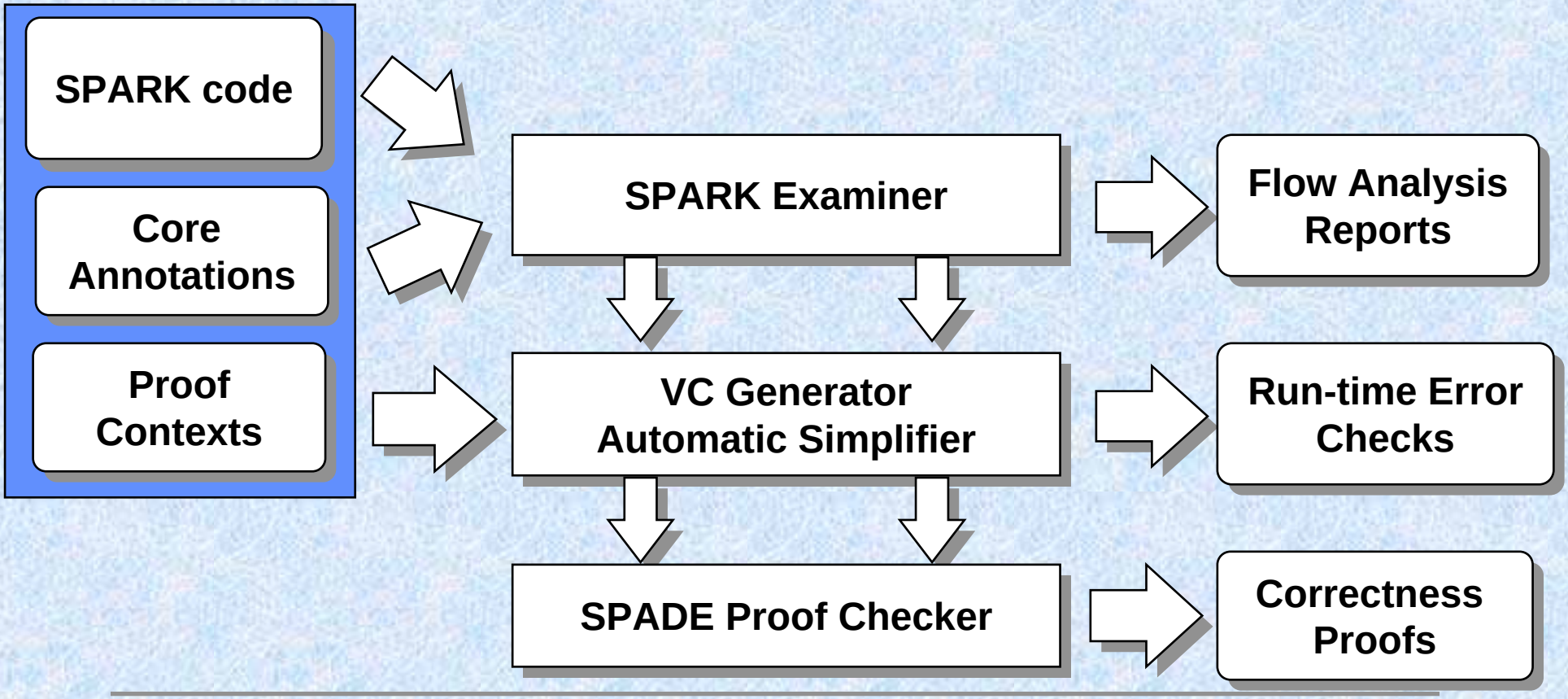
Semantic Analysis

- **formal code verification (against a complete requirement specification);**
- **proof of particular program properties, e.g. proof of absence of run-time errors.**

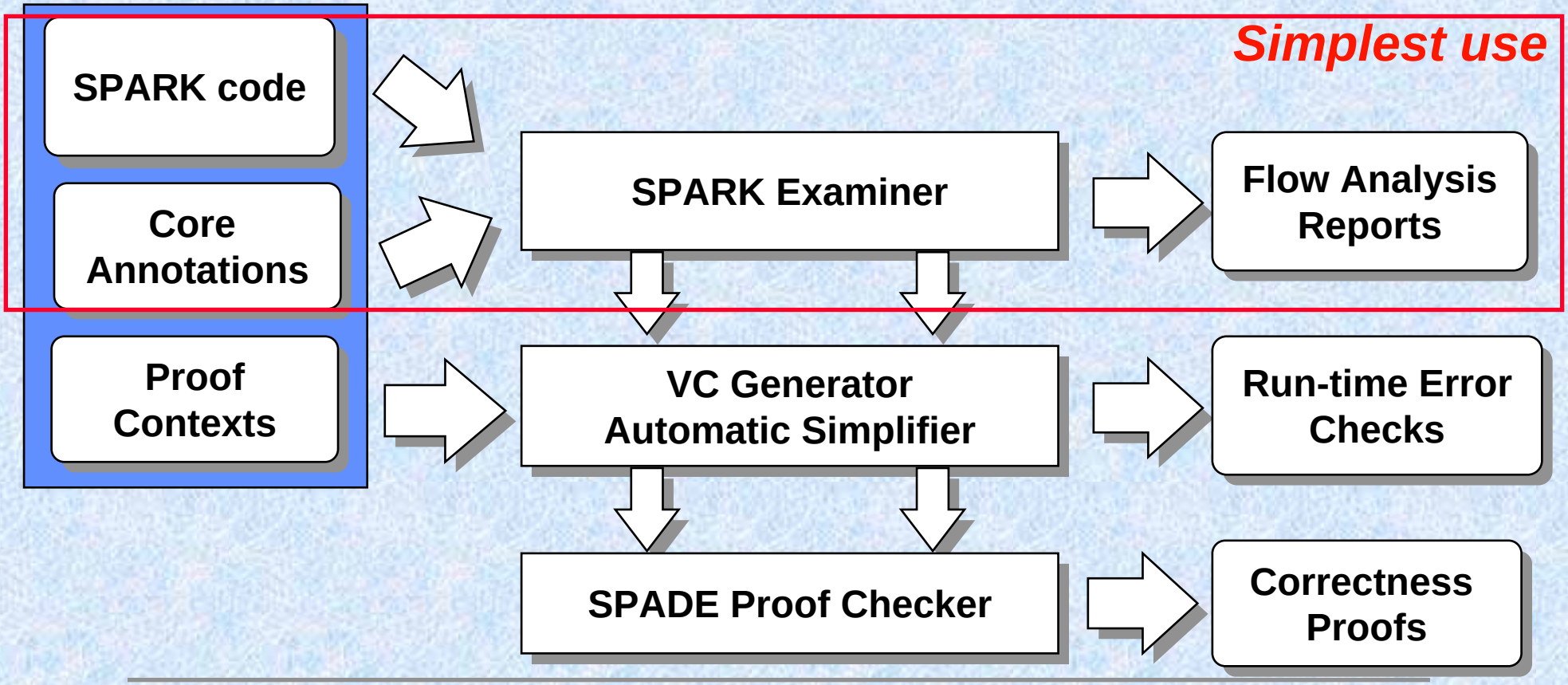
SPARK Proof Contexts

- Declarations **of proof types, constants, variables and functions.**
- Rewrite rules**.**
- Pre- and post-conditions **and invariants of subprograms.**
- Assertions **in subprogram bodies (e.g. loop invariants).**

SPARK Support Tools



SPARK Support Tools



Tool Support

- **The SPARK Examiner checks the conformance to the rules of SPARK.**
- **It checks consistency between Ada code and annotations by control-, data- and information-flow analysis of the code.**
- **The Examiner is written in SPARK. It has been applied to itself.**
- **A verification-condition generator is available.**
- **Verification conditions can be proved using the SPADE Proof-Checker.**

Elimination of an Insecurity

```
A : Vector(1 .. 1);
  ....
procedure P(V : Vector)
is
begin
  A(1) := V(1) + V(1);
  A(1) := A(1) + V(1);
end;
  ....
A(1) := 1.0;
P(A);
```

- **If parameter is passed by copying, the final value of A(1) is 3.0.**
- **If parameter is passed by reference, the final value of A(1) is 4.0.**

Elimination of an Insecurity

- **Using SPARK, this insecurity is detected:**

```
procedure P(V : Vector)
--# global A;
--# derives A from V;
is
begin
  A(1) := V(1) + V(1);
  A(1) := A(1) + V(1);
end;
  ....
A(1) := 1.0;
P(A);
  ^
*** Semantic Error : This parameter is overlapped by an
                     exported global variable.
```

Matrix Multiplication

```
procedure Multiply(X, Y : in     Matrix;
                   Z    : in out Matrix)
is
begin
   for I in MatrixIndex loop
      for J in MatrixIndex loop
         Z(I, J) := 0;
      end loop;
   end loop;
   for I in MatrixIndex loop
      for J in MatrixIndex loop
         for K in MatrixIndex loop
            Z(I, J) := Z(I, J) + X(I, K) * Y(K, J);
         end loop;
      end loop;
   end loop;
end Multiply;
```

Matrix Multiplication

```
procedure Multiply(X, Y : in     Matrix;
                   Z    : in out Matrix)
is
begin
   for I in MatrixIndex loop
      for J in MatrixIndex loop
         Z(I, J) := 0;
      end loop;
   end loop;
   for I in MatrixIndex loop
      for J in MatrixIndex loop
         for K in MatrixIndex loop
            Z(I, J) := Z(I, J) + X(I, K) * Y(K, J);
         end loop;
      end loop;
   end loop;
end Multiply;
```

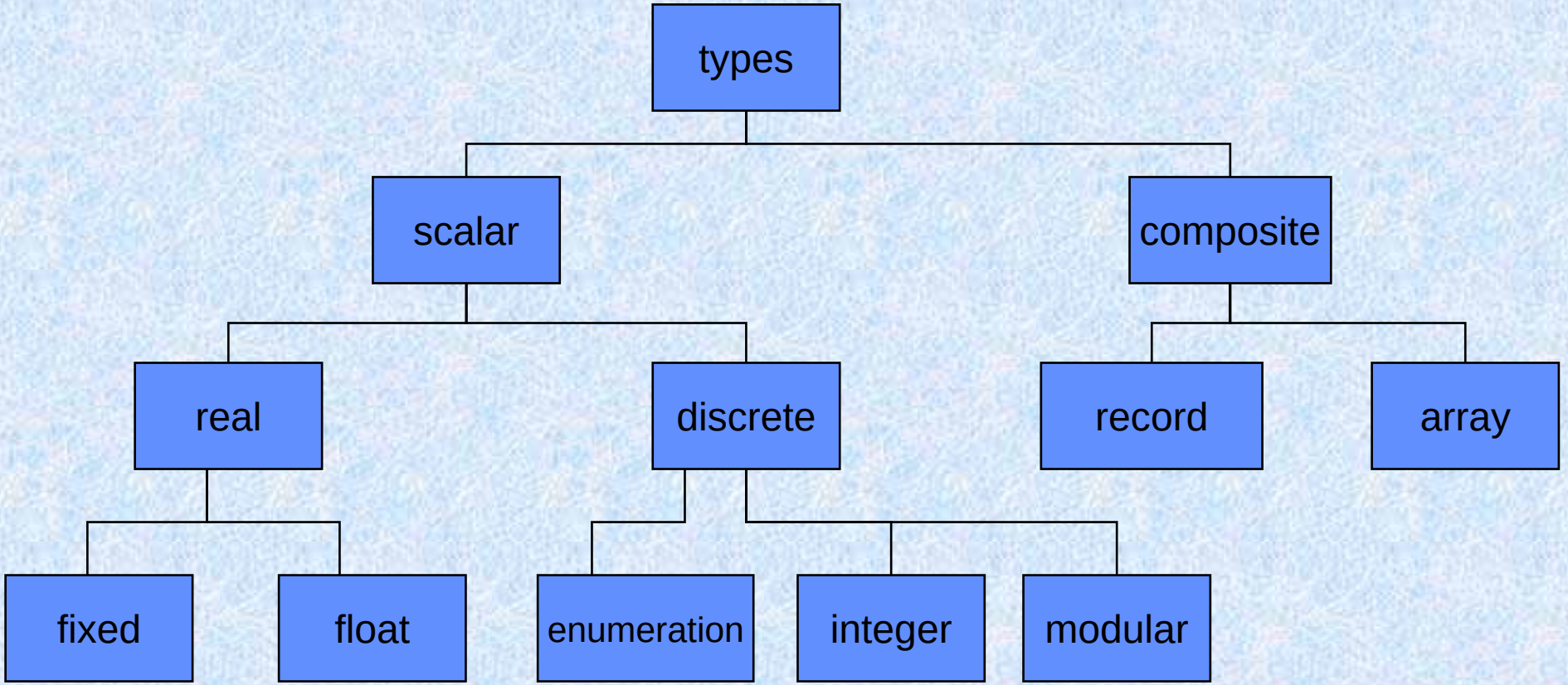
what is the value of `Multiply(A, A, A)` *?*

Tutorial Outline

- The rationale of SPARK
- The core SPARK language
- Data and information flow analysis
- Lunch
- Design building blocks
- SPARK program design
- Formal verification
- Exception freedom
- Effective SPARK use

The Type Model

The type model



General principles

SPARK uses a simplified version of the Ada type model:

- **all types in SPARK are explicitly named;**
- **all type constraints are known at compile time;**
- **only constant values can be given to objects at declaration;**
- **no redefining of infix operators**

Specifying ranges

In SPARK, static ranges must always be non-null. This definition is illegal:

```
type Empty is range 1 .. 0;
```

In SPARK, all range constraints (except those used to control *for* loops) must have static bounds: therefore this is illegal if *X* is a variable:

```
subtype T is Integer range 0 .. X;
```

Modular types

Modular types are available in SPARK 95 and are as Ada 95 except:

- **The Modulus of a type must be a positive power of 2.**
- **Unary operators are not defined for modular types.**

Enumeration types

SPARK forbids overloading, so the following is illegal:

```
type ValveState is (Open, Closed, Unknown);
type SwitchState is (On, Off, Unknown);
```

(Later sessions on design show how to make a virtue out of this restriction!)

The Boolean type

In SPARK, the Boolean type is not ordered.

So >, <, ‘Succ etc. are not permitted.

This greatly simplifies proof especially of run-time exceptions.

Real types

Use of the equality operator with floating point is discouraged (by Examiner nagging).

Fixed point types are as in Ada.

There are significant limitations on what can be formally proved about real number (especially floating point) operations.

Attributes

Most Ada attributes are permitted in SPARK.

The following standard attributes are not permitted because they either:

- **relate to excluded language features or**
- **could result in dynamic allocation, in which case the program would not be bounded in memory consumption.**

```
'address, 'callable, 'constrained, 'count,
'first_bit, 'image, 'last_bit, 'position,
'storage_size, 'terminated, 'value, 'width
```

Arrays (1)

The constraint on anonymous types extends to arrays, so both the index type and the element type of an array must be specified using named type marks. For instance:

```
type IllegalArray is array(1..10) of integer range 0..5;
```

must be defined as:

```
type Index is range 1..10;
subtype Element is integer range 0..5;
type LegalArray is array(Index) of Element;
```

Relational operators are not defined (except for strings). The logical operators are defined for arrays of Booleans.

Arrays (2) - unconstrained

Unconstrained array types are permitted, but only the following uses are allowed:

- **as a formal parameter to a subprogram;**
- **as an actual parameter (where the formal is unconstrained)**
- **referencing an element;**
- **application of relational operators to formal parameters of type string;**
- **taking `'first`, `'last`, `'range` and `'length` attributes of them.**

These limitations ensure that the space occupied by a program is bounded at compile time.

Records

In SPARK, records may not have discriminants or variant parts.

In line with the rest of the type system, all record fields must be of a named (sub)type.

Fields may not be given default values.

The only allowed operations on an entire record are assignment and equality.

Example:

```
type SpeedR is record
   Value  : SpeedValueT;
   Status : StatusT;
end record;
```

Tagged records and record extensions

- **No class-wide operations**
- **No abstract or controlled types**
- **Tagged types and type extensions may only be declared in the specification of library unit packages.**
- **At most one *root tagged type* or *type extension* may be declared in each package**
- **A subprogram declaration may not have the same name as a potentially inheritable subprogram unless it successfully overrides it.**
- **The ancestor part of an extension aggregate may not be a type mark.**

Declaring objects

Declarations of objects are as in Ada, except that every object must be given a named type mark.

Objects may be initialized at declaration, provided the initial value is either static, or is an aggregate containing static values.

This rule prevents dependency on elaboration order.

Aggregates

An aggregate must specify exactly one value for every component of the object.

In SPARK, an aggregate must be explicitly qualified with the object (sub)type.

Either positional or named association may be used, but they cannot be mixed in the same aggregate.

An `others` clause is permitted as the last choice in an array aggregate when using either positional or named association.

Expressions and Statements

Expressions

SPARK differs from Ada in that:

- **Slices are not allowed. This would lead to the introduction of anonymous subtypes.**
- **Operators cannot be called using prefix notation. That is they cannot be used as functions.**
- **Type conversion is more restrictive in SPARK.**
- **Initialization expressions must be *constant*.**
  - **known at compile time**
  - **no references to functions, unconstrained array types or variable names**
  - **does not contain array indexing or record field selection**

Statements and control flow

Statements and control flow

SPARK allows most of the Ada statements and control constructs.

The main differences are that `goto` is prohibited and that the use of `return` and `exit` is restricted.

The following slides

- **describe the rationale behind why certain constructs are prohibited in SPARK,**
- **summarise what is allowed by SPARK.**

Rationale

The restrictions on `goto`, `return` and `exit` ensure that the control flow graph of a SPARK subprogram obeys the rules of a "semi-structured flow graph" (SSFG) grammar, thus they have a form amenable to flow analysis. (see "Graph grammars and global program flow analysis" by Farrow,R.,Kennedy, K. and Zucconi, L., Proc. 17th Annual IEEE Symposium on Foundations of Computer Science, 1975).

This flow analysis means that the Examiner can check correspondence between annotations and code: this is necessary to overcome some of the insecurities in Ada. Checking annotations against code also helps during program development.

The restrictions also make SPARK programs more readable without significantly reducing the expressive power of the language.

Overview

There are some restrictions on expressions in assignment statements (e.g. no “sliding” on array assignments).

The `if` and `case` statements are unchanged from Ada.

Block statements and `goto` are prohibited in SPARK.

There are restrictions on the parameter specification in a `for` loop. All other loops are unchanged from Ada.

There are significant restrictions on the places where `exit` and `return` statements may be used.

`return` *statements*

- **Procedures may not contain `return` statements**
- **Functions must contain exactly one `return` statement:**
  - **this must include an expression**
  - **and must be the last statement in the function.**
- **These restrictions ensure that flow analysis can be performed.**

`for` *loops*

The parameter specification of a `for` loop must contain an explicit type mark: this is in line with prohibiting anonymous subtypes.

Ada permits the following:

```
for I in 1 .. 10 loop
```

whereas SPARK requires

```
for I in Integer range 1 .. 10 loop
```

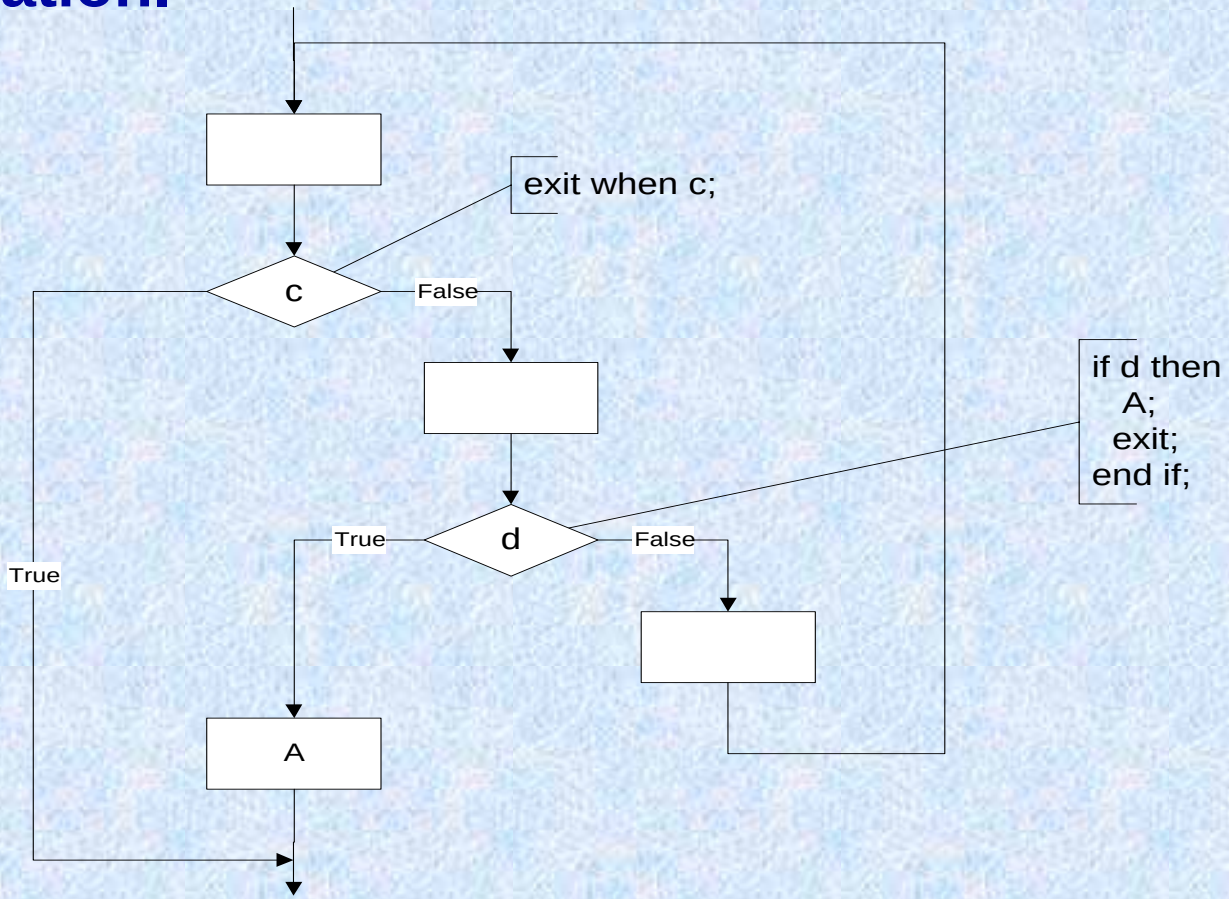
`exit` *(1)*

The restrictions on *`exit`* statements ensure that all the conditions controlling exit from the loop are traversed in every complete loop iteration. This ensures that flow analysis of the loop is possible.

- **An *`exit`* statement always applies to its innermost enclosing loop. (It may have a loop name for documentation purposes).**
- **If an *`exit`* statement contains a *`when`* clause then its closest containing statement must be a loop statement.**
- **If an *`exit`* statement does not contain a *`when`* clause then its closest containing statement must be an *`if`* statement, which has no *`elsif`* or *`else`* clauses and whose closest containing statement is a loop statement. This allows for a 'last wish' before exiting the loop.**

`exit` *(2)*

Every *`exit`* condition must be traversed in each loop iteration.



Non-terminating Loops

- **The main control loop of a SPARK prgoram may be required to run continuously**
- **A plain loop:**

```
loop
   ControlAlgorithm;
end loop;
```

  may be used for this purpose

- **A plain loop must be:**
  - **the last statement,**
  - **at the outermost scope,**
  - **of the main subprogram**
- **For analysis purposes a plain loop behaves as if its last statement is `exit when False;`**

Subprograms

Overview

- **SPARK has a system of core formal comments or annotations which simplify the Ada visibility rules and allow the Examiner to check declared information flow against the actual information flow.**
- **There is a further set of annotations to support formal verification.**
- **SPARK functions may not have side effects.**
  - **There is a sharp distinction in SPARK between procedures and functions: functions *inspect*, procedures may *change*.**
- **Recursion is prohibited.**
- **There are simplifications and restrictions on the specification of subprogram parameters: these clarify the code and prevent unwanted aliasing effects.**

Annotations

Subprogram annotations in SPARK come in three parts:

- **the** global **annotation: this makes visible any global variables accessed by the subprogram.**
- **the (optional)** derives **annotation: this specifies the information flow between the parameters (and global variables) of a procedure; it also specifies the import/export mode for each global.**
- **the** constraint **annotations: these are used to specify formal pre- and post- conditions for formal verification - these are only necessary if the subprogram is to be formally verified.**

The `global` *annotation - definition*

This has the form:

```
--# global {[mode] Variable_Name {, Variable_Name} ;}
```

Note that it may be split across multiple lines, as long as each line commences `--#`.

All variables referenced by a subprogram which are not in its formal parameter list must be mentioned in the `global` annotation. This ensures that complete flow analysis is possible and that subprograms can be analysed independently.

Global variables can be viewed as formal parameters for which the actual parameters are always the same.

The `global` *annotation - restrictions*

A variable may only appear once in any `global` annotation.

A name cannot appear both in the `global` annotation and as a formal parameter of a subprogram.

A variable named in a `global` annotation may not be redeclared immediately within the procedure. For example, in the following, the declaration of *`I`* is illegal: another name must be used.

```
procedure Inner(K: in integer)
--# global  in out I;
--# derives I from I, K;
is
  I : integer;
  ...
```


Functions

- **A function has an implicit dependency relation: it derives its return value from all its imports (parameters and / or global variables).**
- **Functions** never **have derives annotations.**
- **Function global annotations may have mode** *in* **(although modes are usually omitted).**
  - **a consequence of these rules is that functions cannot have side effects.**

Example:

```
function StackIsEmpty return Boolean;
--# global Stack;

function StackIsEmpty return Boolean;
--# global in Stack;
```

Syntax of derives annotations

This has the form:

```
--# derives [dependency_clause {& dependency_clause }];
```

where

```
dependency_clause ::=
   entire_variable_list from [imported_variable_list]

entire_variable_list ::= variable_name {,variable_name}

imported_variable_list ::= [*]|[*,]entire_variable_list
```

The `derives` *annotation (1)*

Not all the exports of a subprogram will be derived from the same set of imports: `&` is used to specify different derivations:

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

This means that the final value of `X` is derived from the initial values of `X` and `Y`, while the final value of `Z` is derived from the initial values of `Z` and `Y`.

If several exports are derived from the same set of imports then the following abbreviation can be used:

```
--# derives X, Y from Z, W;
```

This means that the final values of `X` and `Y` are (both) derived from the initial values of `Z` and `W`.

The `derives` *annotation (2)*

Where the final value of a variable depends upon its initial value then an asterisk can be used as an abbreviation.

```
--# derives X, Z from *, Y;
```

is equivalent to:

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

A variable may appear in the import list implicitly as well as explicitly via the asterisk abbreviation:

```
--# derives X, Z from *, X, Y;
```

is equivalent to:

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

The `derives` *annotation (3)*

There are various rules concerning the form of derives **annotations. In summary:**

- **All the variables in the global annotation and the parameter list must appear in the derives annotation.**
- **The import/export status of variables in the derives annotation must be compatible with the modes of those in the parameter list.**
- **A name cannot appear more than once as an exported variable in a derives annotation.**
- **A name cannot appear more than once in the same imported variable list.**
- **the derives annotation must be consistent with the information flow relations of the subprogram body - if this body exists, and is not hidden.**

The `null` derives

- Sometimes, under circumstances to be covered in the Interfacing lecture, we want to express the idea that a subprogram imports values but no visible export is derived from them.
- This can be expressed thus:

```
--# derives null from X, Y, Z;
```

- If there are some actual exports, the null clause must appear last:

```
--# derives A    from B &
--#         C    from D &
--#         null from X, Y, Z;
```

- The null derives is an unusual construct with uses in interfacing and where activities are deliberately moved “outside the SPARK boundary”.

Procedures

- **A procedure** may **have a derives annotation if information flow analysis is required.**
- **A procedure may have global annotations.**

Examples:

```
procedure BusyWait;
--# derives;

procedure Push(Value : in Integer);
--# global in out Stack;
--# derives Stack from Value, Stack;
```

Where to annotate subprograms

If a subprogram has a separate declaration the annotation goes with the declaration.

An annotation at declaration immediately follows the semicolon which ends the declaration (though intervening comments are permitted).

An annotation on the subprogram body goes between the subprogram specification and the word `is`.

Subprogram calls

The following rules improve the clarity of subprogram calls and remove possible sources of confusion in what they mean:

- **Actual parameter lists may use either named or positional association but not both.**
- **Default parameter values may not be specified: every call of every subprogram must specify every parameter exactly once.**

There are also rules to prevent aliasing and recursion.

Preventing aliasing of exports (1)

- **“Aliasing” occurs when the same variable (or component of a composite variable) is known by more than one name at the same place in the program.**
- **The aliasing of a subprogram’s exports can make the subprogram “erroneous”; it also makes the meaning of the code obscure.**
- **The rules of SPARK ensure that the aliasing of exports cannot occur.**

Preventing aliasing of exports (2)

For every procedure call statement, the rules are:

- **If a variable, V, appears in the global list of a procedure and is exported then neither V nor any of its subcomponents can be an actual parameter.**
- **A variable which occurs as an actual parameter of a procedure whose corresponding formal parameter is exported may not appear in the global list of that procedure.**
- **If a variable V occurs as an actual parameter whose corresponding formal parameter is exported, then neither V nor any of its subcomponents can appear as another actual parameter of the procedure in the same statement.**

Preventing recursion

In the general case, it is impossible to determine whether a program containing recursive calls will be bounded in either time or space. For this reason SPARK does not allow recursion.

The following rules concerning subprogram calls and declaration, combined with rules on package inheritance, prevent recursion:

- **Subprograms may only be declared separately from their bodies (or body stubs) in package specifications.**

- **In SPARK every call of a subprogram, in the compilation unit in which its proper body or body stub is declared, must follow the declaration.**

Packages

Packages

SPARK packages are much the same as in Ada.

The package rules of SPARK:

- **clarify the meaning of programs by simplifying the Ada visibility rules;**
- **ensure complete flow analysis is possible;**
- **prevent circularity of `with` clauses (to prevent recursion).**

Overview

- **SPARK makes a few restrictions on where package and subprogram declarations are allowed.**
- **Three new annotations are introduced: `inherit`, `own` and `initializes`.**
- **Package initialisation is simplified.**
- **Private types are much the same as in Ada.**
- **Renaming is permitted only in very restricted circumstances.**

Package structure

- **A subprogram declaration is only allowed inside the visible part of a package specification (not the private part).**
- **A package specification is only allowed as a library unit or nested immediately within a subprogram body or package body.**

Visibility

- **The *`with`* clause is required exactly as in Ada.**
- **The *`use`* clause is prohibited: all entities (except operators) from other packages must be referred to using the dot notation.**
- **In SPARK 95 a *`use type`* clause may appear**
- **The *`inherit`* clause controls access to global entities outside packages.**

`inherit` *clauses*

The *inherit* **annotation takes the form:**

```
--# inherit PackageName {, PackageName} ;
```

and must always appear immediately before the specification of the package to which it relates (following any with **clause).**

- **A package *P* is said to inherit a package *Q* if the inherit clause of *P* contains the simple name *Q*.**
- **A package P can only inherit a package Q if**
  - **the declaration of P is within the scope of Q, and**
  - **every package whose body contains the declaration of P, but not that of Q, inherits Q.**

`inherit` *clauses (2)*

- **A name in a package P can only denote an entity declared outside the declarative region of P if it is**
  - **a package inherited by P, or**
  - **an entity declared in the visible part of a package which is inherited by P, or**
  - **an entity declared in the private part or body of a package which is inherited by P and encloses P.**
  - **an own variable of a package which is inherited by P (when its name occurs in an annotation within P).**

- **A package P must inherit package Q if**
  - **subprograms in P reference state from Q (either directly or indirectly);**
  - **entities declared in the visible part of Q are used in P.**

Inheritance

- **The inherit annotation is used to announce dependencies on other packages**
- **When needed it always appears on the package spec, and not the body, even if inherited entities are used only in the body**
  - **the reason is similar to the need to announce own variables at the spec level; visible subprogram specifications will need to refer to state inherited from within the body**
- **The example illustrates the inherit annotation, and the transitive nature of the derives annotation**

Example of Inheritance

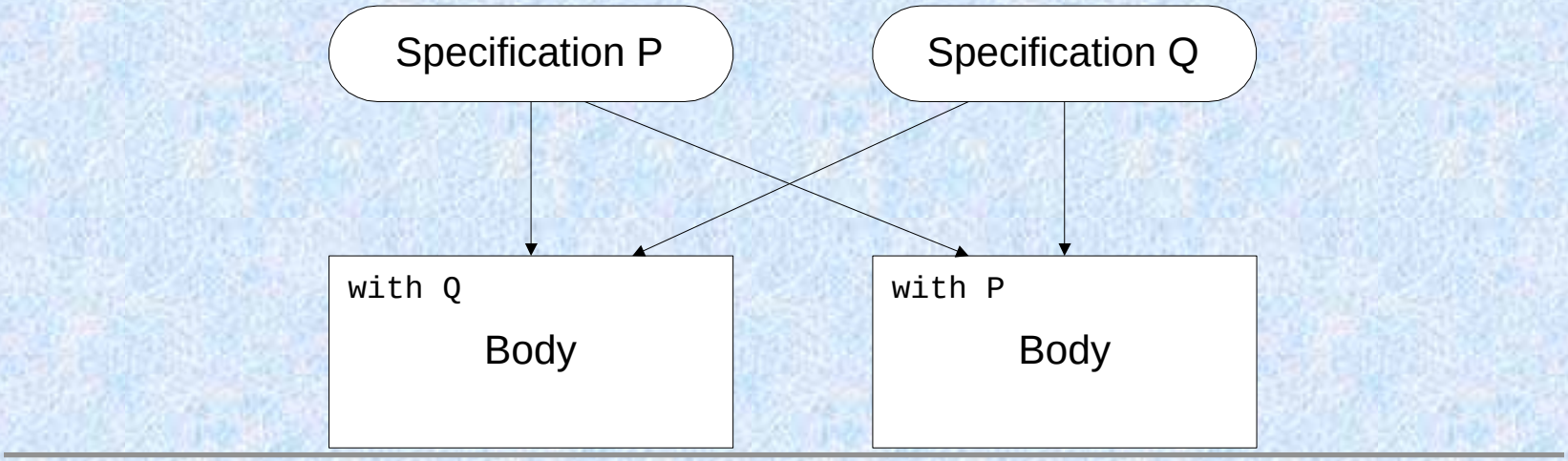
```
--# inherit Stack;
package Application is
...
   procedure Use_Stack;
   --# global  in out Stack.State;
   --# derives Stack.State from Stack.State;
end Application;
```

```
with Stack;
package body Application is
...
   procedure Use_Stack is
   begin
      ... Stack.Push(...); ... Stack.Pop(...); ...
   end Use_Stack;
end Application;
```

inherit and interdependency

- **SPARK requires that inheritance paths be non-circular. Package P may not inherit Q if Q directly or indirectly inherits P.**

  As inherits annotations appear on the package specifications this eliminates circular dependencies between packages. The following structure is permitted in Ada but not in SPARK.



Package annotations

Both the following annotations appear in the package specification:

- **The** own variable clause **announces the constituents of the package state to other packages in the system.**

Example:

```
--# own A, B, State;
```

- **The** initialisation specification **announces which own variables will be initialised at package elaboration, either at declaration or in the executable statements of the package body.**

Example:

```
--# initializes A, State;
```

own variables

- **Every own variable is either**
  - **a** concrete **Ada variable declared immediately within the declarative region of the package, or**
  - **an** abstract **own variable which is refined in the package body.**
- **Every Ada variable declared in the package specification must be an own variable of the package.**
- **Every Ada variable declared immediately within the package body must be**
  - **an own variable, or**
  - **a refinement consituent of an abstract own variable.**
- **An own variable of a package should only be updated directly in the initialisation of that package or by a procedure declared in that package.**

Refinement definition

- **A refinement definition appears in the package body, it expresses every abstract own variable in terms of its constituents.**

Example:

```
--# own OldState is C, D &
--#     NewState is E, F;
```

- **A refinement definition must be provided for every abstract own variable declared in the package specification.**
- **For a package P, a refinement constituent is either**
  - **an own variable (abstract or concrete) from an embedded package declared in P, or**
  - **a concrete Ada variable declared immediately within the declarative region of P.**

own variables and refinement

```
package P
--# own NewState, A, B;
is
  A : ...;
end P;

package body P
--# own NewState is Q.State, C;
is
  B, C : ...;

  package Q
  --# own State;
    ...
  end Q;
end P;
```

Package initialisation

- **The initialisation part of a package may not contain calls to user-defined subprograms and may only update `own` variables of that package.**
- **The `initializes` annotation must specify all the `own` variables which are initialised either at declaration or in the package initialisation part. This annotation immediately follows the `own` variable annotation in the package specification.**
- **This annotation ensures that any data flow errors relating to the package's `own` variables are detected at the main program level.**

own variables modes

- **An own variable, or refinement, declaration may include a mode; this indicates a connection to the external environment**

```
package Sensor
--# own in Inputs;
is …
```

- **Own variables with modes are called external variables**
- **Only mode *in* or *out* is permitted (not *in out*)**
- **External variables may not appear in initializes annotations**

- **The significance of own variable modes is covered in the lecture on Interfacing.**

Refinement and external variables

- **Refinement is not allowed to change a mode once one is given, so:**
  - **an abstract own variable without a mode can be refined to any combination of unmoded, mode *in* and mode *out* constituents; but,**
  - **all constituents of an abstract own variable with a mode must have the same mode**
- **e.g.**

```
package P
--# own    State,
--#     in Sensor;
is
   ...
```

```
package body P
--# own State is     LastVal,
--#              in  AckPort,
--#              out Register &
--#     Sensor is in Sensor1,
--#               in Sensor2;
is
   ...
```

SPARK 95 Child packages

- Ada 95 allows separate packages to be declared as “children” of a parent.
- “Private” children offer a natural way of achieving encapsulation and top-down refinement of program state. This is a neat alternative to the use of embedded packages, which many people find difficult to work with.
- “Public” children allow the facilities of a package to be extended without the need to alter the package itself, thus avoiding the needs for recompilation and (often) substantial retesting.
- SPARK 95 simplifies the rules for child packages and emphasises the distinction of purpose between public and private children; the former are for package extension and the latter for package abstraction.

Public child example

```
package Parent
is
  type T is private;

  procedure Op(X : in out T);
  --# derives X from X;

private
  type T is range 0..1000;
end Parent;
```

```
--# inherit Parent;
package Parent.Child
is
  procedure Op2(X : in out Parent.T);
  --# derives X from X;
end Parent.Child;
```

```
package body Parent.Child
is
  procedure Op2(X : in out Parent.T)
  is
  begin
    Parent.Op(X);
    X := X + 100; -- "+" visible here
  end Op2;
end Parent.Child;
```

Private child example (1)

```
package Controls
--# own in State;
is
  type Buttons is (Pressed, NotPressed);

  procedure ReadReset(Setting : out Buttons);
  --# global in State;
  --# derives Setting from State;

  procedure ReadMode(Setting : out Buttons);
  --# global in State;
  --# derives Setting from State;
end Controls;
```

Private child example (2)

```
--# inherit Controls;
private package Controls.Reset
--# own in State;
is
  procedure Read(Setting : out Controls.Buttons);
  --# global in State;
  --# derives Setting from State;
end Controls.Reset;

--# inherit Controls;
private package Controls.Mode
--# own in State;
is
  procedure Read(Setting : out Controls.Buttons);
  --# global in State;
  --# derives Setting from State;
end Controls.Mode;
```

Private child example (3)

```
with Controls.Reset, Controls.Mode;
package body Controls
--# own State is in Controls.Reset.State,
--#             in Controls.Mode.State;
is
  procedure ReadReset(Setting : out Buttons)
  --# global in Reset.State;
  --# derives Setting from Reset.State;
  is
  begin
    Reset.Read(Setting);
  end ReadReset;

  procedure ReadMode(Setting : out Buttons)
  --# global in Mode.State;
  --# derives Setting from Mode.State;
  is
  begin
    Mode.Read(Setting);
  end ReadMode;
end Controls;
```


Overview of SPARK child packages

- **SPARK 95 differs from Ada 95 thus:**
  - **A private child is visible to its parent, but a public child is not.**
  - **Visibility of siblings is restricted to being between those of the same type (public or private).**
  - **Both public and private children can see their parent's specification. However, a private child cannot call parent subprograms or refer to abstract own variables of its parent.**
  - **The SPARK package-related concepts of inheritance, own variables and refinement have been extended to cover child packages, introducing additional SPARK-specific language rules analagous to those in SPARK 83.**

(Note SPARK 95 does not include child* subprograms*)

Public/Private child summary

- A private child is visible to its parent, but a public child is not.
- Both public and private children can see their parent's specification. However, a private child cannot call parent subprograms or refer to abstract own variables of its parent.
- A public child is permitted to inherit packages not inherited by its parent.
- The own variables of a public child, unlike those of private children, are completely independent from those of its parent and so there are no corresponding rules relating to refinement and initialization.

Renaming

- **Renaming is only permitted for subprograms and this is more restrictive than in Ada. The only permitted effect of renaming is to remove the package prefix.**
  - **Operator renaming is permitted at the start of a package specification.**
  - **When package P is embedded, renaming of subprograms declared in P are permitted directly after the declaration of P.**
  - **Renaming is permitted at the start of the declarative region of a package body.**
  - **The renaming may not change the basic name of the entity or the names of its parameters.**

Operator renaming

- An operator may be renamed in either a package specification or a package body.
- Operator renaming is necessary in the absence of the use clause to make visible operators on types defined in other packages.

Example:

```
package Types is
   type Status is (Valid, Invalid);
end Types;

with Types;
--# inherit Types;
package Controller is
   function "="(Left, Right: Types.Status)
                    return Boolean renames Types."=";
   ...
```

Use type clause (SPARK 95)

- **SPARK 95 includes the *use type* clause which is a more convenient way of making operators visible.**

- **SPARK 83 - renaming**

```
with MyTypes;
--# inherit MyTypes;
package P is
  function "="(Left, Right: MyTypes.T)
    return Boolean renames MyTypes."=";

  function IsEqual(X, Y: MyTypes.T)
    return Boolean;
end P;
```

- **SPARK 95 - use type**

```
with MyTypes;
use type MyTypes.T;
--# inherit MyTypes;
package P is
  function IsEqual(
      X, Y: MyTypes.T) return
        Boolean;
end P;
```

Subprogram renaming

- **A subprogram may only be renamed in a package body.**
- **Subprogram renaming is permitted as a lexical convenience for use when making frequent calls to subprograms whose fully qualified name is very long.**
- **When a subprogram is renamed, the fully qualified name is removed from scope.**

SPARK and Ada

- **A package may not be declared within an enclosing package specification.**
- **SPARK requires inherit clauses.**
- **Own variables must be named in a package.**
- **A package initialisation can only be used to initialise own variables of that package.**
- **Private types cannot have discriminants. (In fact no type can have a discriminant).**

Tutorial Outline

- The rationale of SPARK
- The core SPARK language
- Data and information flow analysis
- Lunch
- Design building blocks
- SPARK program design
- Formal verification
- Exception freedom
- Effective SPARK use

Introduction

- **The methods of analysis used here are described in detail in the paper "Information-flow and data-flow analysis of while programs" by Bergeretti and Carré, ACM Trans. on Prog. Lang. and Systems, Vol.7 (1985), 37-61, which is provided as Appendix A of the Examiner Manual.**
- **These notes are intended mainly to give a simple interpretation, and graphical illustrations, of the ideas in the paper.**

Flow of Information Through a Subprogram

- Values of variables may be used to construct values of expressions.
- Values of expressions may be used in obtaining values of variables
  - by assignment,
  - by influencing which assignments are performed.
- (Initial) values of variables may be used in obtaining (final) values of variables,
  - by combination of the above effects,
  - by variable values being preserved.
- These three kinds of information flow can be represented by three "information flow relations" for a subprogram.

The Information-Flow Relations

- **Note the term "program statement" means either a simple or compound statement, or even a sequence of statements (i.e. the term has its Pascal meaning). It can therefore even represent the entire sequence of statements of a main (sub)program in SPARK.**

The Information-Flow Relations

- V is the set of variables of a (sub)program.
- E is the set of instances of expressions within it.

- **For a program statement S,**
  - D_s is the set of variables which S may update,
  - P_s is the set of variables which S may preserve (not update).

- **We also have three binary relations,**
  - λ_s from V to E,
  - μ_s from E to V,
  - ρ_s from V to V.

The Information-Flow Relations

- A binary relation R from P to Q is a set of ordered pairs (i,j) where i belongs to P and j belongs to Q. One can represent it by a Boolean matrix, where certain coefficients (i,j) are true and the others are false.
- For instance, λ_s can be represented by a Boolean matrix with one row corresponding to each member of V and one column corresponding to each member of E.

The Information-Flow Relations

- " v λ_s e " can be read as "the value of the variable v on entry to S may be used in the evaluation of the expression e in S".
- " e μ_s v " can be read as "a value of the expression e may be used in obtaining the value of the variable v on exit from S".
- " v ρ_s v' " can be read as "the value of the variable v on entry to S may be used in obtaining the value of the variable v' on exit from S".

The Null Statement

- Defines nothing, preserves everything, so…

$$D_s = \{\}$$

$$P_s = \mathrm{V}$$

$$\lambda_s = \{\}$$

$$\mu_s = \{\}$$

$$\rho_s = \mathrm{I}$$

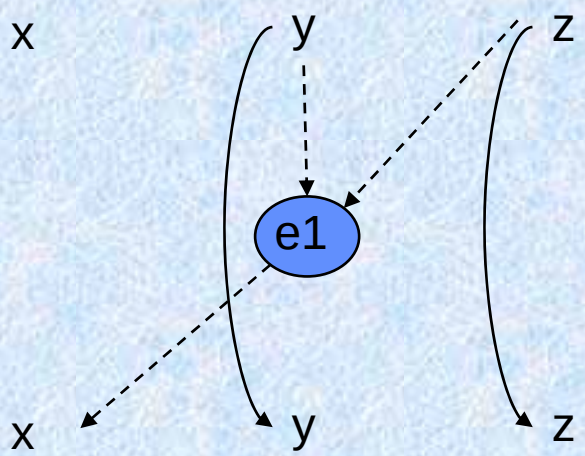
Assignment Statements

```
begin
   x := y + z;
```

e1

```
end;
```

Assignment Statements




```
begin

x := y + z;

end
```

$D_s = \{ x \}$

$P_s = \{ y, z \}$

$\lambda_s = \{ (y, e1), (z, e1) \}$

$\mu_s = \{ (e1, x) \}$

$\rho_s = \{ (y, x), (z, x), (y, y), (z, z) \}$

Sequences of Statements

```
begin

      x := y + z;
```

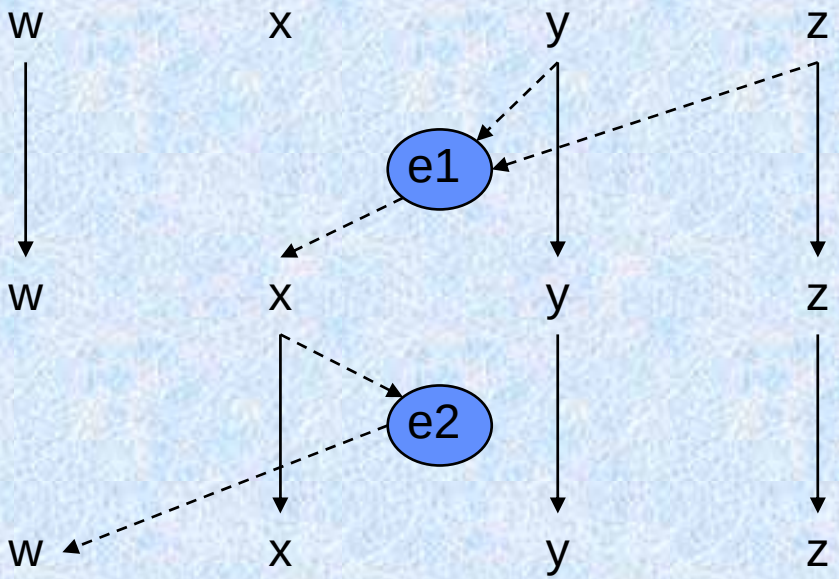
e1

```
      w := 2 * x;
```

e2

```
end;
```

Sequences of Statements




```
begin
   x := y + z;

   w := 2 * x;
end
```

D_s = { *w, x* }

P_s = { *y, z* }

λ_s = { *(y, e1), (z, e1), (y, e2), (z, e2)* }

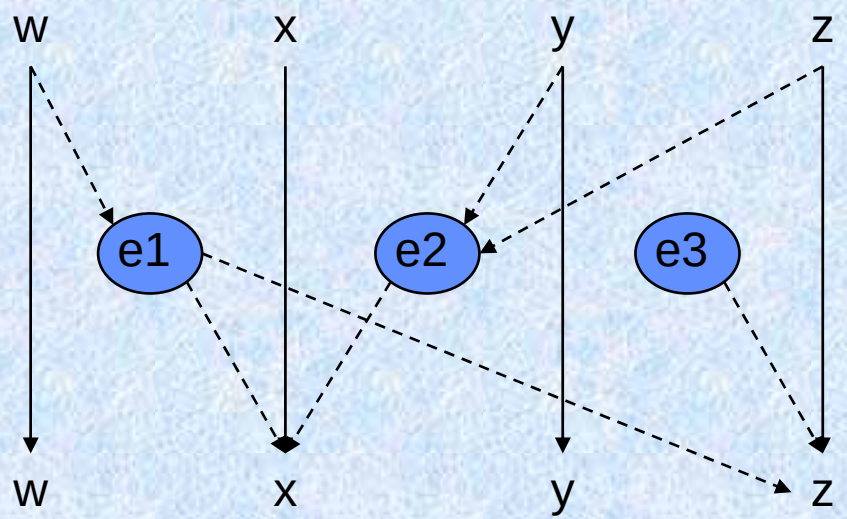
μ_s = { *(e1, w), (e1, x), (e2, w)* }

ρ_s = { *(y, w), (z, w), (y, x), (z, x), (y, y), (z, z)* }

Conditional Statements

```
if  w > 0  then
      e1
   x := y + z
          e2
else
   z := 2;
        e3
end if;
```

Conditional Statements




```
if w > 0 then -- e1
   x := y + z; -- e2
else
   Z := 2;     -- e3
end if;
```

$D_s = \{ x, z \}$

$P_s = \{ w, x, y, z \}$

$\lambda_s = \{ (w, e1), (y, e2), (z, e2) \}$

$\mu_s = \{ (e1, x), (e1, z), (e2, x), (e3, z) \}$

$\rho_s = \{ (w, w), (w, x), (x, x), (y, x), (z, x), (y, y), (w, z), (z, z) \}$

Loop Statements

- **Loop statements can be "unrolled", and their flow relations found by repeated application of those for if-statements and sequences of statements - see Bergeretti and Carré, Section 2.2 and Section 5.**

```
if C then
   S;
   while C loop
      S;
   end loop;
end if;
```

... and so on.

Transitive closure can be calculated efficiently using Warshall's algorithm

Loop Statements

```
while f1(x, y) loop
```
e1

```
   x := f2(y);  y := f3(z);
```
e2 *e3*

```
end loop;
```

$D_s = \{ x, y \}$

$P_s = \{ x, y, z \}$

$\lambda_s = \{ (x, e1), (y, e1), (z, e1), (y, e2), (z, e2), (z, e3) \}$

$\mu_s = \{ (e1, x), (e1, y), (e2, x), (e2, y), (e3, x), (e3, y) \}$

$\rho_s = \{ (x, x), (y, x), (z, x), (x, y), (y, y), (z, y), (z, z) \}$

Procedure Call Statements

- **Procedure calls are considered to be a set of simultaneous assignments defined by the derives annotation of the called subprogram**

```
begin

   P(x, (y+r), z);

end
```

```
procedure P(A : in out T;
            B : in     T;
            C :    out T);
--# derives A from A, B &
--#         C from A;
```

ρ_s = { $(x, x), (y, x), (r, x), (x, z), (r,r), (y,y)$ }

The Data Flow Relations

- **The Examiner also computes two data-flow relations,**
  - **θ_s from V to E,**
  - **θ'_s from V to E.**
- **" v θ_s e " means "the value of the variable v on entry to S may be used directly in the evaluation of the expression e in S".**
- **" v θ'_s e " means "the value of the variable v on entry to S is always used directly in the evaluation of the expression e in S".**
- **The definitions of these relations, for different statement kinds, are given in Table 2 of the ACM paper.**

The Data Flow Relations

```
begin
   if x > 0 then          -- e1
      y := 1;             -- e2
   end if;
   z := 2 * y;            -- e3
end ;
```

D_s = { y, z }

P_s = { x, y }

λ_s = { $(x, e1), (x, e3), (y, e3)$ }

θ_s = { $(x, e1), (y, e3)$ }

θ'_s = { $(x, e1)$ }

μ_s = { $(e1, y), (e1, z), (e2, y), (e2, z), (e3, z)$ }

ρ_s = { $(x, x), (x, y), (y, y), (x, z), (y, z)$ }

Interpretation of Examiner Flow Errors

- **The specification of a subprogram gives**
  - **the set of its imported variables;**
  - **the set of its exported variables;**
  - **its dependency relation.**
- **The Examiner checks consistency of this information with the computed flow relations in the following ways. (The section numbers that follow refer to sections of the Examiner Manual, which explain error messages.)**

8.3.1 Data Flow Errors (use of undefined variables)

- **If v θ' e and v is not imported, an unconditional data-flow error arises in evaluating e.**

- **If this condition does not arise, but v θ e, and v is not imported, a conditional data-flow error arises in evaluating e.**

8.3.2 Data Flow Anomalies and Ineffective Statements

- **If for an expression e, there does not exist an exported variable v such that e μ v, execution of the statement associated with e is ineffective.**

8.3.3 Invariant Conditionals and Stable Exit Conditions

- **The stability of expressions in a loop is determined from the λs and μs relations for the body of the loop, as indicated in Section 4.4 of the ACM paper.**

8.3.4 Discrepancies between Specified Dependency Relations and Code

- **If the specified dependency relation of a procedure states that a variable v is derived from a variable w, but for the computed flow relation ρ the condition w ρ v does not hold, a message indicating an unconditional error is issued.**
- **If for some exported variable v, a condition w ρ v is found to hold, which does not appear in the specified dependency relation, a message to this effect is issued, prefixed by three question marks (???). (In this case the detected dependency of v on w could perhaps be associated with a non-executable program path.)**

Arrays and Flow Analysis

- **Note that the Examiner treats array objects as single, entire objects. Since Release 3.0 of the tool, updating and reading of fields of records is analysed in terms of those fields, rather than being treated as operations on records in their entirety.**
- **Array objects should be initialised using aggregates, wherever this is reasonable. The Examiner recognises that initialisation of an array using an aggregate initialises all its elements.**
- **Initializing an array in a loop will result an a indication of a data flow error even if the array is, ultimately, fully defined.**

Some Examples

```
X := Y + Z;  W := 2 * X;
```

$V = \{W, X, Y, Z\}, \quad E = \{e1 \equiv Y+Z, e2 \equiv 2 * X\}$

$D = \{W, X\}, \quad P = \{Y, Z\}$

$\lambda = \{(Y,e1), (Z, e1), (Y, e2), (Z, e2)\}$

$\mu = \{(e1, W), (e1, X), (e2, W)\}$

$\rho = \{(Y, W), (Z, W), (Y, X), (Z, X), (Y, Y), (Z, Z)\}$

Some Examples

```
X := Y + Z;  W := 2 * X;
```

$D = \{W, X\}$, $P = \{Y, Z\}$

$$D = \begin{array}{c} \\ w \\ x \\ y \\ z \end{array} \begin{array}{c} \begin{array}{cccc} w & x & y & z \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \end{array} \qquad P = \begin{array}{c} \\ w \\ x \\ y \\ z \end{array} \begin{array}{c} \begin{array}{cccc} w & x & y & z \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{array}$$

Some Examples

```
X := Y + Z;  W := 2 * X;
```

$\lambda = \{(Y,e1), (Z, e1), (Y, e2), (Z, e2)\}$

$\mu = \{(e1, W), (e1, X), (e2, W)\}$

$\rho = \{(Y, W), (Z, W), (Y, X), (Z, X), (Y, Y), (Z, Z)\}$

$$\lambda = \begin{array}{c} \\ w \\ x \\ y \\ z \end{array} \begin{array}{c} \begin{array}{cc} e1 & e2 \end{array} \\ \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \end{array} \qquad \mu = \begin{array}{c} \\ e1 \\ e2 \end{array} \begin{array}{c} \begin{array}{cccc} w & x & y & z \end{array} \\ \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \end{array} \qquad \rho = \begin{array}{c} \\ w \\ x \\ y \\ z \end{array} \begin{array}{c} \begin{array}{cccc} w & x & y & z \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix} \end{array}$$

Some Examples

$$\lambda = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \qquad \mu = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \qquad \rho = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix}$$

$$P = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\rho = \lambda \, \mu \text{ or } P$$

Correct Implementation of 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;
```

$$\lambda\,1 = \begin{bmatrix}1\\0\\0\end{bmatrix} \quad \mu\,1 = \begin{pmatrix}0&0&1\end{pmatrix} \quad \rho\,1 = \begin{bmatrix}0&0&1\\0&0&0\\0&0&0\end{bmatrix}$$

$$\lambda = \begin{bmatrix}1&0&1\\0&1&0\\0&0&0\end{bmatrix} \quad \mu = \begin{bmatrix}0&1&1\\1&0&0\\0&1&0\end{bmatrix} \quad \rho = \begin{bmatrix}0&1&1\\1&0&0\\0&0&0\end{bmatrix}$$

Final Statement Incorrectly Sets Y to X

```
T := X;   X := Y;   Y:= X;
```

$$\lambda = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad \mu = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix} \quad \rho = \begin{array}{c} \\ x \\ y \\ t \end{array} \begin{array}{c} \begin{matrix} x & y & t \end{matrix} \\ \begin{pmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$

ρ shows that the initial value of Y can effect the final value of Y, contrary to the derives annotation.

??? Imported value of Y may be used in derivation of Y

Data Flow Errors

First statement now assigns to Y instead of T:

Y := X; X := Y; Y := T;

$$\theta' = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \mu = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \quad \rho = \begin{array}{c} \\ x \\ y \\ t \end{array} \begin{array}{c} \begin{matrix} x & y & t \end{matrix} \\ \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \end{pmatrix} \end{array}$$

ρ shows that the uninitialized variable T may be used to define the export Y

??? The undefined initial value of T may be used in the derivation of Y

Optional information flow analysis (1)

- **To increase flexibility in the use of SPARK, information flow analysis has been made optional for SPARK 95.**
- **If information flow analysis is not to be performed, the only information required for analysis is whether each parameter and global variable is imported, exported or both.**
- **It is important to note that verification activities (including the generation of run-time checks but excluding path functions) retain their validity as long as a program is free from data-flow errors. (i.e. as long as all variables are given values before they are used.)**

Optional information flow analysis (2)

- **Provided global modes have been provided, data flow analysis is selected with the command line option:**

  ```
  -flow_analysis=data
  ```

- **In this case the derives annotation is not used and may be omitted.**
- **All flow error messages that result are valid; however, some cases of ineffective statements and stability may not be detected.**
- **Recommendation is to perform information flow analysis at lower levels in the call hierarchy and switch to data flow analysis at higher levels (e.g. scheduler).**
- **A data flow option, using moded globals, is not possible in SPARK 83**

Tutorial Outline

- The rationale of SPARK
- The core SPARK language
- Data and information flow analysis
- Lunch
- Design building blocks
- SPARK program design
- Formal verification
- Exception freedom
- Effective SPARK use

Tutorial Outline

- The rationale of SPARK
- The core SPARK language
- Data and information flow analysis
- Lunch
- Design building blocks
- SPARK program design
- Formal verification
- Exception freedom
- Effective SPARK use

Design Building Blocks

Abstract data types and Abstract State machines

Language Support for Abstraction

- **Support through subprograms allow extension of virtual machine by adding higher level operations**
- **Abstraction to new data objects often more useful**
  - ***Abstract Data Types* and *Abstract State Machines***

```
package Stacks is
   type Stack is private;

   procedure Clear(S : out Stack);
   procedure Push(S: in out Stack; X: in Integer);
   procedure Pop(S: in out Stack; X: out Integer);
private
   --
end Stacks;
```

Abstract State Machine

- ADTs give the ability to declare objects, and operate on them
- ASMs declare one object for us, and the operations on it
- Many computing processes may be viewed as ASMs
  - eg database manager, with operations to insert, modify, remove records

```
package The_Stack is
   procedure Clear;
   procedure Push(X : in  Integer);
   procedure Pop (X : out Integer);
end The_Stack;
```

Dependency Relations

- When designing a procedure, the designer chooses its exported variables and determines which imported variables are needed to derive the exports

- In SPARK this relationship is captured by the derives annotation
  - SPARK Examiner checks that this relationship is satisfied by the code

```
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;
```

Example: SPARK ADT

```
package Stacks is
   type Stack is limited private;
   function EmptyStack(S : in Stack) return Boolean;
   function FullStack (S : in Stack) return Boolean;

   procedure ClearStack(S :    out Stack);


   procedure Push(S : in out Stack; X : in Integer);


   procedure Pop(S : in out Stack;  X : out Integer);


private
  ...
end Stacks;
```

Example: SPARK ADT

```
package Stacks is
   type Stack is limited private;
   function EmptyStack(S : in Stack) return Boolean;
   function FullStack (S : in Stack) return Boolean;

   procedure ClearStack(S :    out Stack);
   --# derives S from ;

   procedure Push(S : in out Stack; X : in Integer);
   --# derives S from S, X;

   procedure Pop(S : in out Stack;  X : out Integer);
   --# derives S, X from S;

private
--# hide Stacks
end Stacks;
```

Refinement of ADT Package Spec

```
package Stacks is
   type Stack is limited private;
   function EmptyStack (S : Stack) return Boolean;
   function FullStack  (S : Stack) return Boolean;
   procedure ClearStack(S : out Stack);
   --# derives S from ;
   procedure Push(S : in out Stack; X : in Integer);
   --# derives S from S, X;
   procedure Pop(S : in out Stack; X : out Integer);
   --# derives S, X from S;
private
   StackSize : constant := 100;
   type PointerRange is range 0 .. StackSize;
   subtype IndexRange is PointerRange range 1 .. StackSize;
   type    Vector is array(IndexRange) of Integer;
   type Stack is record
      StackVector  : Vector;
      StackPointer : PointerRange;
   end record;
end Stacks;
```

ADT Package Body (1)

```
package body Stacks is
   function EmptyStack(S : Stack) return Boolean is
   begin
      return S.StackPointer = 0;
   end EmptyStack;

   function FullStack(S : Stack) return Boolean is
   begin
      return S.StackPointer = StackSize;
   end FullStack;

   procedure ClearStack(S : out Stack)
   is
   begin
      S := Stack'(Vector'(others => 0), 0);
   end ClearStack;
```

ADT Package Body (2)

```
   procedure Push(S : in out Stack; X : in Integer)
    is
    begin
       S.StackPointer := S.StackPointer + 1;
       S.StackVector(S.StackPointer) := X;
    end Push;

    procedure Pop(S : in out Stack; X : out Integer)
    is
    begin
       X := S.StackVector(S.StackPointer);
       S.StackPointer := S.StackPointer - 1;
    end Pop;

end Stacks;
```

Global variables

```
procedure Outer is
   I, L : Integer;
   procedure Inner (K : in Integer) is
      J : Integer;
   begin
      J := K + I;
      I := J + 2;
   end Inner;
begin
   I := 4;
   L := 2:
   Inner(L);
end Outer;
```

Global variables

```
procedure Outer is
   I, L : Integer;
   procedure Inner (K : in Integer) is
      J : Integer;
   begin
      J := K + I;
      I := J + 2;
   end Inner;
begin
   I := 4;
   L := 2:
   Inner(L);
end Outer;
```

Global Annotation

- **Imports or exports global to the procedure are announced in the global annotation**

```
procedure Inner (K : in Integer)
--# global in out I;
--# derives I from I, K;
is
   J : Integer;
begin
   J := K + I;
   I := J + 2;
end Inner;
```

Own and Initializes Annotations

- To check dependency relationships involving globals, the Examiner needs to know about the existence of package variables
- The “own” annotation announces package variables declared in the package spec or body
  - these are known as own variables
- The “initializes” annotation announces which of these are to be initialized by package elaboration
- A special form of the own variable annotation is used to show communication between a SPARK program and its external environment

ASM Package Spec

```
package Stack


is
   procedure Push(X : in Integer);


   procedure Pop(X : out Integer);


end Stack;
```

ASM Package Spec

```
package Stack
--# own S, Top;
--# initializes Top;
is
   procedure Push(X : in Integer);




   procedure Pop(X : out Integer);




end Stack;
```

ASM Package Spec

```
package Stack
--# own S, Top;
--# initializes Top;
is
   procedure Push(X : in Integer);
   --# global  in out S, Top;


   procedure Pop(X : out Integer);
   --# global  in out Top;


end Stack;
```

ASM Package Spec

```
package Stack
--# own S, Top;
--# initializes Top;
is
   procedure Push(X : in Integer);
   --# global  in out S, Top;
   --# derives S   from S, Top, X &
   --#         Top from Top;

   procedure Pop(X : out Integer);
   --# global  in out Top;
   --#         in     S;
   --# derives Top from Top &
   --#         X   from S, Top;
end Stack;
```

ASM Package Spec

```
package Stack
--# own S, Top;
--# initializes Top;
is
   procedure Push(X : in Integer);
   --# global  in out S, Top;
   --# derives S   from S, Top, X &
   --#         Top from Top;

   procedure Pop(X : out Integer);
   --# global  in out Top;
   --#         in     S;
   --# derives Top from Top &
   --#         X   from S, Top;
end Stack;
```

This is too much detail!

ASM Package Body (1)

```
package body Stack is

   StackSize : constant := 100;
   type TopRange is Integer range 0 .. StackSize;
   subtype IndexRange is TopRange range 1 .. StackSize;
   type Vector is array(IndexRange) of Integer;
   S   : Vector;
   Top : TopRange;

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

ASM Package Body (2)

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

begin -- initialization
    Top := 0;
end Stack;
```

ASM Package Spec using Refinement

```
package Stack
--# own State;          -- abstract own 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 Stack;
```

ASM Package Body using Refinement (1)

```
package body Stack
--# own State is S, Top;   -- refinement definition
is
   StackSize : constant := 100;
   type TopRange is range 0 .. StackSize;
   subtype IndexRange is TopRange range 1 .. StackSize;
   type Vector is array(IndexRange) of Integer;
   S : Vector;   Top : TopRange;
   procedure Push(X : in Integer)
   --# global  in out S, Top;
   --# derives S from S, Top, X & Top from Top;
   is
   begin
      Top := Top + 1;
      S(Top) := X;
   end Push;
```

ASM Package Body using Refinement (2)

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

begin -- initialization
   Top := 0;
   S := Vector'(others => 0);
end Stack;
```

Refinement Observations

- **Both S and Top must now be initialized**
  - since the initializes annotation announcement applies to all refinement constituents
- **The subprogram bodies must now have refined derives and global annotations**
  - since the subprogram specifications refer to the abstract state
- **The refinement constituents may themselves be abstract own variables**
  - for instance if embedded or private child packages are used

ADTs and ASMs

- **ADTs constructed as packages with private types**
  - **package spec has a visible part defining interface, i.e. entities to be exported**
  - **package spec also has private part defining the types (for separate compilation)**
  - **package body defines implementation detail**
- **ASMs constructed as packages with private state**
  - **package spec just has visible part, again defining interface**
  - **package body defines state and other implementation detail**
  - **package state must be announced in own variable annotation**

Visibility of ASM State

- **The own annotation makes the existence of state visible to users**
  - **this is because annotations of subprograms using the ASM must reflect the dependencies on ASM state**
- **But only the existence of the state is needed. Not the detail.**
- **Refinement allows "detail hiding"**
  - **package state can be represented as an "abstract own variable" in the spec, then refined into "concrete" own variables in the body.**
  - **note that the abstract own variable is simply a place holder**

Annotations for Formal Verification

```
procedure CheckTemperature (Temperature : in Real;
                           Override    : in Boolean);
--# global  in out SystemState;
--# derives SystemState from SystemState,
--#                           Temperature,
--#                           Override;
--# pre  True;
--# post ((Temperature >
--#           SystemConsts.MaxSafeTemp) and not Override
--#          -> SystemState = OverHeatState)
--#      and
--#      ((Temperature <=
--#           SystemConsts.MaxSafeTemp) or Override
--#          -> SystemState = SystemState~);
```

Interfacing

Pragmas and Representation clauses

- **In Ada pragmas provide compiler directives, and representation clauses specify how types and objects should be mapped onto the underlying machine.**
- **In general SPARK checks the location of pragmas but not their legality.**
  - **SPARK recognises pragma *`interface/import`* since it must know not to expect a body.**
  - **In SPARK 95, pragma *`elaborate_body`* is recognised**
  - **The location of pragmas in SPARK is slightly more restricted than in Ada.**
- **SPARK ignores representation clauses (apart from recognising the presence of an address clause in certain cases).**
- **The Examiner raises a warning when a representation clause or unrecognised pragma is encountered.**

Hidden text

- **Hidden text is a piece of the program which will be ignored by the Examiner. It may be of any form, including full Ada.**
- **The parts of the program which may be hidden are:**
  - **subprogram body;**
  - **package private part;**
  - **package body;**
  - **package initialisation.**
- **Hidden text may be used either to ignore incomplete parts of a program during examination or to hide parts of a program where one needs to use full Ada.**

```
procedure Secret
--# global  This, That;
--# derives This from That;
is
   --# hide Secret
    ...  -- hidden text
end Secret;
```

Interfacing to the External Environment

- **Useful programs interact with their environment**
  - **Values supplied by the environment are volatile**
  - **Values sent to the environment may not have any visible effect** inside the SPARK program itself
- **SPARK annotations can be used to capture these properties**
  - **By using particular idiomatic forms of annotation**
  - **By using external variables (own variables declared with modes)**
- **Early identification of the boundary between the SPARK program and its environment, together with the method of communicating across it, is an essential design step**

Idiomatic form of External Sequences(1)

A procedure (in package *Time*) reads an external clock.

```
procedure Read_Clock(T : out Time);
--# derives T from ;
```

- **The above annotations imply that the time never changes and may result in flow errors indicating that successive calls to *Read_Clock* are ineffective.**
- **A solution is to assume there is some time state which changes outside the program and model this state with an abstract variable of the *Time* package.**

```
procedure Read_Clock(T : out Time);
--# global  in out Time_Sequence;
--# derives T, Time_Sequence from Time_Sequence;
```


Idiomatic form of External Sequences(2)

A procedure (in package *Time*) waits for a specified time interval.

```
procedure Wait(T : in Time);
--# derives  ?? ;
```

- **T must appear in the derives annotations but there is no state which it affects in the system, all that changes is the time. The elapsed time will depend on the parameter T.**
- **Again assume there is some time state which changes outside the program and model this state with an abstract variable of the *Time* package.**

```
procedure Wait(T : in Time);
--# global  in out Time_Sequence;
--# derives Time_Sequence from T, Time_Sequence;
```


External Variables

- **Since Release 6.0 of the Examiner facilities have been added to simplify the description of interactions with the environment**
- **Own variables representing values obtained from or sent to the environment can be declared with a mode; these are called** external variables**:**

```
package Actuator
--# own out Settings;
is ...
```

- **Global and derives annotations involving external variables are written in their obvious form, without any side effects:**

```
--# derives Actuator.Setting from Value; -- not *, Value;
```

- **The Examiner reconstructs the volatile flow analysis effects automatically**

Data streams example using External Variables

```
package InPorts
--# own in InSeq;
--  no initialization needed or permitted
is

  procedure ReadPort(InputVal : out Integer);
  --# global  in InSeq;
  --# derives InputVal from InSeq;

  function EndOfSeq return Boolean;
  --# global InSeq;
...

package OutPorts
--# own out OutSeq;
is

  procedure WritePort(OutputVal : in Integer);
  --# global  out OutSeq;
  --# derives OutSeq from OutputVal;
```

Use of the null derives form

- **The use of external variables means that sometimes we need to import such a variable but derive nothing from it. For example, the busy wait procedure introduced earlier**

```
procedure Wait(T : in Time);
```

- **If the clock values are obtained from external variable *Time_Sequence* then it and *T* are clearly imports to procedure *Wait*.**
- **Nothing within the SPARK program is derived from the imports (the effect is the consumption of clock ticks in the environment)**
- **In such cases we may use the special derives form:**

```
--# derives null from Time_Sequence, T;
```


Additional Rules

- **Ada variables which are marked as external variables may only be used directly in simple assignment statements (not general expressions)**
- **Functions may globally reference external variables but the function can only be used directly in simple assignment statements**

```
loop
   exit when EndOfSeq;
   ...
end loop;
```

must be written

```
loop
   Done := EndOfSeq;
   exit when Done;
   ...
end loop;
```

these restrictions are to prevent evaluation order effects

- **Because the semantics of external variables differ from ordinary variables the Examiner will warn if a variable is declared as external but no address clause is found for it**

Example - a Simple Sensor

Specification

```
package WaterHighSensor
--# own in State;
is
   function IsActive return Boolean;
   --# global State;

end WaterHighSensor;
```

Example - a Simple Sensor

Body (1)

```
with System.Storage_Elements;
package body WaterHighSensor
--# own State is in HighSensorPort;
--  refinement of external abstract name to chosen memory mapped variable
is
   type Byte is range 0..255;
   ActiveValue : constant Byte := 255;

   HighSensorPort : Byte;
   for HighSensorPort'Address use
      System.Storage_Elements.To_Address
         (16#FFFF_FFFF#);
   --  the Examiner will expect an address clause
```

Example - a Simple Sensor

Body (2)

```
function IsActive return Boolean
--# global HighSensorPort;
--  second annotation in refined, code-level terms
is
   RawVal : Byte;
   Result : Boolean;
begin
   RawVal := HighSensorPort;
   if RawVal'Valid then
      Result := RawVal = ActiveValue;
   else
      Result := True; -- "safe" value
   end if;
   return Result;
end IsActive;
end WaterHighSensor;
```

Example - a Complex IO Device (1)

Output device with the following specification:

if value_to_write = last_value_written then do nothing
else
store value_to_write in last_value_written
write value_to_write to out register
busy wait until ack value (16#FFFF_FFFF) received at status port.

```
package Device
--# own State;
--# initializes State;
is
  procedure Write (X : in Integer);
  --# global in out State;
  --# derives State from State, X;
end Device;
```

Example - a Complex IO Device (2)

```
package body Device
--# own State is        OldX,
--#             in      StatusPort,
--#                out Register;
--  refinement on to mix of external and ordinary variables
is
  OldX : Integer := 0; --  only component that needs initialization
  StatusPort : Integer;
  --  address clause
  Register : Integer;
  --  address clause
```

Example - a Complex IO Device (3)

```
procedure WriteReg (X : in Integer)
--# global out Register;
--# derives Register from X;
is
begin
  Register := X;
end WriteReg;

procedure ReadAck (OK : out Boolean)
--# global in StatusPort;
--# derives OK from StatusPort;
is
  RawValue : Integer;
begin
  RawValue := StatusPort; -- only assignment allowed here
  OK := RawValue = 16#FFFF_FFFF#;
end ReadAck;
```

Example - a Complex IO Device (4)

```
   procedure Write (X : in Integer)
   --# global in out OldX;
   --#           out Register;
   --#        in     StatusPort;
   --# derives OldX,Register from OldX, X &
   --#          null           from StatusPort
   is
     OK : Boolean;
   begin
     if X /= OldX then
       OldX := X;
       WriteReg (X);
       loop
         ReadAck (OK);
         exit when OK;
       end loop;
     end if;
   end Write;
end Device;
```

Interfacing to C (1)

- **Interfacing to other languages (most commonly C and Assembler) is a common source of problems.**

- **When faced with having to call a C API, apply the following steps:**
  - **Read the API documentation!**
  - **Does the C API have persistent state? Yes – then create SPARK abstract own variables to model it. Consider initialisation of that state.**
  - **Construct a SPARK subprogram specification for the C API, including flow relations.**
  - **Bind the SPARK procedure to the C API using Ada95 LRM Annex B facilities.**

Interfacing to C (2)

- **Example – A C implementation of the SHA-1 Secure Hash Algorithm.**
- **We are given:**

```
typedef unsigned char BYTE;

int sha_1 (BYTE *data, BYTE *result);
```

- **What does this tell us? Not much!**

Interfacing to C (3)

- **Step 1 – read the documentation**

```
typedef unsigned char BYTE;

/* data is a pointer to 1024 bytes of data to be hashed.

   result is a pointer to 20 bytes where the result is placed.

   function returns an integer in the range 0 to 3.  0 indicates
   no error has occurred. */
int sha_1 (BYTE *data, BYTE *result);
```

- **OK – this tells us enough to make progress.**

- **Step 2 – Is there any persistent state?**
  - **No…so no SPARK abstract own variables are needed.**

Interfacing to C (4)

- **Step 3 – Construct a SPARK subprogram specification, including information flow**
  - **Big decision – is this a procedure or a function in SPARK?**
  - **A C function that returns a value and has a side-effect via a parameter isn't really a function at all – so it had better be a procedure in SPARK.**
  - **Is each parameter and global referenced, updated or both?**
  - **We need some types:**

```
type Byte is mod 256;
type Buffer_Index is range 1 .. 1024;
type Result_Index is range 1 .. 20;
-- Assume here obvious C-style data layout.  Could force with rep.
-- clauses if necessary.
type Buffer_Type is array (Buffer_Index) of Byte;
type Result_Type is array (Result_Index) of Byte;
-- Assumes C "int" is same as Ada "Integer"
subtype Error_Code is Integer range 0 .. 3;
```

Interfacing to C (5)

```
procedure Sha1 (Data   : in     Buffer_Type;
                Result :    out Result_Type;
                Error  :    out Error_Code);
--# derives Result, Error from Data;
```

- This “looks right” from a SPARK point of view, and gives us a clean SPARK-friendly interface to the API.
- Step 4 – Bind SPARK procedure to C API
  - You MUST read LRM B.1 through B.3
  - Understand what “Convention C” means
  - Check that your compiler follows the LRM implementation advice and read your compiler’s annex M.

Interfacing to C (6)

```
procedure Sha1
  (Data    : in     Buffer_Type;
   Result  :    out Result_Type;
   Error   :    out Error_Code)
is
   --# hide Sha1;
   function Internal_Sha1 (Data   : in     Buffer_Type; --  in by reference
                           Result : access Result_Type) -- out by reference
                          return Error_Code;
   pragma Import (C, Internal_Sha1, "sha_1");

   Local_Result : aliased Result_Type;
begin
   Error  := Internal_Sha1 (Data, Local_Result'Access);
   Result := Local_Result;
end Sha1;
```

- Note use of Ada95 access parameters, ‘Access, aliased variable, and Import with Convention “C”.
- Body is, of course, hidden from the Examiner.

Reference material – LRM Annex B.

- **LRM Annex B.3 section 63 – 71 give implementation advice on the C Convention:**

63 **An implementation should support the following interface correspondences between Ada and C.**

64 **·An Ada procedure corresponds to a void-returning C function.**

65 **·An Ada function corresponds to a non-void C function.**

66 **·An Ada** in **scalar parameter is passed as a scalar argument to a C function.**

67 **·An Ada** in **parameter of an access-to-object type with designated type T is passed as a t* argument to a C function, where t is the C type corresponding to the Ada type T.**

68 **·An Ada** access **T parameter, or an Ada** out **or** in out **parameter of an elementary type T, is passed as a t* argument to a C function, where t is the C type corresponding to the Ada type T. In the case of an elementary** out **or** in out **parameter, a pointer to a temporary copy is used to preserve by-copy semantics.**

69 **·An Ada parameter of a record type T, of any mode, is passed as a t* argument to a C function, where t is the C struct corresponding to the Ada type T.**

70 **·An Ada parameter of an array type with component type T, of any mode, is passed as a t* argument to a C function, where t is the C type corresponding to the Ada type T.**

71 **·An Ada parameter of an access-to-subprogram type is passed as a pointer to a C function whose prototype corresponds to the designated subprogram's specification.**

Tutorial Outline

- The rationale of SPARK
- The core SPARK language
- Data and information flow analysis
- Lunch
- Design building blocks
- SPARK program design *INFORMED*
- Formal verification
- Exception freedom
- Effective SPARK use

Introduction

- **The design of a SPARK program can have a major impact on the size, clarity, and stability of the SPARK mandatory annotations, and hence the cost of their maintenance.**
- **Keys factors of good SPARK programs are**
  - **correct classification of state**
  - **the abstraction and localisation of state.**
- **Good SPARK design leads to derives annotations which add value to the software.**

Importance of Architecture

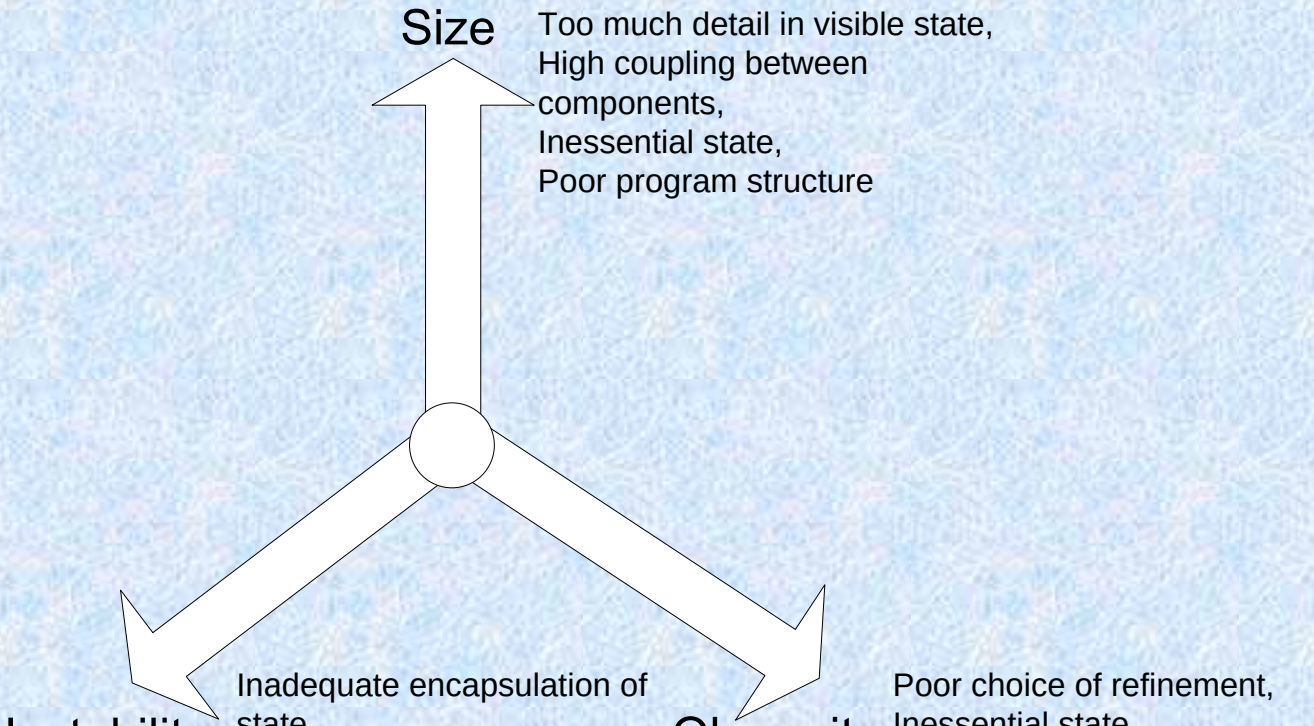
The choice of architecture is vitally important in determining the following properties of a program:

- **clarity;**
- **simplicity of analysis;**
- **simplicity of correctness proof;**
- **ease of testing/validation;**
- **stability.**

Size/Clarity/Stability

- **With the exception of** derives**, the annotations grow roughly linearly with respect to program size.**
- **The** derives **annotations, though, grow more rapidly. The size depends on:**
  - **The amount of static data or "state" in a program.**
  - **The degree of coupling between these state items.**
  - **The "height" of a subprogram in the call-tree.**
  - **The structure or architecture of the program's compilation units.**
- **The** derives **annotation affects the size and maintainability of SPARK programs.**

Size/Clarity/Stability(2)



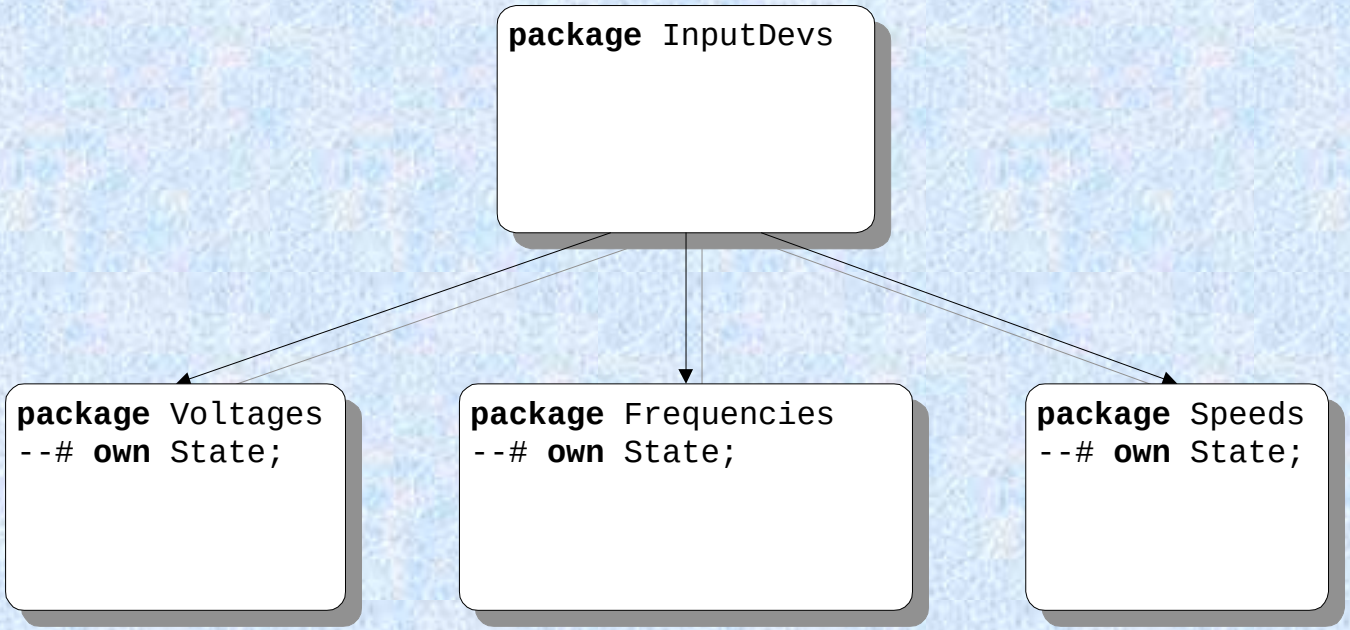
Importance of Architecture

- **Of particular importance are**
  - **the agglomeration of state;**
  - **location of state components.**
- **The SPARK annotations - and in particular global and derives annotations - describe how information flows between these state components.**
- **Good architectural choices are reflected in compact, easily comprehensible derives annotations.**
- **Poor choices, leading to many complex data transfers, lead to large and unwieldy derives lists.**
- **Derives annotations can provide a measure of the quality of the architecture.**

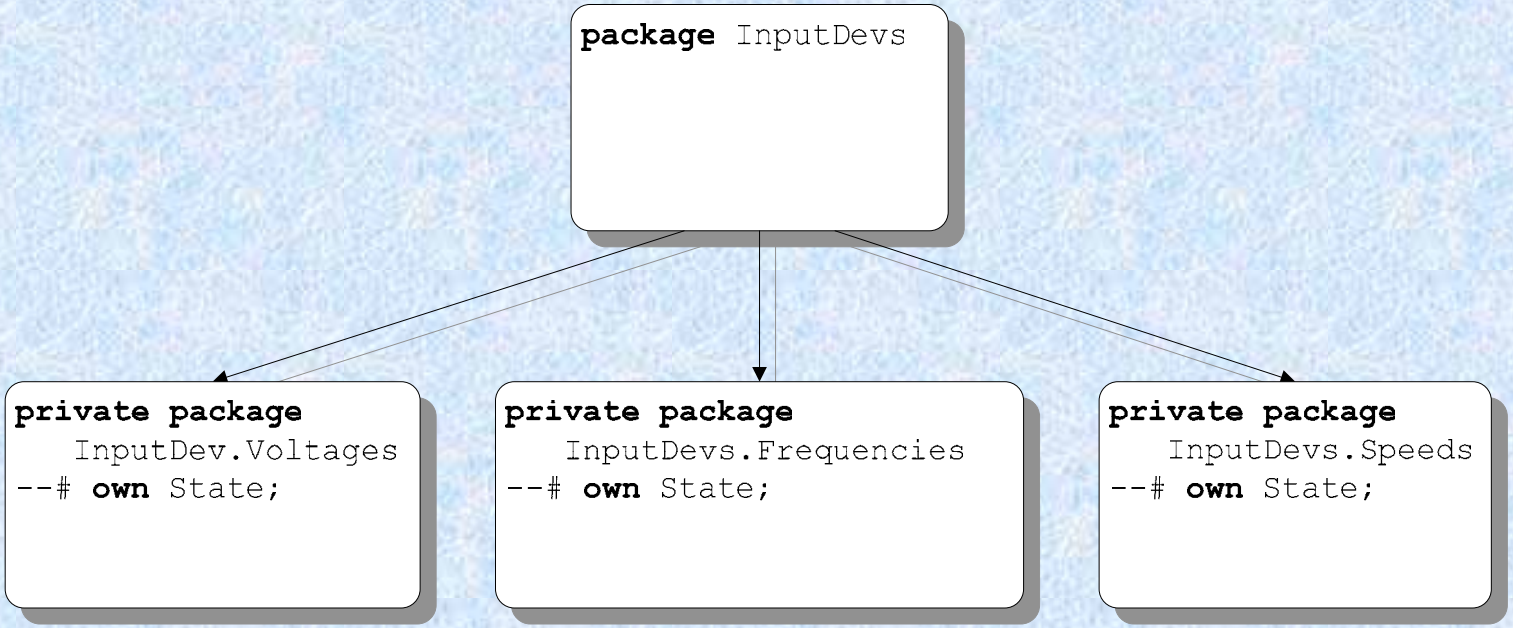
Designing for clear derives annotations

- **SPARK enforces a hierarchical package structure within a program. We shall consider how the following aspects of program design affect derives annotations:**
  - **unnecessary state;**
  - **localisation of state and the use of abstract state machines;**
  - **location of state and package hierarchy;**
  - **appropriate use of SPARK data refinement.**
- **We shall also see that good SPARK design leads to derives annotations which add value to the software.**

An input interface package



A better solution



Object Oriented Design

- **Encapsulation**
- **Abstraction**
- **Loose coupling**
- **Cohesion**
- **Hierarchy**

Object Oriented Design

- **Encapsulation**
- **Abstraction**
- **Loose coupling**
- **Cohesion**
- **Hierarchy**

SPARK can directly assist with achieving these design goals: e.g. Annotation size is a sensitive measure of coupling between objects.

INFORMED

Information
flow
oriented
method
for
(object)
design.

Principles

- **Application-oriented annotations**
- **Careful selection of the SPARK boundary**
- **Minimised information flow**
- **Separation of the essential from the inessential**
- **Early use of static analysis**

Design steps (simplified)

- **Identification of system boundary, SPARK boundary and communication across them.**
- **Identification and location of system state.**
- **Handling initialization of state.**
- **Handling secondary considerations.**
- **Implementing object bodies.**

Design steps (simplified)

- **Identification of system boundary, SPARK boundary and communication across them.**
- **Identification and location of system state.**
- **Handling initialization of state.**
- **Handling secondary considerations.**
- **Implementing object bodies.**

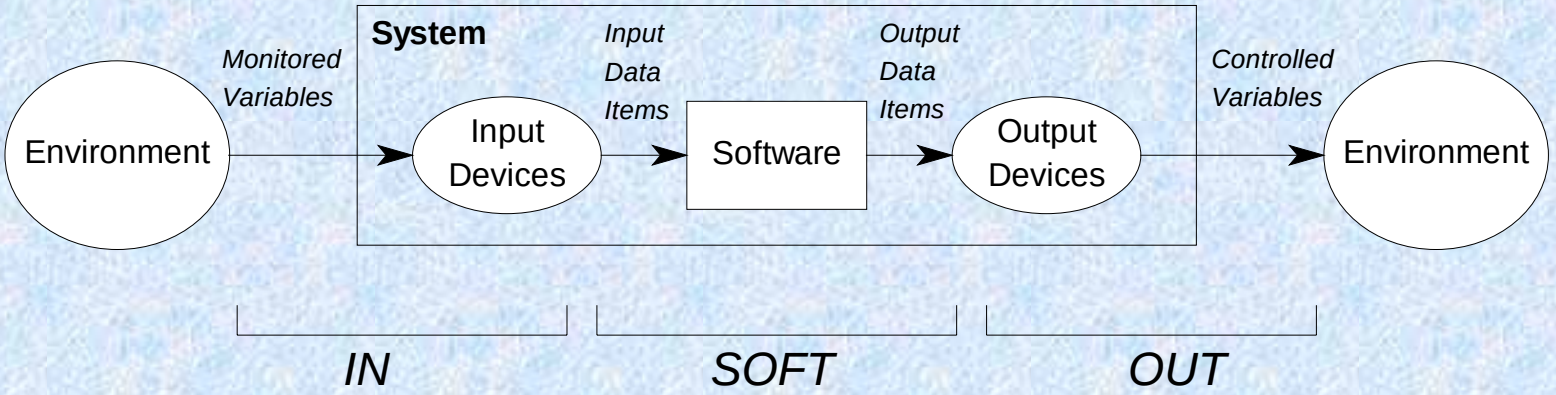
System and SPARK Boundaries

- **Identification of the System Boundary**
  - identify the boundary of the *system* for which INFORMED is being used to provide the software.
  - identify the *physical* inputs and outputs of the system.

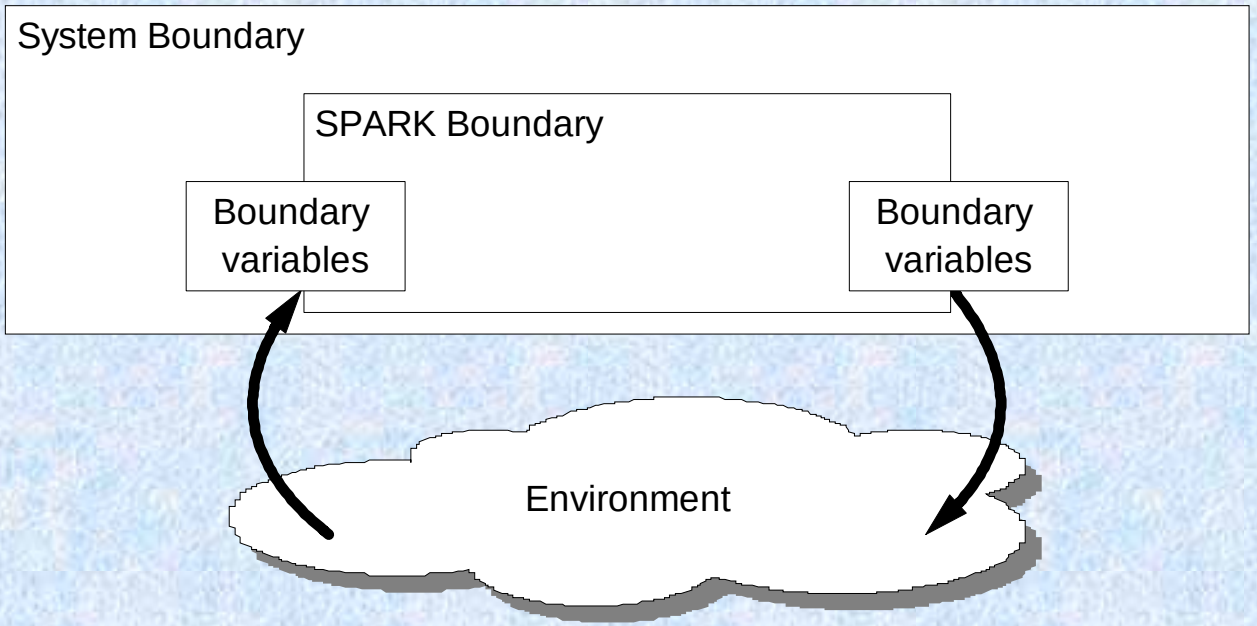
System and SPARK Boundaries

- **Identification of the SPARK boundary.**
  - **select a SPARK boundary within the overall system boundary**
  - **define boundary variables to give controlled interfaces across the SPARK boundary annotated in problem domain terms.**
  - **consider adding boundary abstraction layers.**

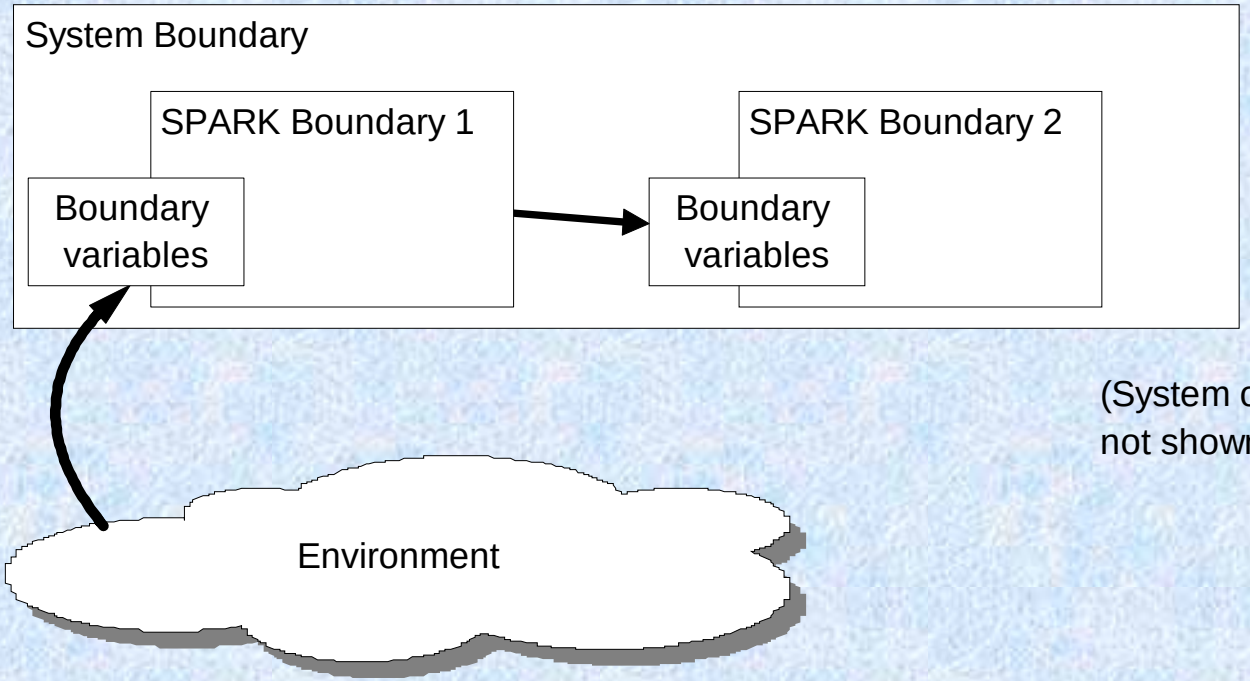
Parnas & Madey Model



System and SPARK Boundaries



Co-operating SPARK Subsystems



(System outputs not shown)

Design steps (simplified)

- Identification of system boundary, SPARK boundary and communication across them.
- **Identification and location of system state.**
- Handling initialization of state.
- Handling secondary considerations.
- Implementing object bodies.

Identification and Localization of State

- **What must be stored?**
- **Where should it be stored?**
  - **consider effect of choice on main program annotations**
- **How should it be stored?**
  - **variable package**
  - **instance of type package**
  - **concrete Ada variable**

A brief diversion: cohesion and type packages

```
package BasicTypes is
   type SwitchType is (On, Off, Unknown);
   type ValveType  is (Open, Closed, Unknown);
end BasicTypes;

Y : BasicTypes.SwitchType;
X : BasicTypes.ValveType;

...

if X = BasicTypes.Open and Y = BasicTypes.Off ...
```

which is much nicer as:

```
package Switch is
   type T is (On, Off, Unknown);
end Switch;

package Valve is
   type T is (Open, Closed, Unknown);
end Valve;

Y : Switch.T;
X : Valve.T;

...

if X = Valve.Open and Y = Switch.Off ...
```

Design steps (simplified)

- Identification of system boundary, SPARK boundary and communication across them.
- Identification and location of system state.
- **Handling initialization of state.**
- Handling secondary considerations.
- Implementing object bodies.

State Initialization

- **Initialized prior to program execution**
  - **implicitly by environment**
  - **explicitly in package elaboration or declarations**
- **Initialized during program execution**
  - **by executable statement**

Design steps (simplified)

- Identification of system boundary, SPARK boundary and communication across them.
- Identification and location of system state.
- Handling initialization of state.
- **Handling secondary considerations.**
- Implementing object bodies.

Secondary Considerations

- **Examples:**
  - test points
  - configuration data
  - caches
  - data logging

Secondary Considerations

- **Principles:**
  - don’t ignore, partition
  - don’t allow primary design to be distorted
  - accommodate by:
    - moving the SPARK boundary
    - encapsulation
    - abstraction

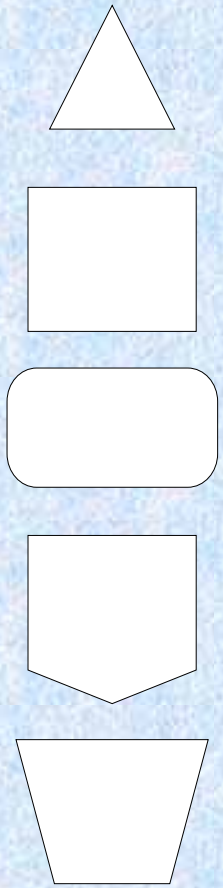
Design steps (simplified)

- Identification of system boundary, SPARK boundary and communication across them.
- Identification and location of system state.
- Handling initialization of state.
- Handling secondary considerations.
- **Implementing object bodies.**

Implementing Objects

- **May identify sub-systems which can be tackled in INFORMED way**
- **Otherwise essentially top-down refinement; but:**
  - **defer implementation using *hide* directive**
  - **use Examiner regularly**
  - **use annotations as a guide to partitioning.**

INFORMED Components



Main program

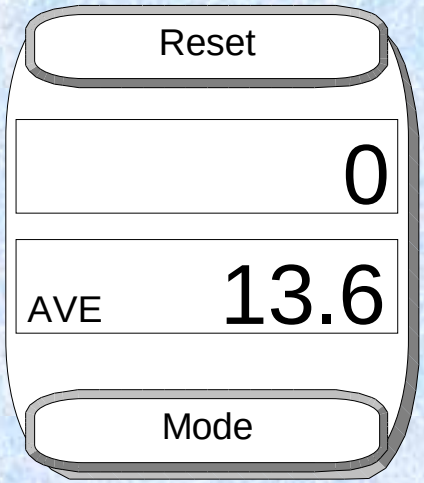
Variable package *(ASM)*

Type package *(ADT)*

Boundary variable

Utility layer

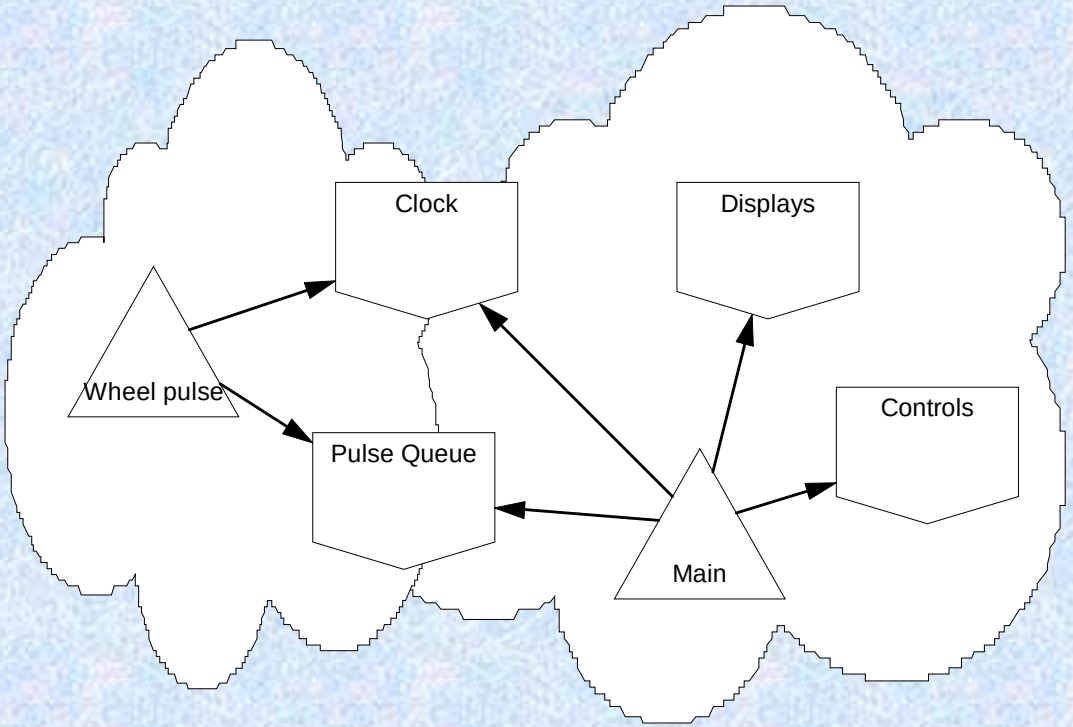
A Cycle Computer



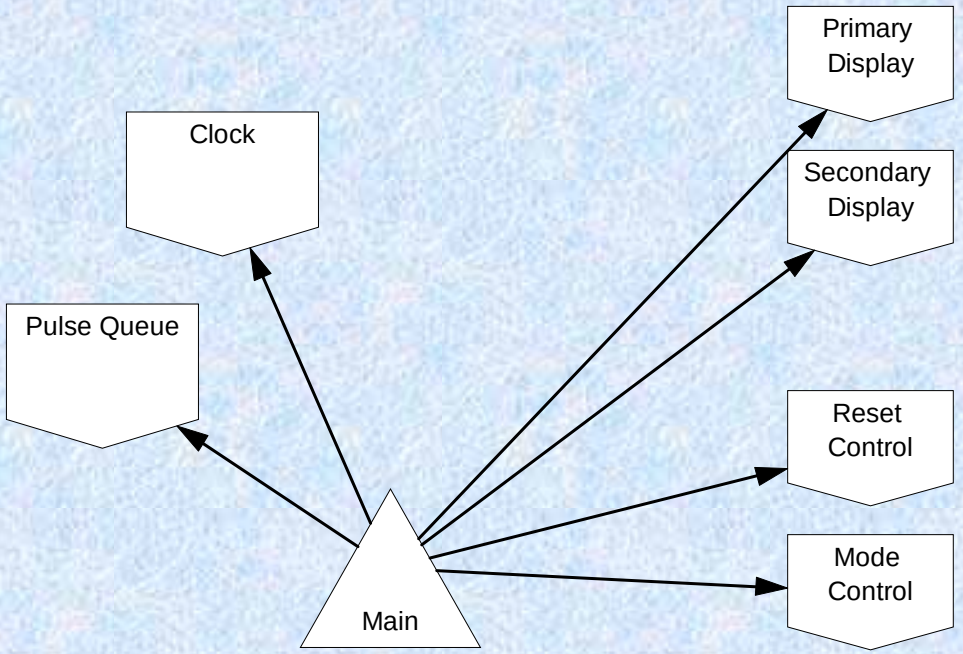
Boundary Considerations

Identification of system boundary, selection of SPARK boundary and definition of boundary variables.

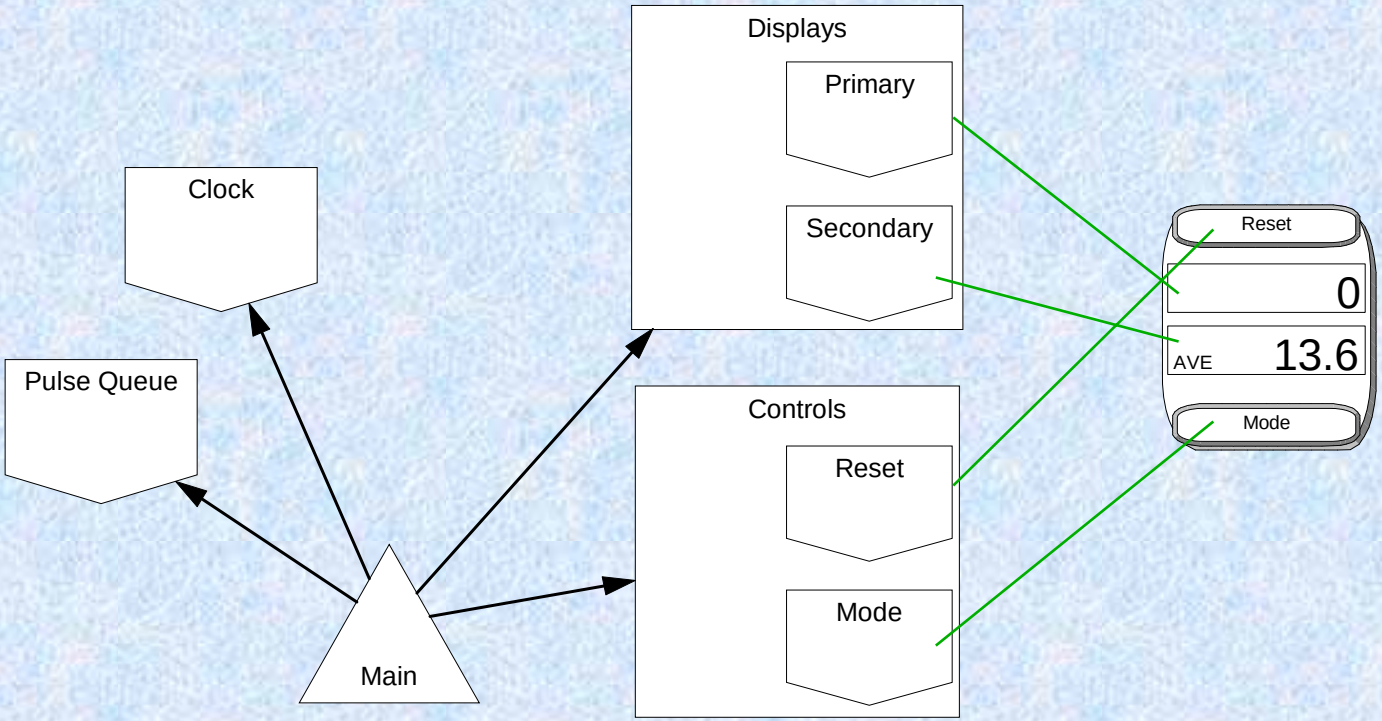
Implementation as Two SPARK Sub-systems



Boundary Variables and Abstractions



Boundary Variables and Abstractions

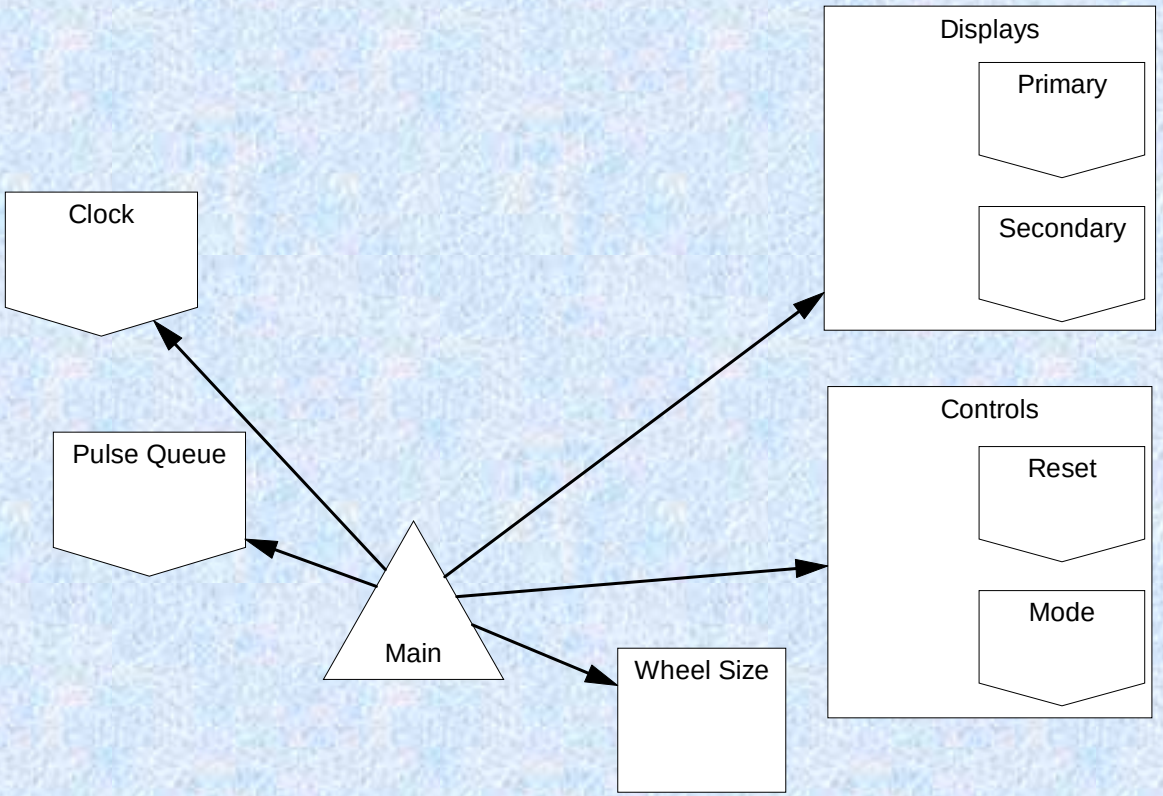


Location of State

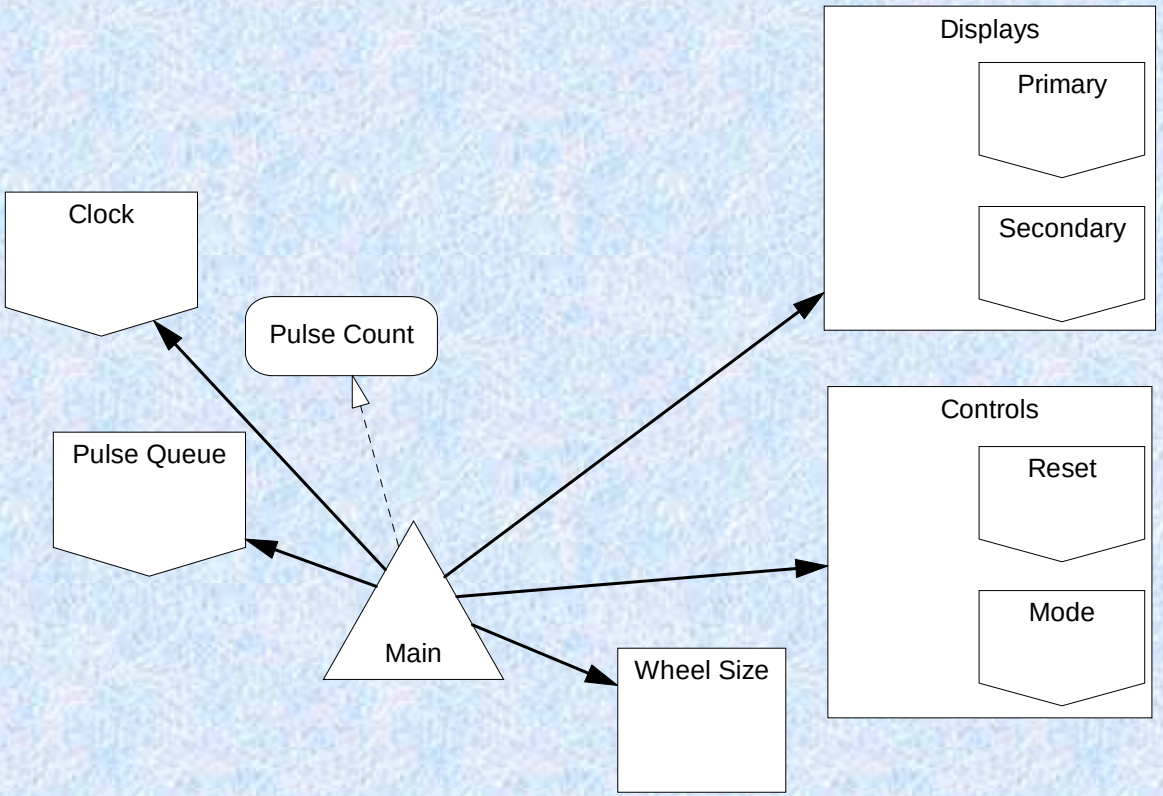
Where and how to store:

- wheel size
- total numbers of pulses received
- averages of pulse intervals
- clock values for stopwatch function

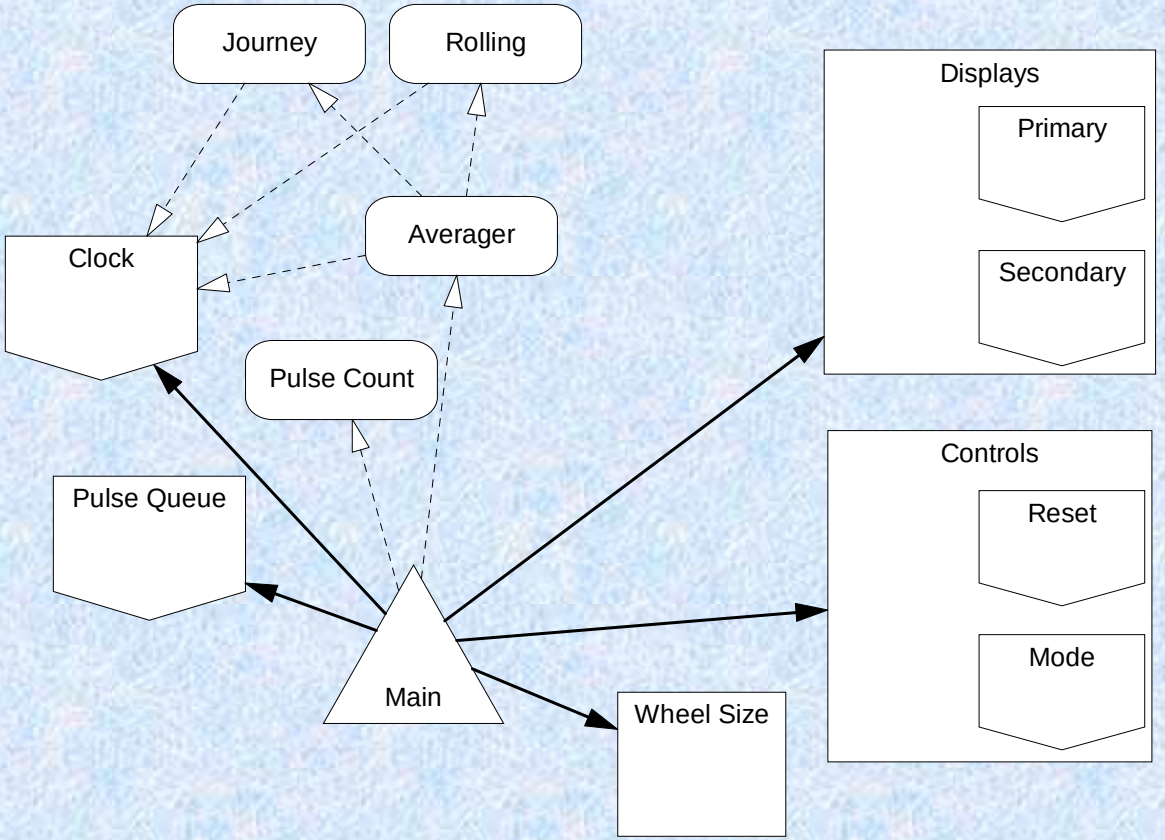
Location of State (1)



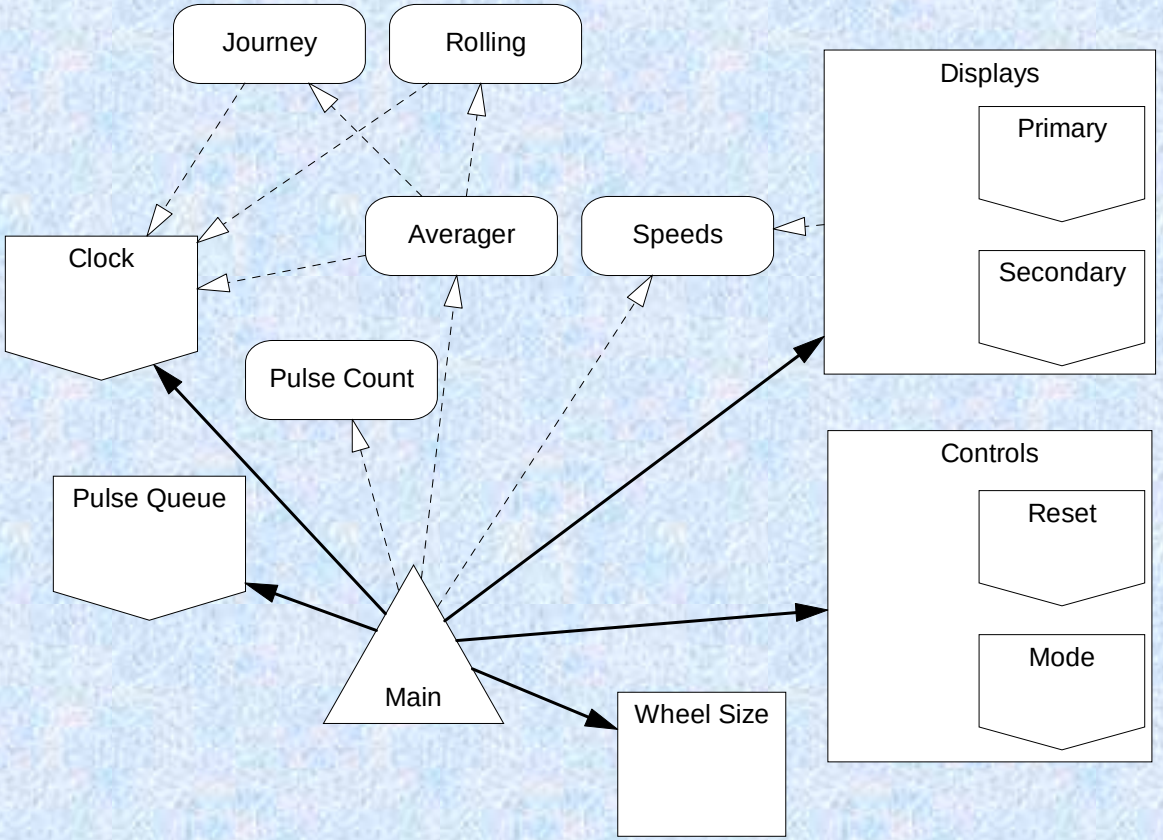
Location of State (2)



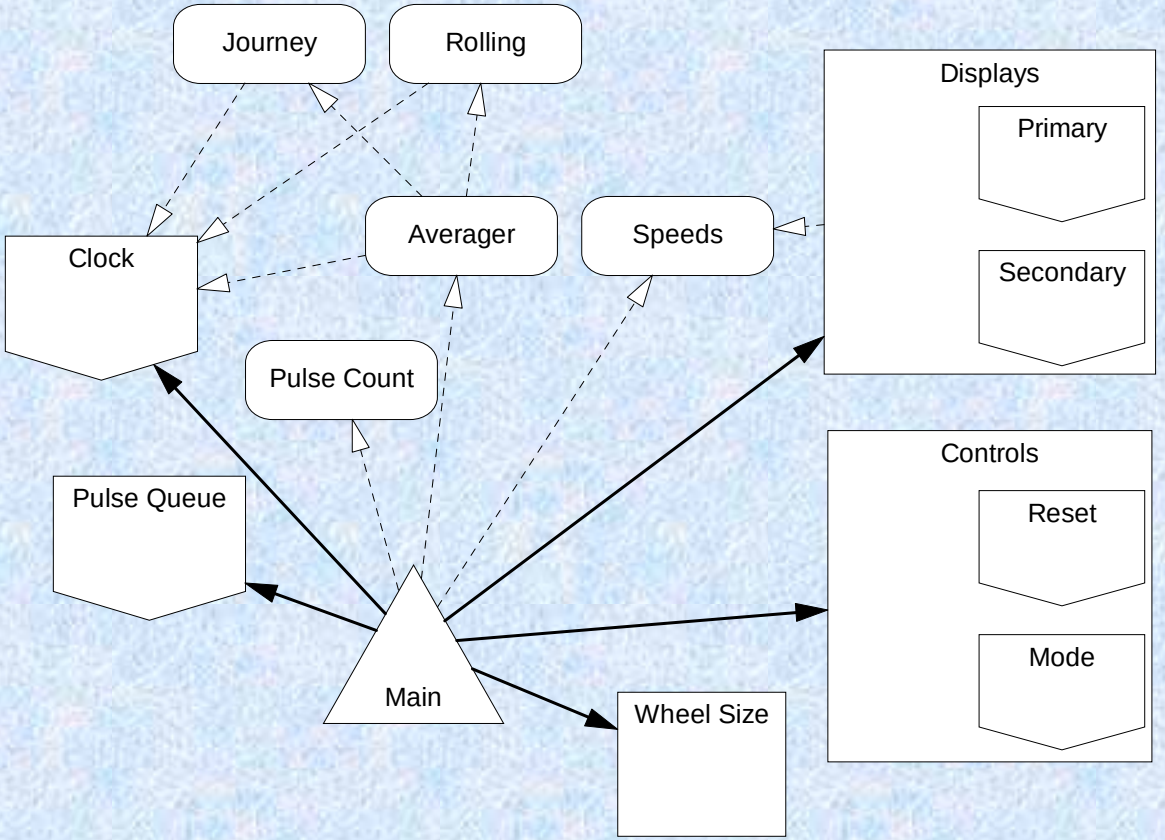
Location of State (3)



Complete Design



Complete Design




```
--# global
--# in Clock.State,
--#    Pulse_Queue.State,
--#    Control.State;
--# out Display.State;
--# in Wheel.Size;
--# derives
--#   Display.State
--# from
--#   Clock.State,
--#   Pulse_Queue.State,
--#   Control.State,
--#   Wheel.Size;
```

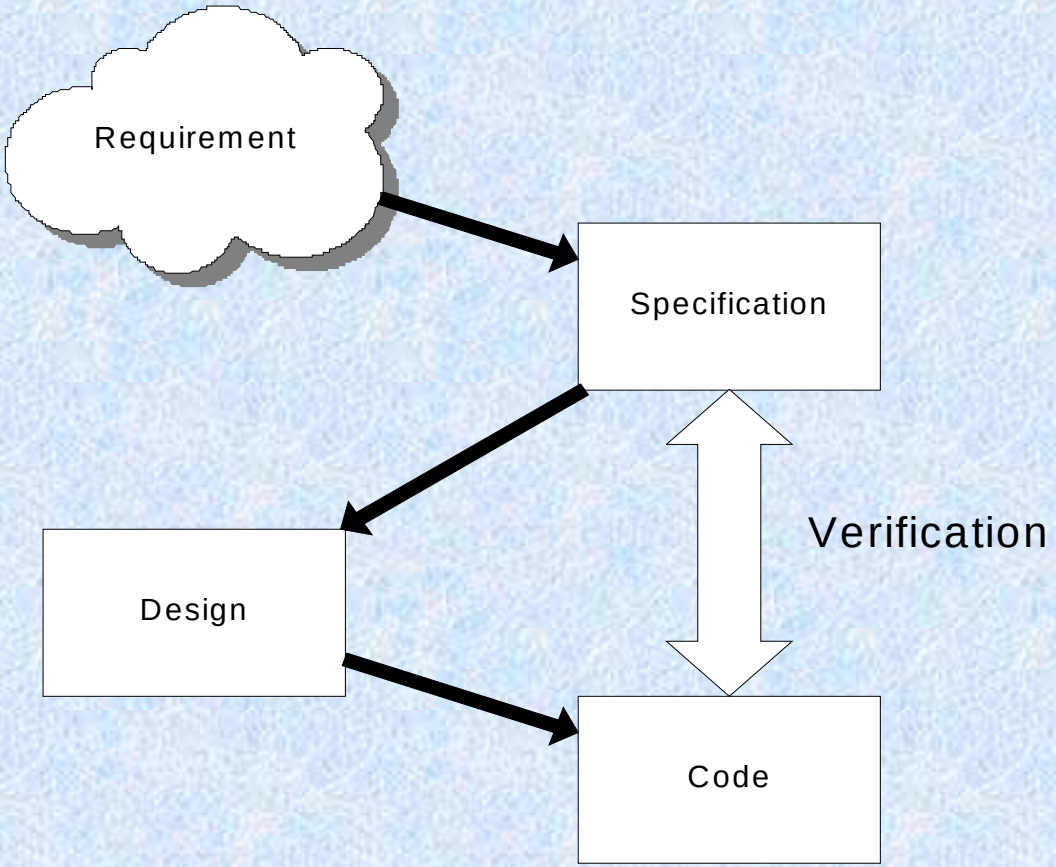
Tutorial Outline

- The rationale of SPARK
- The core SPARK language
- Data and information flow analysis
- Lunch
- Design building blocks
- SPARK program design
- Formal verification
- Exception freedom
- Effective SPARK use

What is a verification?

- **The use of SPARK and flow analysis protects us from certain types of error but does not show that a program correctly implements some specification.**
- **Verification is concerned with the** meaning **of the program and whether the specification has been met.**
- **SPARK facilitates rigorous verification because of the unambiguous meaning of a SPARK text.**

Verification



A simple specification in English

if X greater than or equal to 10 then X becomes 1

if X equals 0 then X remains 0

if X less than 10 then X becomes -1

Note the ambiguity when X = 0; this is fairly typical and would require clarification.

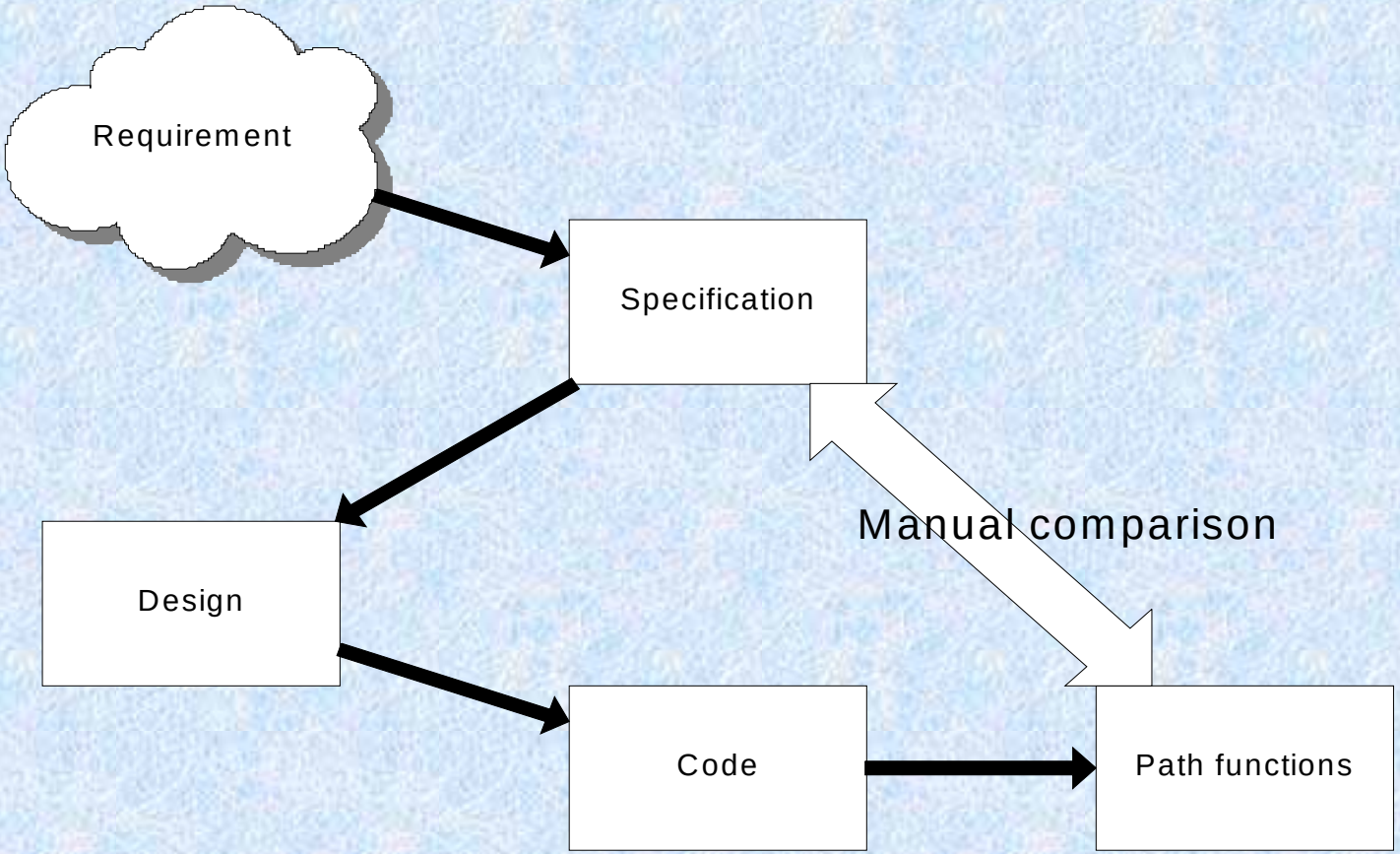
A flawed implementation

```
procedure Process(X : in out integer)
--# derives X from X;
is
begin
  if    X > 10 then X := 1;
  elsif X = 0  then X := 0;
  elsif X < 10 then X := -1;
  end if;
end Process;
```

Generation of Path Functions

- *A “serial-to-parallel” transformation of a SPARK program allowing informal semantic analysis.*
- *Can be applied to code containing only mandatory annotations.*

Path function analysis



```
Statement: start 1 successor(s)
  Successor statement:  finish.
    Path  1
      Traversal condition:
    1:  x > 10 .
      Action:
        x' := 1 .
    Path  2
      Traversal condition:
    1:  x = 0 .
      Action:
        x' := 0 .
```

```
    Path  3
      Traversal condition:
    1:  x <> 0 .
    2:  x < 10 .
      Action:
        x' := - 1 .
    Path  4
      Traversal condition:
    1:  x = 10 .
      Action:
        unit function .
```

An extension to the specification

if X greater than or equal to 100 then X becomes 10

otherwise perform the earlier transformation of X.

An implementation

```
procedure Process2(X : in out integer)
--# derives X from X;
is
begin
  if   X >= 100 then X := 10;
  else Process(X);
  end if;
end Process2;
```

```
Statement: start      1 successor(s)
  Successor statement:  finish.
    Path  1
      Traversal condition:
    1:  x >= 100 .
      Action:
        x' := 10 .
    Path  2
      Traversal condition:
    1:  x < 100 .
      Action:
        x' := process__x(x) .
```

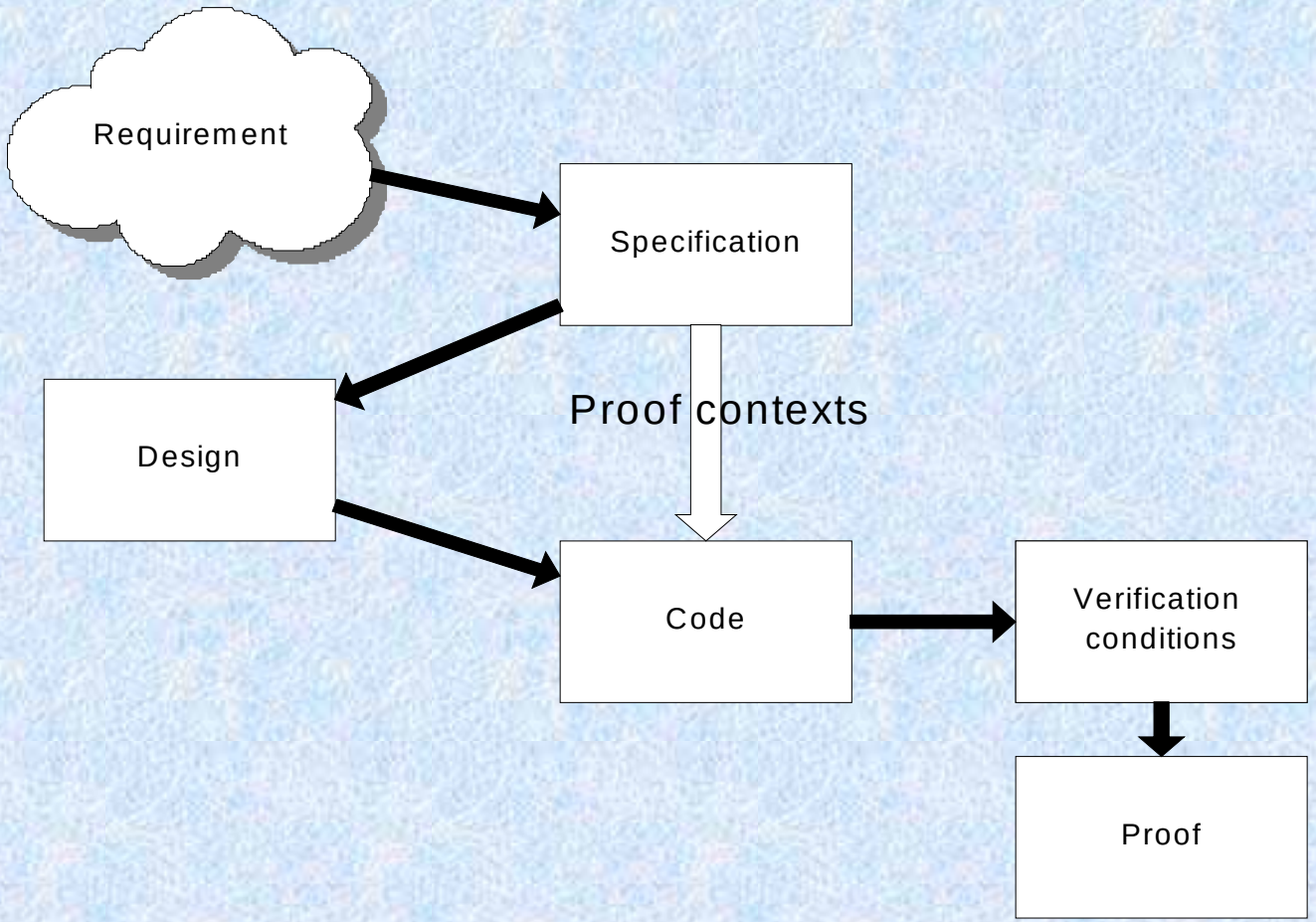
Generation of Path Functions

- *A “serial-to-parallel” transformation of a SPARK program allowing informal semantic analysis.*
- *Can be applied to code containing only mandatory annotations.*
- *Time consuming.*
- *Essentially retrospective.*
- *Only as rigorous as those reading the path functions.*
- *Obsolescent?*

Generation of Verification Conditions

- *The generation from code, suitably annotated with a specification, of the proof obligations required to show its compliance with that specification.*

Program proof



A flawed implementation (2)

```
procedure Process(X : in out integer)
--# derives X from X;
--# post ((X~ >= 10) -> X = 1) and
--#      ((X~ = 0) -> X = 0) and
--#      (((X~ /= 0) and (X~ < 10))
--#                  -> X = -1);
is
begin
  if    X > 10 then X := 1;
  elsif X = 0  then X := 0;
  elsif X < 10 then X := -1;
  end if;
end Process;
```

```
procedure_process_1.
H1:    true .
H2:    x > 10 .
        ->
C1:    (x >= 10) -> (1 = 1) .
C2:    (x = 0) -> (1 = 0) .
C3:    ((x <> 0) and (x < 10)) -> (1 =  - 1) .


procedure_process_2.
H1:    true .
H2:    not (x > 10) .
H3:    x = 0 .
        ->
C1:    (x >= 10) -> (0 = 1) .
C2:    (x = 0) -> (0 = 0) .
C3:    ((x <> 0) and (x < 10)) -> (0 =  - 1) .
```


```
procedure_process_3.
H1:    true .
H2:    not (x > 10) .
H3:    not (x = 0) .
H4:    x < 10 .
        ->
C1:    (x >= 10) -> ( - 1 = 1) .
C2:    (x = 0) -> ( - 1 = 0) .
C3:    ((x <> 0) and (x < 10)) -> ( - 1 =  - 1) .

procedure_process_4.
H1:    true .
H2:    not (x > 10) .
H3:    not (x = 0) .
H4:    not (x < 10) .
        ->
C1:    (x >= 10) -> (x = 1) .
C2:    (x = 0) -> (x = 0) .
C3:    ((x <> 0) and (x < 10)) -> (x =  - 1) .
```

```
procedure_process_1.
*** true .          /* all conclusions proved */


procedure_process_2.
*** true .          /* all conclusions proved */


procedure_process_3.
*** true .          /* all conclusions proved */


procedure_process_4.
H1:    true .
       ->
C1:    false .
```

A correct implementation

```
procedure Process(X : in out integer)
--# derives X from X;
--# post ((X~ >= 10) -> X = 1) and
--#      ((X~ = 0) -> X = 0) and
--#      (((X~ /= 0) and (X~ < 10))
--#                -> X = -1);
is
begin
  if    X >= 10 then X := 1;
  elsif X = 0   then X := 0;
  else  X := -1;
  end if;
end Process;
```

```
For path(s) from start to finish:

procedure_process_1.
*** true .          /* all conclusions proved */


procedure_process_2.
*** true .          /* all conclusions proved */


procedure_process_3.
*** true .          /* all conclusions proved */
```

An implementation (2)

```
procedure Process2(X : in out integer)
--# derives X from X;
--# post ((X~ >= 100) -> X = 10) and
--#      (((X~ >= 10) and (X~ < 100))
--#                  -> X = 1) and
--#      ((X~ = 0) -> X = 0) and
--#      (((X~ /= 0) and (X~ < 10))
--#                  -> X = -1);
is
begin
  if   X >= 100 then X := 10;
  else Process(X);
  end if;
end Process2;
```

```
For path(s) from start to finish:

procedure_process2_1.
*** true .          /* all conclusions proved */


procedure_process2_2.
H1:     x < 100 .
H2:     x >= 10 -> x__1 = 1 .
H3:     x = 0 -> x__1 = 0 .
H4:     x <> 0 and x < 10 -> x__1 = - 1 .
        ->
C1:     x >= 10 and x < 100 -> x__1 = 1 .
```

Use of proof functions

We can factor common parts of the specification and avoid repeating parts of the proof by using Proof Functions

These have the signature of an SPARK function but appear in annotation context only:

```
--# function Max(X, Y : integer) return integer;
```

The meaning of a proof function is provided in proof rules:

```
max(X,Y) may_be_replaced_by X if [X>=Y].
max(X,Y) may_be_replaced_by Y if [X < Y].
```

New post conditions

```
--# function PostProcess(X : integer)
--#                 return integer;

procedure Process(X : in out integer)
--# derives X from X;
--# post X = PostProcess(X~);
...

procedure Process2(X : in out integer)
--# derives X from X;
--# post ((X~ >= 100) -> X = 10) and
--#      ((X~ < 100)  -> X = PostProcess(X~));
...
```

Proof involving abstract own variables

Problem: How can we deal with abstract own variables?

```
package P
--# own State; -- abstract!
--# initializes State;
is
   --# function F1 (X : ???) return ???;

   procedure Op1;
   --# global in out State;
   --# derives State from *;
   --# post State = F1 (State~);
end P;
```

Proof involving abstract own variables

*Answer: Use an **abstract proof type** and **type announcement**.*

```
package P
--# own State : SType; -- type announcement
--# initializes State;
is
   --# type SType is abstract;
   --# function F1 (X : SType) return SType;

   procedure Op1;
   --# global in out State;
   --# derives State from *;
   --# post State = F1 (State~);
end P;
```


Generation of Verification Conditions

- *The generation from code, suitably annotated with a specification, of the proof obligations required to show its compliance with that specification.*
- *Allows “formal verification”.*
- *Def Stan 00-55 compliant.*
- *Can be cheaper than retrospective analysis but give higher levels of assurance.*
- *Not subjective.*

Tutorial Outline

- The rationale of SPARK
- The core SPARK language
- Data and information flow analysis
- Lunch
- Design building blocks
- SPARK program design
- Formal verification
- Exception freedom
- Effective SPARK use

Run-Time Errors

SPARK provides protection against many kinds of program defects, not only those detected by a compiler, but also, for instance:

- **failure to initialize variables;**
- **side-effects of function subprograms;**
- **aliasing of procedure parameters.**

However, there remains an important class of errors — run-time errors **— against which we do not have automatic protection unless we attempt to prove such errors are impossible.**

Sources of Run-Time Errors in Ada

(Only those shown in bold **are possible in the SPARK subset.)**

- **CONSTRAINT_ERROR:** access check, discriminant check, **index check,** length check**, range check, division check, overflow check**..
- PROGRAM_ERROR: erroneous execution, incorrect order dependence, return not executed in function subprogram, elaboration check.
- **STORAGE_ERROR:** exhaustion of dynamic heap storage, "**stack overflow**".
- TASKING_ERROR: exceptions raised during inter-task communication.

A simple example

In the assignment statement

```
A(I + J) := P / Q;
```

the following run-time errors might arise:

- **`I+J` might overflow the base-type of the index range's subtype;**
- **`I+J` might be outside the index range's subtype;**
- **`P/Q` might overflow the base-type of the element type;**
- **`P/Q` might be outside the element subtype.**

A rigorous demonstration that such errors cannot occur involves formal proof.

Basis for run-time error checks

- **The proof obligations to show the absence of run-time errors in SPARK have been derived from its formal definition: we can associate a dynamic well-formation check with each language construct in turn. These can be used to address the remaining sources of potential run-time errors under the Ada CONSTRAINT_ERROR and NUMERIC_ERROR exceptions.**

- **STORAGE_ERROR exceptions are not explicitly addressed. However, given the absence of recursion from SPARK and the elimination of dynamic data structures, such problems become much more tractable.**

Tool support

- **Detection of possible run-time errors by static code analysis alone is not feasible.**
- **Although compiled code may contain checks, so that run-time errors are detected as soon as they occur, there is no time to recover from them (in avionic control systems, for instance), nor are appropriate recovery actions necessarily straightforward to specify.**
- **The consequences of a run-time error can be quite as severe as those of any other kind of program error.**
- **The approach adopted with SPARK is to generate proof obligations which, if proved, establish the absence of the relevant run-time errors from the code. Conditions which cannot be proved highlight problem areas in the code for consideration.**

Generation of proof obligations

- **The verification-condition generator of the SPARK Examiner with Run-Time Checker can produce proof obligations for the absence of run-time errors.**
- **Each proof obligation formula corresponds to a construct (a statement or expression) whose evaluation might give rise to a run-time error; together with an "execution path" to the construct. We attempt to establish that for each such path, the error cannot arise in practice.**
- **The proof obligations are expressed as "verification conditions", with hypotheses (derived from the path traversed, plus other assertions) which may be assumed, and conclusions which must be established.**

Proof obligations: Example (1)

```
procedure R (a, b: in SmallInteger; c, d: out SmallInteger)
--# derives c, d from a, b;
--# pre a in SmallInteger and b in SmallInteger;
is
begin
  if a >= 0 and b >= 0 then
    if a >= b then
      c := a - b;  --# check a - b in SmallInteger;
    else
      c := b - a;  --# check b - a in SmallInteger;
    end if;
  elsif a < 0 and b < 0 then
    c := - (b + 1);       --# check - (b + 1) in SmallInteger;
  else
    c := a + b;    --# check a + b in SmallInteger;
  end if;
  d := (a + b) / 2;       --# check 2 /= 0 and (a + b) / 2 in SmallInteger;
end R;
```

Proof obligations: Example (2)

```
Procedure_R_1.
H1:   -128 <= a .
H2:   a <= 127.
H3:   -128 <= b.
H4:   b <= 127.
H5:   a >= 0.
H6:   b >= 0.
H7:   a >= b.
      ->
C1:   -128 <= a - b.
C2:   a - b <= 127.
```

```
Procedure_R_2.
H1:   -128 <= a.
H2:   a <= 127.
H3:   -128 <= b.
H4:   b <= 127.
H5:   a >= 0.
H6:   b >= 0.
H7:   not (a >= b).
      ->
C1:   -128 <= b - a.
C2:   b - a <= 127.
```

```
Procedure_R_3.
H1:   -128 <= a .
H2:   a <= 127.
H3:   -128 <= b.
H4:   b <= 127.
H5:   not ((a >= 0) and (b >= 0)).
H6:   a < 0.
H7:   b < 0.
      ->
C1:   -128 <= - (b + 1).
C2:   - (b + 1) <= 127.
```

```
Procedure_R_4.
H1:   -128 <= a.
H2:   a <= 127.
H3:   -128 <= b.
H4:   b <= 127.
H5:   not ((a >= 0) and (b >= 0)).
H6:   not ((a < 0) and (b < 0)).
      ->
C1:   -128 <= a + b.
C2:   a + b <= 127.
```

Automatic Proof

- **Showing the absence of run-time errors requires the proof of a large number of small theorems.**
- **Experience shows that the majority of these theorems can be proved by an automatic prover, such as the SPADE Automatic Simplifier.**
- **The remaining proof obligation conditions can be discharged in one of three ways:**
  - **interactive proof (with the SPADE Proof Checker);**
  - **code modification (defensive programming);**
  - **introduction of design assumptions, e.g. as preconditions and/or loop-invariants.**

Demonstration

- **This demonstration illustrates the use of RTC proof on a real program.**
- **We will use the model solution to the Heating-Controller problem.**
- **We will use the Examiner, SparkSimp, and POGS tools.**

Demo (1) - RTC Generation

- **Generating VCs for RTC freedom is easy - simply use the “rtc” or “exp” Examiner options.**
  - **“rtc” generates VCs for all checks, except Ada’s Overflow_Check.**
  - **“exp” generates VCs for all checks, including Overflow_Check.**
  - **Optionally, the “real” option may also be used to generate VCs for real-number (I.e. floating and fixed point) expressions. This is** not enabled **by default.**

- **spark /exp heating_answer95.adb**

Demo (2) - POGS

- **POGS is a tool that reads and summarises the state of a proof effort by reading all the VCG, SIV, SLG, and PLG files in a directory tree.**
- **Options**
  - **/I - Ignore timestamps on SIV files.**
  - **/p - Plain output (less detail, but easier to "diff")**
- **Example run: pogs**
- **Note we have not run the Simplifier yet, so we don't really expect to see many proven VCs!**

POGS Summary

```
Total subprograms fully proved by examiner:                4
Total subprograms fully proved by simplifier:              0
Total subprograms fully proved by checker:                 0
Total subprograms fully proved by review:                  0
Total subprograms with at least one undischarged VC:      12  <<<
                                                          ---
Total subprograms for which VCs have been generated:      16

Total VCs by type:
                                  ---------Proved By--------
                          Total Examiner Simplifier Checker Review Undischarged
Assertion or Postcondition:  38    20        0        0       0        18
Precondition check:           0     0        0        0       0         0
Check statement:              0     0        0        0       0         0
Runtime check:               41     0        0        0       0        41
Refinement VCs:              12    12        0        0       0         0
Inheritance VCs               0     0        0        0       0         0
=============================================================================
Totals:                      91    32        0        0       0        59     <<<
% Totals:                          35%       0%       0%      0%       64%    <<<
```

Demo (3) - Simplification

- **SparkSimp is a “make” tool for the Simplifier. Documented in the Simplifier manual.**
- **SparkSimp traverses the current working directory tree, finds all the VCG files that need simplifying, and applies the simplifier to them.**
- **Main options:**
  - **/a - Simplify all VCG files, regardless of timestamp.**
  - **/n - dry run - find VCG files but don’t actually run the simplifier.**
  - **/l - log Simplifier screen output for XXX.vcg to XXX.log**
- **sparksimp /a /n /l -- dry run with logging for all VCG files**
- **sparksimp /a /l**

Demo (4) - POGS again

- Let's run POGS again to see what's changed:

```
Total subprograms fully proved by examiner:                4
Total subprograms fully proved by simplifier:              7
Total subprograms fully proved by checker:                 0
Total subprograms fully proved by review:                  0
Total subprograms with at least one undischarged VC:       5  <<<
                                                          ---
Total subprograms for which VCs have been generated:      16

Total VCs by type:

                                  ---------Proved By--------
                          Total Examiner Simplifier Checker Review Undischarged
Assertion or Postcondition:  38    20       18        0       0        0
Precondition check:           0     0        0        0       0        0
Check statement:              0     0        0        0       0        0
Runtime check:               41     0       34        0       0        7
Refinement VCs:              12    12        0        0       0        0
Inheritance VCs               0     0        0        0       0        0
=============================================================================
Totals:                      91    32       52        0       0        7     <<<
% Totals:                          35%      57%       0%      0%       7%    <<<
```

Demo (5) - Remaining VCs

- The remaining VCs can be reviewed for additional confidence. For example, POGS says that VC 1 of procedure CheckAdvanceButton is not fully proven. Let's have a look in checkadvancebutton.siv:

```
procedure_checkadvancebutton_1.
H1:    time >= 0 .
H2:    time <= 86399 .
H3:    advancebutton__slow <= advancebutton__currentmode(advancebutton__state) .
H4:    advancebutton__currentmode(advancebutton__state) <= advancebutton__fast .
H5:    advancebutton__currentmode(advancebutton__state) = advancebutton__slow .
       ->
C1:    time + 60 >= integer__base__first .
C2:    time + 60 <= integer__base__last .
```

- That's OK - The Examiner doesn't know Integer'Base'Range. For any sensible computer (e.g. 32-bit machine), this VC is trivially True.
- We can tell the Examiner about the range of Integer for our compiler using a configuration file.

Demo (6) – Adding compiler specifics

- **Use a config file (gnat.cfg) containing the following compiler specific information:**

```
package Standard is
    type Integer is range -2**31 .. 2**31 - 1;
end Standard;
```

- **Rerun the Examiner with config file option and re-simplify**

- **spark /exp /conf=gnat heating_answer95.adb**
- **sparksimp /a /l**

Demo (7) - POGS again

- Let's run POGS again to see what's changed:

```
Total subprograms fully proved by examiner:                 4
Total subprograms fully proved by simplifier:              11
Total subprograms fully proved by checker:                  0
Total subprograms fully proved by review:                   0
Total subprograms with at least one undischarged VC:        1  <<<
                                                           ---
Total subprograms for which VCs have been generated:       16

Total VCs by type:
                                    ---------Proved By--------
                          Total Examiner Simplifier Checker Review Undischarged
Assertion or Postcondition:  38    20       18        0       0        0
Precondition check:           0     0        0        0       0        0
Check statement:              0     0        0        0       0        0
Runtime check:               41     0       40        0       0        1
Refinement VCs:              12    12        0        0       0        0
Inheritance VCs               0     0        0        0       0        0
=============================================================================
Totals:                      91    32       58        0       0        1     <<<
% Totals:                          35%      63%       0%      0%       1%    <<<
```

Demo (8) - Remaining VCs

- **The remaining VCs can be reviewed for additional confidence. For example, POGS says that VC 10 of procedure CheckProgramSwtich is not fully proven.**

```
procedure_checkprogramswitch_10.
H1:    for_all(i___1 : programswitch__positions, programswitch__on1 <= i___1  and
       i___1 <= programswitch__off2 -> 0 <= element(onofftime, [i___1]) and
       element(onofftime, [i___1]) <= 86399) .
H4:    programswitch__auto <= switchposition__1 .
H5:    switchposition__1 <= programswitch__off2 .
H6:    switchposition__1 = programswitch__on1 or (switchposition__1 =
          programswitch__off1 or (switchposition__1 = programswitch__on2 or
          switchposition__1 = programswitch__off2)) .
       ->
C1:    element(onofftime, [switchposition__1]) >= 0 .
C2:    element(onofftime, [switchposition__1]) <= 86399 .
C3:    programswitch__on1 <= switchposition__1 .
```

- **We can tell POGS that we have reviewed this VC and give the reason in a .PRV file.**

Demo(9) - .PRV file

- **The .PRV file contains a list of the VCs that we have proved by review. We can justify our proof in this file.**
- **checkprogramswitch.prv could look like this:**

```
-- .prv for procedure CheckProgramSwitch
-- The following VCs were proved by review.
10 -- Proved by case analysis on switchposition__1
```

Demo(10) – POGS again

- Let's run POGS again to see what's changed:

```
Total subprograms fully proved by examiner:                4
Total subprograms fully proved by simplifier:             11
Total subprograms fully proved by checker:                 0
Total subprograms fully proved by review:                  1
Total subprograms with at least one undischarged VC:       0
                                                          ---
Total subprograms for which VCs have been generated:      16

Total VCs by type:

                                  ---------Proved By--------
                          Total Examiner Simplifier Checker Review Undischarged
Assertion or Postcondition:  38    20       18        0       0        0
Precondition check:           0     0        0        0       0        0
Check statement:              0     0        0        0       0        0
Runtime check:               41     0       40        0       1        0
Refinement VCs:              12    12        0        0       0        0
Inheritance VCs               0     0        0        0       0        0
=============================================================================
Totals:                      91    32       58        0       1        0
% Totals:                          35%      63%       0%      1%       0%
```

Demo - Summary

- **Proof is easier than you think!**
  - **Tools are better - Examiner and Simplifier are much improved. SparkSimp and POGS are very useful.**
  - **CPU cycles are cheap - how many 1GHz+ processors have you got sitting around the office?**
- **Give it a go…**
  - **spark /exp …**
  - **sparksimp**
  - **pogs**
  - **That's it…**

Summary

- **Automated tool support for proof of the absence of run-time errors can eliminate concerns over most of the potential sources of run-time errors within SPARK code, particularly where good programming practice has been used (for types and subtypes, appropriate defensive programming, etc.)**
- **VCs which remain undischarged after this can be investigated further and**
  - **eliminated through further defensive programming;**
  - **eliminated through "beefing up" code annotations to record additional design information; or**
  - **eliminated through (documented) rigorous argument.**
- **This fortification (of code and/or annotations) may go through more than one iteration in practice, before all potential errors have been properly addressed.**

Tutorial Outline

- The rationale of SPARK
- The core SPARK language
- Data and information flow analysis
- Lunch
- Design building blocks
- SPARK program design
- Formal verification
- Exception freedom
- Effective SPARK use

Good Projects

- **SPARK has been allowed to influence the design.**
- **Annotations generated in parallel with code.**
- **Early and frequent use of Examiner.**
- **Leading to:**
  - **correctness by construction**
  - **time savings**
  - **cost savings, especially in testing**

Less-Good Projects

- SPARK considerations ignored during design.
- Retrospective “SPARKification” of code.
- Examination only after testing.
- Working round and fighting against Examiner.
- Leading to:
  - large annotations
  - low error detection rate
  - perception of poor value for money

Design principles

- **Allow analyis to influence design:**
  - **use INFORMED**
  - **be prepared to refactor design if annotations “explode”**
  - **Examine early and often**
- **If the Examiner “gets in the way” step back and ask “what aspect of my design is causing this?”**
- **Don’t be afraid to mix Ada and SPARK if the result is more elegant than a pure SPARK solution.**

Project lifecycle organization

- **Identify hardware interfaces early**
  - provide useful abstractions in package specs
  - low-level hardware interfacing can then be implemented in parallel with use of the device in other code
- **Maintain indexes and meta files as part of CM system**
  - use superindex mechanism to allow analysis of locally-modified code
- **Don’t allow ‘check in’ unless code is examined**
- **Don’t allow coders to compile!**

Quality and validation activities

- Establish acceptance process for acceptable warnings and flow error messages
- Comment all such acceptances in the code
- Perform reviews on SPARK listing files not source files
- Scale back unit test effort:
  - SPARK programs benefit little from “mindless” unit test
  - SPARK programs tend to integrate very easily
  - Better to establish test coverage from requirements
- Start exception-freedom proofs once code starts to stabilize
  - i.e. not right at the beginning but do not defer until project is finished
- Never do “formal acceptance” testing before analysis

Above all:

- **Do not confuse apparent progress with real progress:**
  - **it is not lines of code that matter but lines of *delivered*, *working*, *certified* code.**
  - **SPARK may slow apparent progress but delivers higher real progress**

Resources

- **www.sparkada.com**
- **sparkinfo@praxis-his.com**
- **"High Integrity Software: The SPARK Approach to Safety and Security" by John Barnes, April 2003, Addison-Wesley, ISBN 0-321-13616-0**