CIS 890: Safety Critical Systems

Lecture: *Tool Development and Support*

INFORMED Design Method for SPARK

SPARK INFORMED

- Design method for SPARK focused on embedded, critical control functions
- "INFORMED" = INformation Flow Oriented MEthod of (object) Design
  - using information flow as a central tool in the design of the objects or entities making up the system
- Aims to produce systems with high cohesion and loose coupling (which should be evidenced by fewer & less complicated annotations)

An Overview of Design

- Encapsulation
- Abstraction
- Loose Coupling
- Cohesion
- Hierarchy

Encapulation

- Encapsulation is the clear separation of specification from implementation
- Clients should not be concerned with internal behavior
- Necessary for loose coupling
- Naturally fits with the notion of *software contract*

Abstraction

- Abstraction is a necessary component of encapsulation
- Allows us to strip away certain levels of detail when taking an external view of an object
- Hiding unnecessary detail allows us to focus on the essential properties of an object

Loose Coupling

- Coupling is a measure of the connections between objects
- Highly couple objects interact in ways that make their separation difficult
- Undesirable, high levels of coupling arise when abstractions are poor and encapsulation inadequate
- In SPARK
  - appearance of a package name only as a prefix in an *inherits* annotation indicates weak coupling
  - appearance in a *global* or *derives* annotation indicates strong coupling
  - when excessive coupling results, it will be revealed by the size and complexity of SPARK's *derives* and *globals* annotations

Coupling

- Insert examples of strong/weak coupling
  - see Boiler Water monitor (valve type inclusion is weak coupling)

Cohesion

- Whereas coupling is measure between objects, cohesion is a property of an object
- Cohesion is a measure of focus or singleness of purpose

Hierarchy

- Certain objects are contained inside others and cannot be reached directly
- Establishing a clear hierarchy, encouraged and checked by the rules of SPARK contributes to the goal of keeping the flow of information under control -- loose coupling

Key Building Blocks

INFORMED designs can be implemented using just three key building blocks…

- Main programs
- Variable packages
- Type packages

Main Program

The main procedure forms the root of computation

- A main program is the top-level, entry point controlling the behavior a system or sub-system.
- Typically the SPARK implementation of an INFORMED design will have only one such main program (annotated with SPARK's `main_program` annotation); however, it is possible to a build system where each scheduled thread is regarded as a main program with the scheduling taking place outside the SPARK system boundary.

Main Program Example

```
with A, B, C;
--# inherit A, B, C, D;
--# main_program;
procedure Main
--# global in out A.State, B.State, ...
--# derives ...
is
   procedure Initialize;
   ....
begin -- Main
   Initialize;
  loop
   ControlProcedure;
  end loop;
end Main;
```

Main Program Issues

- For anything other than very small systems, the control procedure is likely to be decomposed into several smaller procedures.
- A useful aim here is to make each such decomposed procedure responsible for a single "mode" of the system's behavior.
- Sometimes SPARK programs do not to have a main program at all.
- In these cases top-level control is provided by some scheduler that is either custom written or provided by the operating system.
- In these cases a main program might be used as a substitute for the scheduler for analysis purposes.

Variable Packages

A variable package has state

- A variable package is a SPARK package that contains static data or "state" as revealed by an own variable annotation.

Variable Package Example

A variable package has some internal state indicated by the own annotation (which may also represent an abstraction of the internal state)

```
package Stack
--# own State;
is
procedure Clear;
--# global out State;
--# derives State from ;
procedure Push(X : in Integer);
--# global in out State;
--# derives State from State, X;
procedure Pop(X : out Integer);
--# global in out State;
--# derives X, State from State;
end Stack;
```

Abstract name of internal state (no details of implementation revealed)

Indicates (abstractly) how the internal state is affected by the procedure

Type Packages

A type package provides types or helper sub-programs

- A type package does not have state and therefore does not require an `own` variable annotation
- Exposes a concrete type, a private type, or a limited private type
  - recall: a limited private type is a private type with keyword `limited` -- this disallows the use of equality and assignment on elements of that type (Barnes pp. 173-174).
- For a private type, the package will also provide a set of operations that may be performed on that type

Type Package Example

```
package Stack
is
type T is private;
procedure Clear(S : out T);
--# derives S from ;
procedure Push(S : in out T;
               X : in Integer);
--# derives S from S, X;
procedure Pop(S : in out T;
              X : out Integer);
--# derives X, S from S;
private
--# hide Stack;
-- full Ada declaration of type T would go here
end Stack;
```

Declare type which clients use to declare their own stack state -- can now have multiple instances

State of stack now passed as an argument to each procedure

Using a Type Package

```
MyStack: Stacks.T;
```

A client declaring a private type of Stacks

```
MyStack: Stacks.T := Stacks.Empty;
```

Sometimes it is useful to expose particular values as "deferred constants" (actual value is defined in private part) giving a safe and useful value that can be used to initialize variables of that type. Alternatively, define a procedure that initializes a variable of the abstract type.

Type Packages w/ Concrete Types

Type packages declaring concrete types are required when visibility of a type declaration must be shared by other design elements

- Enumerated types are a common example -- clients need to know the specific values of the type to perform updates and to compare values

Concrete Types

Here we see an appropriate use of concrete types, where the types are appropriately separated into distinct packages

```
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 = Value.Open and Y = Switch.Off
   …
```

Concrete Types

The following strategy is less desirable due to decreased cohesion within package

```
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
```

This overloading is illegal in SPARK (values for two different types get referred to by the same name)

Using Child Packages

Using public children (SPARK 95) gives hierarchy along with meaningful naming

```
package Discretes
is
-- entry point for the package hierarchy only
end Discretes;
package Discretes.Valve
is
type T is (Open, Shut, Unknown);
end Discretes.Valve;
package Discretes.Valve.Butterfly
is
type T is (Open, Shut, Unknown);
end Discretes.Valve.Butterfly;
package Discretes.Switch
is
type T is (On, Off, Unknown);
end Discretes.Switch;
```

The body and private part of a child package can access the body and private part of its parent and the child does not need a with clause for its parent. Moreover, any with clause on the spec of parent implicitly applies to child as well (Barnes, p. 84)

…this is not the same as OO inheritance!

Utility Packages

A special case of "type package"...

- Although the main components of an INFORMED design are those described above there is sometimes the need for additional packages to provide shared services to them.
- Such utility packages never contain state variables (otherwise they would be variable packages) and do not declare types (otherwise they would be type packages).
- Utility packages are nowadays sometimes called "Class Utilities".
- The need for utility packages arises when an operation is required which affects or uses more than one variable package or type package and which it would not be appropriate to consider part of one or another.
- Care should be taken to prevent the proliferation of utility packages.

Boundary Variables

A special case of variable package…

- Boundary variables are particular kinds of variable package which provide interfaces between the software functionality described by the INFORMED design and elements outside it with which it must communicate.
- All communication across this system boundary is via boundary variables.
- Boundary variables provide an abstraction of communication and model the inherent concurrency associated with such communication.
- The entity with which the SPARK program communicates might be some kind of hardware sensor or actuator; or an "API" of some library or co-operating software system.

Boundary Variables

A simple example of a boundary variable package

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

Indicates that the package has some internal state. In this case, the state is not some private variable(s). Rather its some aspect of the environment (hardware) that this package is abstracting. One can think of this package/function as a stub whose actual functionality is filled in by a linking facility external to SPARK.

Boundary Variable Abstraction Layers

- It is often useful to place an abstraction layer between the boundary variables of a system and their users; this approach is appropriate where direct use of the boundary variables would provide insufficient abstraction allowing too much detail to become visible in higher level SPARK annotations.
  - E.g., the hardware interface actually consists of multiple registers and memory mapped ports, but we would like to treat those abstractly as a single entity
- Boundary variable abstractions exploit SPARK's refinement mechanism and use a SPARK package to provide indirect access to the boundary variables it is abstracting.
- These, lower-level, boundary variables are either embedded in the abstraction layer (SPARK 83 or SPARK 95) or are private children of it (SPARK 95 only).
- The abstraction may hide the fact that more than one boundary variable is involved in providing the abstract inputs or may hide some other processing such as calibration that is taking place.

Boundary Variable Abstraction

Example (from INFORMED Cycle Computer)

```
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;
```

Abstract state representing a sensor input stream (with both setting and mode information)

Boundary Variable Abstraction

Example (from INFORMED Cycle Computer)

```
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 package

Represents two different pieces of memory-mapped state

Boundary Variable Abstraction

Example (from INFORMED Cycle Computer)

```
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;
```

Single public state entity is refined into multiple state entities represented in private child packages

Refined info flow contracts indicate read of subcomponents on sensor stream

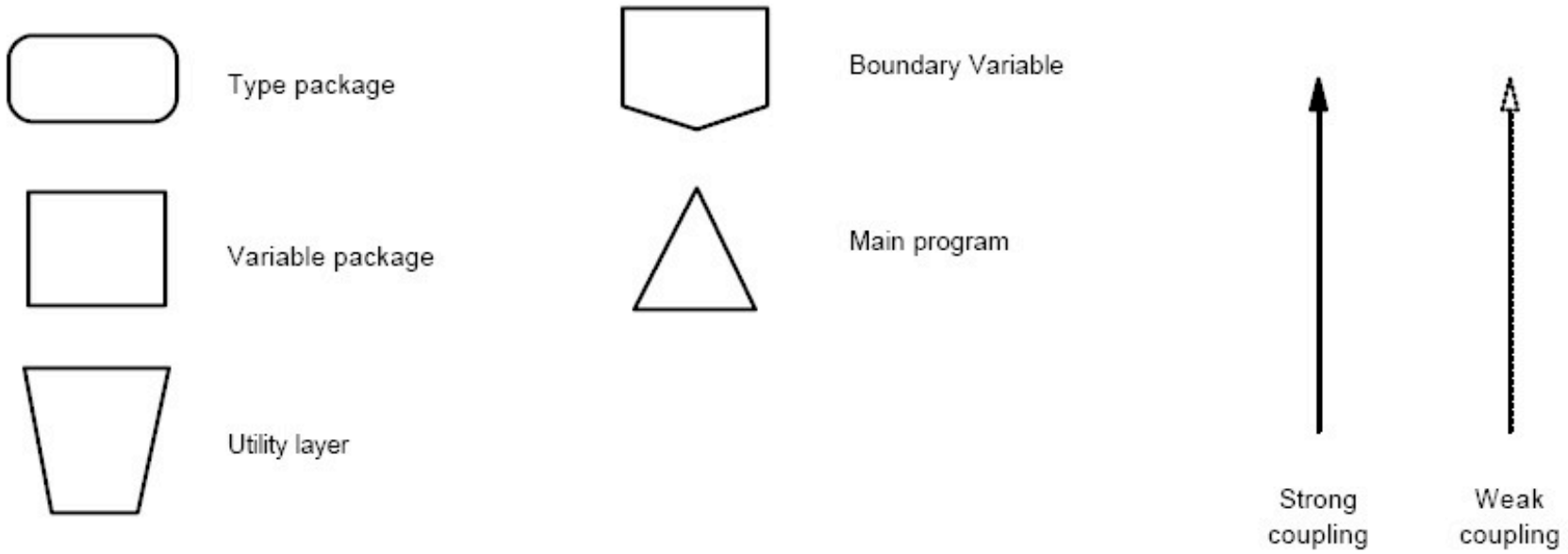
Boundary Variable Abstraction

Issues

- Often the creation of a boundary variable abstraction layer occurs from reasoning in the opposite direction to that described above, where we chose to insert a layer to provide an abstraction of two boundary variables.
- Instead we may have chosen a boundary variable to represent some logically significant input value and then found that it can be implemented in terms of more detailed, embedded, boundary variables.
- In this case we turn the package previously identified as a boundary variable into a boundary abstraction layer and refine it into one or more boundary variables (and perhaps some other processing code) (See room temperature control case study)
- A very important rule concerning boundary variable abstractions is *never* to mix input and output boundary variables in a single abstraction; this invariably leads to confusing and misleading information flow results where inputs incorrectly appear to depend on values previously sent as outputs.

SPARK Design Building Blocks

Graphical notation for SPARK packages and relationships



INFORMED Principles

The design steps outlined below are based on the following principles:

- application-oriented annotations;
- minimized information flow;
- clear separation of the essential from the inessential;
- careful selection of the SPARK implementation boundary; and
- early use of static analysis.

Application-Oriented Annotations

- SPARK annotations provide an expression of the behaviour of the system in parallel with the code itself.
- This description is most useful if it is expressed in problem domain terms rather than in implementation terms.
- For example we might prefer to see annotations in terms of “master switch” and “thermostat”rather than “register A” or “analogue to digital converter 2”.

Minimal Information Flow

- To reason about the behaviour of the software we will inevitably have to reason about the information that flows round it.
- This reasoning is simplified if the information that flows is the minimum that is necessary for the software to perform its task.
- Unnecessary "pumping" of data from one part of the system to another will increase the complexity of the information flows as revealed by SPARK's annotations.
- Methods for minimizing information flow include:
  - minimizing propagation of unnecessary detail;
  - localization and encapsulation of state;
  - avoiding making copies of data; and
  - appropriate use of hierarchy.

Clear Separation of the Essential from the Inessential

- Software designers have to reconcile many, sometimes conflicting, constraints.
- For example, good design might dictate that data is localised and encapsulated; however, the need to monitor data during rig testing of a system might require that data to be placed instead in some global location.
- An important principle of INFORMED is that the dictates of good software engineering, and providing the core functionality of the system in the most elegant way possible, must take priority.
- This does not mean that other peripheral needs are not addressed but that they should not be addressed in ways that unnecessarily distort the basic design.

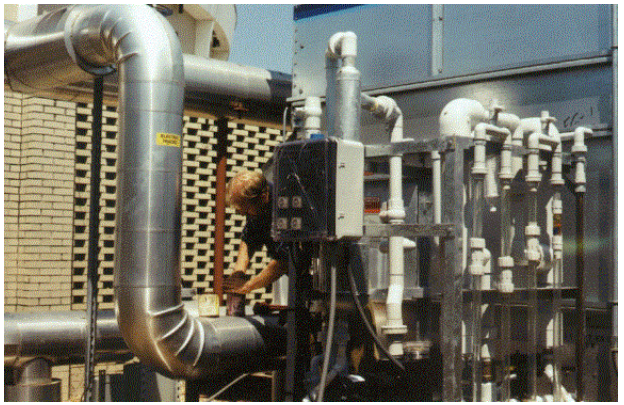
Careful Selection of the SPARK Implementation Boundary

- The above principles are facilitated if careful thought is given to where the boundary of the SPARK system (i.e. those parts of the system that will be seen by the Examiner) lies within the overall system boundary.
- It is not necessarily the case that everything that *could* be included in the SPARK system *should* be.
- For example, the goal of providing annotations in application domain terms may require selection of boundary variables that are higher-level and more abstract than the most fine-grained that could be expressed in SPARK.
- There are also situations in which the best implementation might consist of more than one SPARK system within the overall system boundary.

Early use of Static Analysis

- The design should be submitted to the SPARK Examiner as early as possible and the analysis process should continue as it evolves.
- The use of the Examiner in this way constantly checks the design choices that are being made and makes a major contribution to ensuring that the design aims are met.
- Early use of the Examiner is facilitated by the use of abstraction, by deferring implementation decisions, and by use of the Examiner's "hide" directive.
- Main point: Examiner can be applied to designs and/or incomplete code. Use this capability as early in development as possible.

Boiler Water Monitor



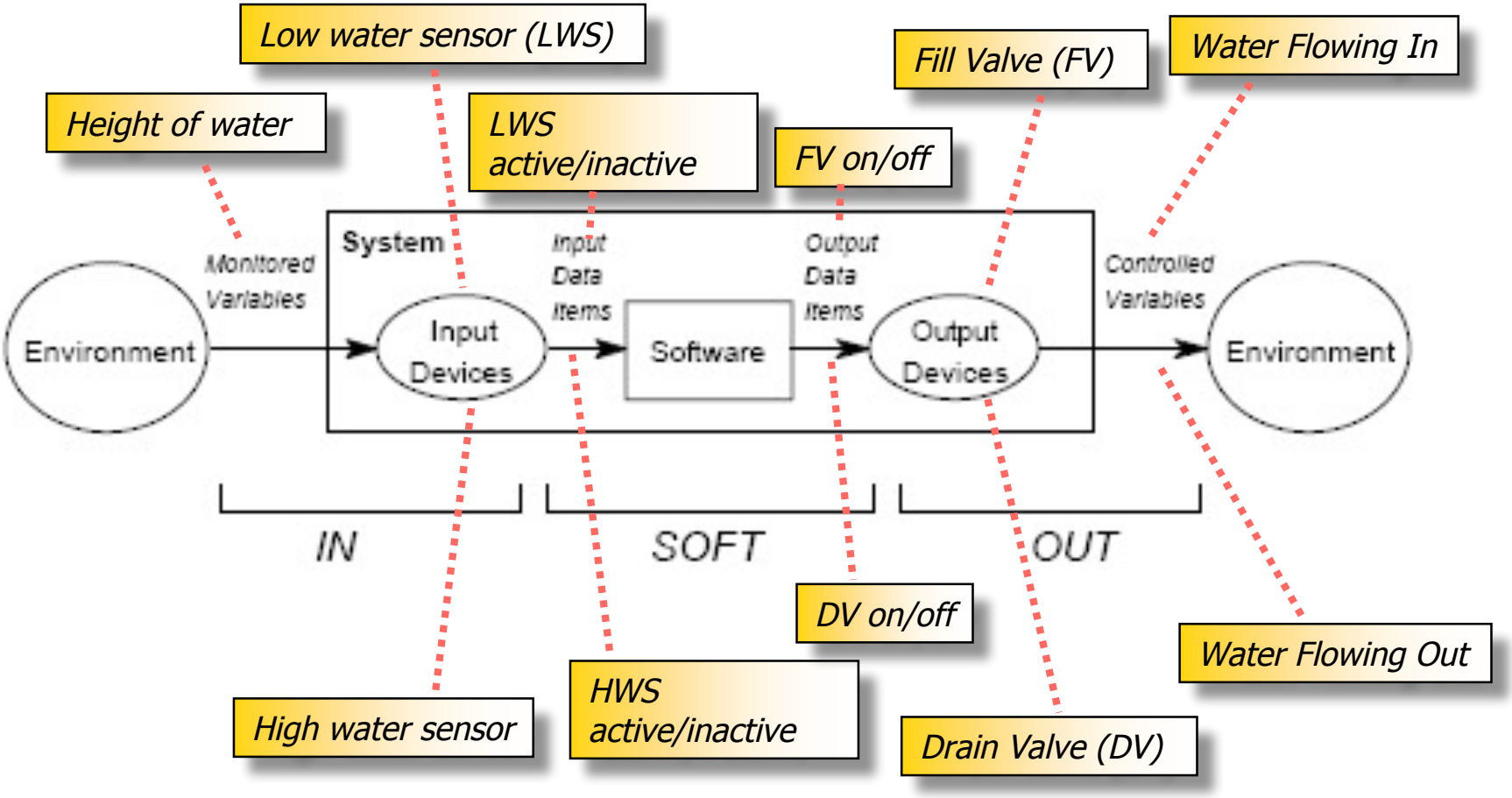
- Build a device to monitor the depth of water in a boiler vessel
- Two sensors are provided
  - water high
  - water low
- When water is low, a fill valve is to be opened
- When water is high, a drain valve is to be opened
- When neither high nor low signal is present, both valves are closed
- To prevent the valves from chattering, some delay of operation is required with the valves only operating after 10 successive consistent signals have been received from the associated sensor.

INFORMED Design Steps

(Step 1) Identification of the *system* boundary, inputs and outputs.

- 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.
  - What are the environmental qualities that influence the system's behavior (*monitored variables*)?
  - What are the environmental qualities that are influenced by (controlled by) the system (*controlled variables*)?

System Boundary



INFORMED Design Steps

(Step 2) Identification of the *SPARK* boundary.

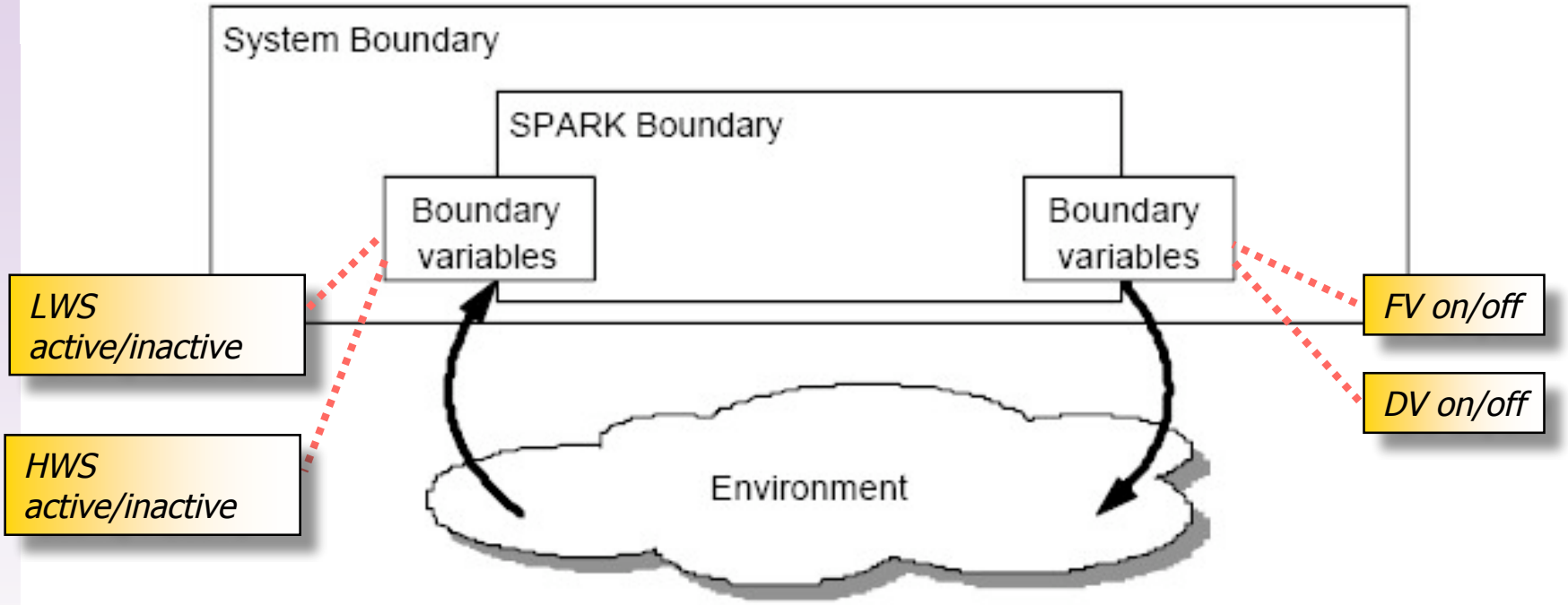
- Select a SPARK boundary within the overall system boundary; this defines the parts of the system that will be modelled and coded in SPARK and which will be subject to SPARK Examination.
- Define boundary variables to give controlled interfaces across the SPARK boundary annotated in problem domain terms.
- This step is tightly bound and iterative with the previous bullet.
- Consider adding boundary abstraction layers.

SPARK Boundary

Issues

- Selection of the boundary variables effectively defines the SPARK system boundary.
- The abstracted input/output values provided by the boundary variables (and boundary variable abstraction layers) are the *Input Data Items* and *Output Data Items* of the Parnas model.
- Processing between the SPARK boundary and the System Boundary (and in abstraction layers) provides the processing described in the *IN* and *OUT* relations of Parnas.

SPARK Boundary

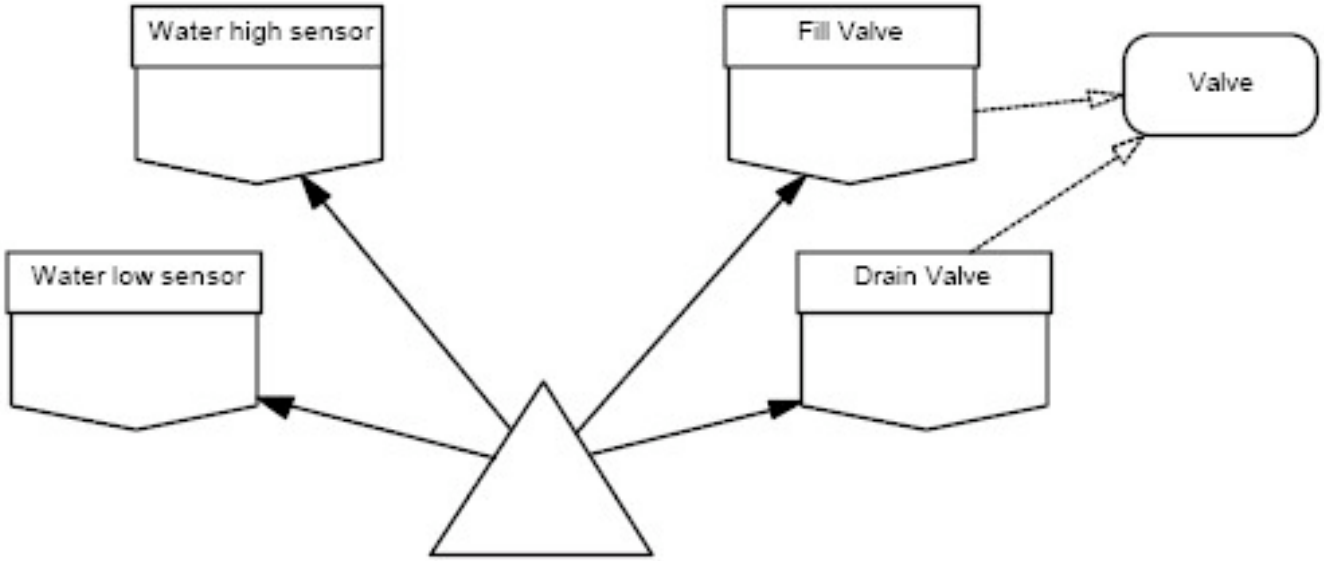


SPARK Boundary

Boiler Water Monitor boundary variables

- We therefore identify four boundary variables: “water high sensor”, “water low sensor”, “fill valve” and “drain valve”.
- Outside the SPARK boundary will be the implementations of those boundary variables and the precise way that the logical signal “water high” is obtained from the physical environment.
- Values of water high/low can be considered Boolean. However, valves are normally “open” or “shut”. Since we have two specific valves that share this characteristic, this suggests the need for a type package to represent this shared characteristic of valves.
- No need for boundary variable abstraction in this system.

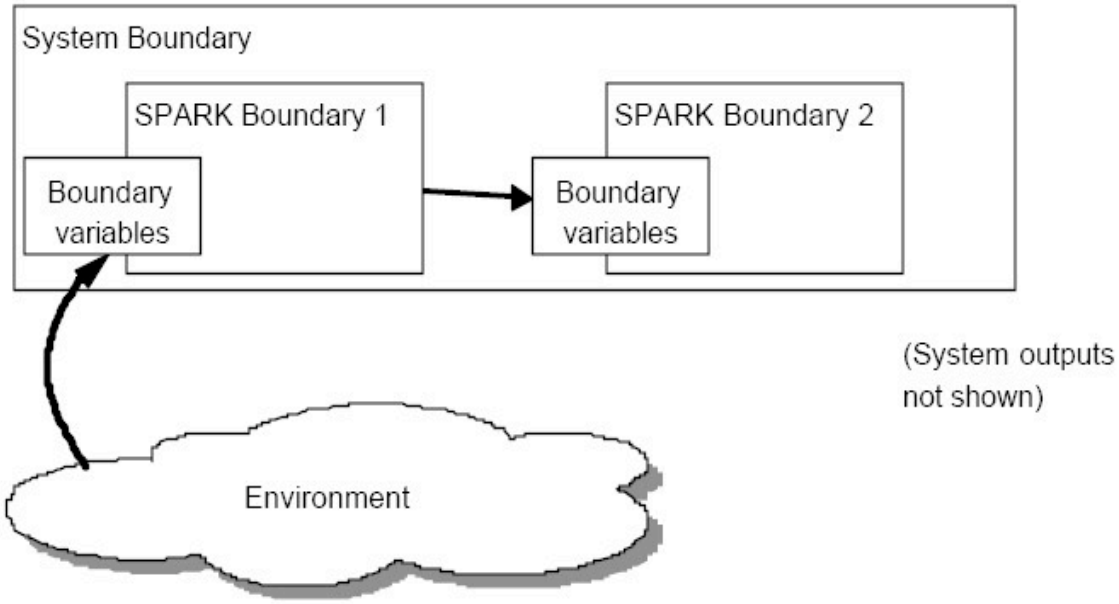
Boiler Water Monitor Design



SPARK Boundary

- Analysis may show that there is a benefit from having more than on SPARK system within the overall system boundary.
- This is most likely when there is a significant amount of input/output processing required
  - Although this could be implemented within SPARK there are advantages in change the scale or level of detail at which the data is considered in different parts of the system.
- One SPARK process can have relatively detailed boundary variables and produce a set of outputs which are read in more abstract form by a boundary variable of a second SPARK process.
  - Often this can all be done within a single SPARK boundary by using boundary variable abstraction layers and normal SPARK refinement; however, there are cases where a division into two subsystems with manual analysis of the join is necessary.

SPARK Boundary



- First SPARK subsystem might be monitoring data received over data bus (e.g., temp & pressure)
  - boundary variables concerned with terminal sub-addresses and other facets of bus communication
  - sub-system could be implemented in SPARK and analyzed to show that received messages were derived from the data bus as expected
- Second SPARK subsystem might implement a control algorithm that depended on temps and pressures obtained from the bus.
  - Second separate subsystem allows to phrase data in terms of logical temps and pressures instead of bus values
- This approach requires manual inspection to verify the border between first/second system.

Benefits of Two SPARK subsystems

- Increased encapsulation of the interface details because the existence of the data bus is no longer visible at the control algorithm level;
- Simplified reasoning about the behaviour of the control algorithm because the information we are processing is presented at an appropriate level of abstraction; and
- More meaningful annotations of the control algorithm with, for example

  --# **derives** Heater.State **from** Temperature.State &
  ReliefValve.State **from** Pressure.State & ...

  replacing

  --# **derives** Heater.State,
  ReliefValve.State **from** Bus.State & ...
- See Cycle Computer example in INFORMED document for a larger example of a design with two SPARK sub-systems

INFORMED Design Steps

(Step 3) Identification and localization of system state

- Identify the essential state of the system; what must be stored?
- Decide in what terms we wish the annotation of the main program to be expressed.
- Any state outside the main program will appear in its annotations, any inside will not.
- Using these considerations, assess where state should be located and whether it should be implemented as variable packages or instances of type packages.
- Identify any natural state hierarchies and use SPARK refinement to model them.
- Introduce utility layers where appropriate.
- Test to see if the resulting provisional design is a loop-free partial ordering of packages and produces a logical and minimal flow of information. Backtrack as necessary.

Boiler Water Example

(Step 3) Identification and localization of system state

- Need to store counts of the number of times the sensors have recorded the water value being high or low. We could locate this state in…
  - global variables used by main
  - inside sensor boundary variables
  - local variables inside main program

Boiler Water Example

(Option 1) Locate count variables as globals to main

- Create two "fault accumulator" variable packages
- Main program polls the raw sensor, and if event is registered, update the variable packages
- Boolean function provided by the variable packages indicates when sufficient number of events are recorded
- State of variable packages shows up in global/derives of main package
- *However, function of main program is to provide link between sensors and actuators -- not to push info into and read from the fault accumulators*
  - This solution introduces unnecessary info flow
  - Further objection is that two identifical variable packages would be needed -- one for each sensor.

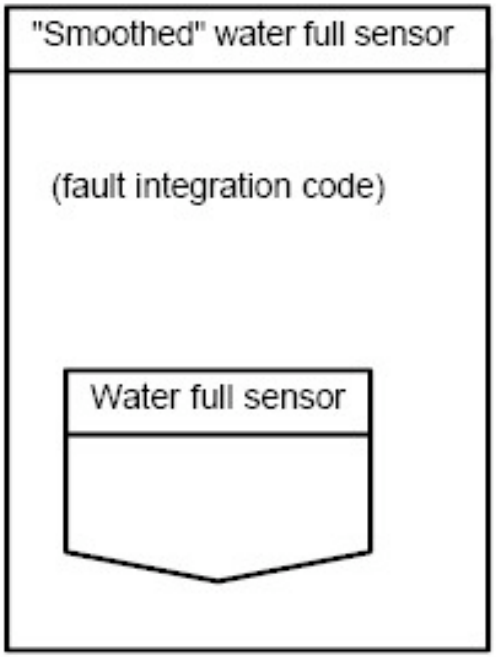
Boiler Water Example

(Option 2) Locate count variables inside sensor boundary variables

- Making the counting of occurrences part of the IN relation of the Parnas model
- This is a better solution because info flow of the main program would be expressed solely in terms of "smoothed" sensor stream values
- Downside: enlargement of boundary variables with identical fault integrator code in each
  - such duplicated code might also be outside the SPARK boundary and would therefore not be subject to static analysis
- A related option is to perform the fault integration in variable packages which enclose the boundary variables (either by embedding (shown on next slide) or by making the boundary variable a private child -- however, this would still yield code duplication.
  - However, this is a useful technique in general (see Heating Control System example)

Boiler Water Example

(Option 2) Fault integration in variable packages which enclose the boundary variables by embedding



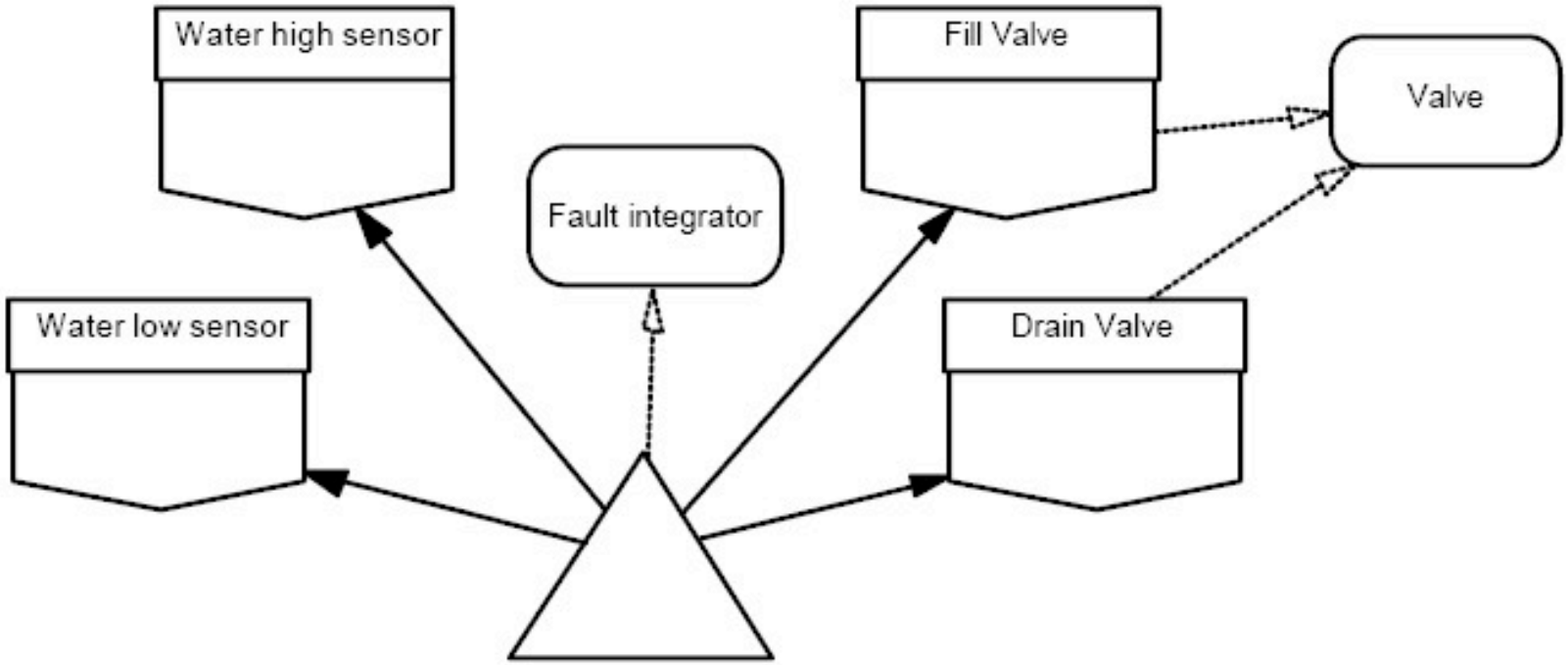
Boiler Water Example

(Option 3) Locate count variables as local variables inside main program

- We desire main program's annotation to be in terms of sensors and actuators -- representing as local variables would achieve this
- Also allows us to establish the correctness of the accumulation logic
- How to represent count history in main?
  - Could use e.g., integer, local variables and increment/decrement them explicitly
    - would involve code duplication and force early commitment to a particular representation
  - Could use variable packages as fault integrators embedded inside main program
    - would achieve abstraction and keep fault integrator logic out of main annotations
    - remaining drawback: code duplication -- two such packages would be needed
  - Declare two variables local to main program, which are instances of a type package
    - locates state in the correct place and allows sharing of fault integration code

Boiler Water Example

(Option 3) Declare two variables local to main program, which are instances of a type package



Identification and Localization of System State

Issues

- Most realistic, useful systems store data values in variables and therefore have "history" or "state".
- Selecting appropriate locations for this state is probably the single most important design decision that influences the amount of information flow in the system.
- The decisions involve deciding *what* must be stored, *where* must it be stored and *how* should it be stored.
- The significance of these decisions cannot be overestimated: it is the presence of system state that causes most of the complexity in understanding, analysing and testing code.
- In the absence of state, invoking an operation with a particular set of arguments always returns a consistent answer; in the presence of state the answer may also depend on some complex history of all previous calls.

What must be stored?

Issues

- Some static data storage is likely to be unavoidable but the amount should be reduced as far as possible.
- An important principle is to avoid data duplication: data should be stored where it can be made be available to its users by means of suitable "accessor functions" rather than by sending copies to be stored by potential users.
- This guidance does not prohibit the passing of data as actual parameters to an operation; it is the static storage of copies of data that should be avoided.
- A clear distinction should be made between state essential to the core functionality of the application and that added in for peripheral reasons such as to allow monitoring during dynamic testing.

Where should it be stored?

Issues

- Data stored inside the main program does not appear in its annotations; this principle could be abused because, if all boundary variables and other state were embedded in the main program, it would have the uninformative dependency relation "--# derives ;".
- As a first guideline, abstract own variables representing the external environment and state variables that we want to show the main program influencing should be outside it.
- Other, incidental, state should be inside the main program.

How Should It Be Stored?

Ways to store *static* data

We can store *static* data in a number of ways:

- in a variable package at library level (global state);
- in a variable package embedded in (or a private child of) another variable package (hierarchical state refinement);
- in a variable package embedded in the main program;
- as an instance of a type package; or
- as a concrete Ada variable.

Boiler Water Example

Water High Sensor Boundary Variable

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

Boiler Water Example

Water Low Sensor Boundary Variable

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

Boiler Water Example

Valve Type Package

```
package Valve
is
      type T is (Open, Shut);
end Valve;
```

Boiler Water Example

Fill Value Boundary Variable

```
with Valve;
--# inherit Valve;
package FillValve
--# own out State;
is
      procedure SetTo(Setting : in Valve.T);
      --# global out State;
      --# derives State from Setting;
end FillValve;
```

Boiler Water Example

Drain Valve Boundary Variable Example

```
with Valve;
--# inherit Valve;
package DrainValve
--# own out State;
is
     procedure SetTo(Setting : in Valve.T);
     --# global out State;
     --# derives State from Setting;
end DrainValve;
```

Boiler Water Example

Fault Integrate Type Package

```
package FaultIntegrator
is
        type T is limited private;
        procedure Init(FI : out T;
                                Threshold : in Positive);
        --# derives FI from Threshold;
        procedure Test(FI : in out T;
                                CurrentEvent : in Boolean;
                                IntegratedEvent : out Boolean);
        --# derives IntegratedEvent,
        --# FI from FI, CurrentEvent;
private
--# hide FaultIntegrator;
end FaultIntegrator;
```

Boiler Water Example

Main

```
with WaterHighSensor,
     WaterLowSensor,
     Valve,
     FillValve,
     DrainValve,
     FaultIntegrator;
--# inherit WaterHighSensor,
--#          WaterLowSensor,
--#          Valve,
--#          FillValve,
--#          DrainValve,
--#          FaultIntegrator;
--# main_program;
  …
```

Boiler Water Example

Main

```
procedure Main
--# global in                WaterHighSensor.State,
--#                          WaterLowSensor.State;
--#                 out FillValve.State,
--#                          DrainValve.State;
--# derives FillValve.State
--#                          from WaterLowSensor.State &
--#                 DrainValve.State
--#                          from WaterHighSensor.State;
Is

      HighIntegrator,
      LowIntegrator : FaultIntegrator.T;

      HighThreshold : constant Positive := 10;
      LowThreshold : constant Positive := 10;
```

Boiler Water Example

Main

```
procedure Control
--# global in          WaterHighSensor.State,
--#                            WaterLowSensor.State;
--#                    out FillValve.State,
--#                            DrainValve.State;
--#                    in out HighIntegrator,
--#                            LowIntegrator;
--# derives FillValve.State,
--#             LowIntegrator
--#                    from LowIntegrator,
--#                            WaterLowSensor.State &
--#             DrainValve.State,
--#             HighIntegrator
--#                    from HighIntegrator,
--#                            WaterHighSensor.State;
is
--# hide Control;
end Control;
```

Boiler Water Example

Main

```
begin -- Main
        FaultIntegrator.Init(HighIntegrator, HighThreshold);
        FaultIntegrator.Init(LowIntegrator, LowThreshold);

        FillValve.SetTo(Valve.Shut);
        DrainValve.SetTo(Valve.Shut);

        loop
                Control;
        end loop;
end Main;
```

INFORMED Design Steps

(Step 4) Handling initialization of state

- Consider how state will be initialized. Does this affect the location choices made?
- Examine and flow analyse the system framework.

Boiler Water Example

(Step 4) Handling initialization of state

- Sensors are initialized in environment
  - associated own variables declared external
- Valves…
  - could be initialized by their boundary variables (presumably setting them closed)
  - but better if we choose explicitly to shut the valves at start of main program
    - simplifies reasoning and improves main program annotations (how?)
- Fault integrators
  - Since SPARK does not allow default values for types, some explicit initialization routine or deferred constant is needed.
  - We can use initialization routine to set hysteresis threshold for each instance of the type thus avoiding problems if the requirements for threshold # changes

Handling Initialization of State

Issues

- Having decided on the location and method of representing the persistent state of a system (i.e. state declared in packages as revealed by SPARK's own variable annotation), early consideration needs to be given to how it will be initialized.
- SPARK makes particular demands in this area which can have a significant effect on program designs.
- The SPARK rules are required in order to be able to check that there are no overall data flow errors at the main program level.

Handling Initialization of State

Initialization during program elaboration; the variable is considered to have a valid value prior to the execution of the main program

- By execution of statements in a package's elaboration part (i.e. between the begin and end of the package's body).
- By providing an initial value at the point of the variable's declaration:

  X : Integer := 0;

  Note that SPARK imposes limits on what may appear in the initialization expression.
- Implicit initialization by the external environment. Typically this is the case with boundary variables representing external data streams (see Section 3.5 and Appendix A). Other, less satisfactory, cases can occur where systems rely on RAM clearance to initialize state variables.

Handling Initialization of State

Initialization during execution of main program; The variable does not have a valid value prior to the execution of the statement concerned

- By an assignment statement. For concrete Ada variables any suitable expression may be assigned; for variables of type packages a suitable deferred constant or function call will be required.
- By a call to a procedure which exports (and does not import) the variable concerned; this is the only way of initializing a variable of a type package implemented as an Ada limited private type.

Handling Initialization of State

Initialization of Own variables

- Own variables initialized during elaboration are known as *initialized own variables* and are annotated with the SPARK --# initializes annotation, those initialized during program execution are not.
- The difference between initialization during elaboration and initialization after execution of the main program starts is fundamental and can be seen in the annotations of the main program.
- External own variables (i.e. those declared with a mode) are also regarded as being initialized prior to the execution of the main program.
- In this case the initialization is provided implicitly by the environment.
- Initialized own variables and external variables of mode IN *may*, and usually will, appear as imports in the main program's derives annotation.
- Other own variables *may not* appear as imports in the main program's derives annotation.

Initialized Own Variables

Consider transferring data from boundary variable "Input" to boundary variable "Output" via a global variable package "Queue"

Initializing Queue at elaboration

```
--# derives Output.State,
--#          Queue.State
--#          from Queue.State,
--#               Input.State;
```

We're updating both Output and Queue

We depend on both the Input and the initial state of the Queue (since it is already initialized, and we are not guaranteed that it is overridden by Input)

Initialized Own Variables

Initializing Queue by a subprogram call from within the main program

```
--# derives Output.State,
--#          Queue.State
--#              from Input.State;
```

We're updating both Output and Queue

Now we don't depend on initial state of Queue since we initialize it within Main

Initialized Own Variables

Queue does not need to be a global variable; it can be local to Main

```
--# derives Output.State,
--#          from Input.State;
```

Only updates to Output are visible

Now we don't depend on initial state of Queue since we initialize it within Main

INFORMED Design Steps

(Step 5) Handling secondary requirements

- Add in secondary requirements such as test points without unnecessary distortion of the core design.
- Examine and flow analyse the system framework.

Handling Secondary Requirements

Slide sub-title

- As mentioned in Section 4.3, software designers frequently have to reconcile conflicting requirements.
- Amongst these requirements are some which INFORMED describes as "secondary requirements" because, although they may be important to the success of the project, they are not derived from the core functionality of the system being designed.

INFORMED Design Steps

(Step 6) Implementing the internal behavior of components

- Implement the chosen variable packages and type packages using top-down refinement with constant cross-checking against the design using the Examiner.
- Repeat these design steps for any identified subsystems.

Implementing the internal behaviour of components

Slide sub-title

- Application of the INFORMED steps identifies components such as variable packages, boundary variables and type packages and their relationships.
- Initially only annotated specifications for these objects are required allowing early static analysis of the design.
- Eventually the stage is reached when the consideration must be given to implementing the desired behaviour of each object.
- The first step should always be to see whether decomposition into further, smaller INFORMED components is possible.

Requirements

Slide sub-title

- A device is needed to monitor the depth of water in a boiler vessel.
- Two sensors are provided “water low” and “water high”.
- When the water is low a fill valve is to be opened.
- When the water is high a drain valve is to be opened.
- When neither high nor low signal is present both valves are closed.
- To prevent the valves chattering some delay of operation is required with the valves only operating after 10 successive, consistent signals have been received from the associated sensor.

Code Display With Red Annotations

Sub-title

```
procedure Control
...
is
      RawTooFull,
      RawTooEmpty,
      TooFull,
      TooEmpty : Boolean;
begin
      RawTooFull := WaterHighSensor.IsActive;
      RawTooEmpty := WaterLowSensor.IsActive;
      FaultIntegrator.Test(HighIntegrator,
                           RawTooFull,
                           --to get
```

...font for code is Courier New

Font for annotations is Tahoma (italics)

Code Display With Red Annotations

Sub-title

```
procedure ControlHigh
--# global in     WaterHighSensor.State;
--#               out DrainValve.State;
--#           in out HighIntegrator;
--# derives DrainValve.State,
--#           HighIntegrator
--#               from HighIntegrator,
--#                    WaterHighSensor.State;
is
      RawTooFull,
      TooFull : Boolean;
begin
   RawTooFull := WaterHighSensor.IsActive;
```

font for code is Courier New

Font for annotations is Tahoma (italics)

Code Display With Red Annotations

Sub-title

```
procedure ControlLow
--# global in     WaterLowSensor.State;
--#               out FillValve.State;
--#           in out LowIntegrator;
--# derives FillValve.State,
--#           LowIntegrator
--#               from LowIntegrator,
--#                    WaterLowSensor.State;
is
      RawTooEmpty,
      TooEmpty : Boolean;
begin
   RawTooEmpty := WaterLowSensor.IsActive;
```

font for code is Courier New

Font for annotations is Tahoma (italics)

Code Display With Red Annotations

Sub-title

```
...
loop
      ControlHigh;
      ControlLow;
end loop;
...
```

...font for code is Courier New

Font for annotations is Tahoma (italics)

Code Display With Red Annotations

Sub-title

```
private
      type T is record
                  Limit : Positive;
                  Counter : Natural;
                  Tripped : Boolean;
            end record;
end FaultIntegrator;
```

...font for code is Courier New

Font for annotations is Tahoma (italics)

Code Display With Red Annotations

Sub-title

```
package body FaultIntegrator
is

      procedure Init(FI : out T;
                     Threshold : in Positive)
      is
      begin
            FI := T'(Limit => Threshold,
                     Counter => 0,
                     Tripped => False);
      end Init;

      procedure Test(FI : in out T;
```

...font for code is Courier New

Font for annotations is Tahoma (italics)

Building Heating System

Requirements

- A building has a number of rooms each containing one or more temperature sensors and one or more heaters.
- A central controller is required to switch the heaters on in each room that is below the desired temperature and off in rooms at or above the desired temperature.
- Where more than one sensor is provided in a room, the controller uses a mean of their readings.

Design Steps

(Step 1) Identify System Boundary

- The system boundary encompasses the rooms, temperature sensors and heater actuators.
- People lighting log fires in some rooms and opening windows in others are outside the system boundary.
- The physical inputs are the temperature of each individual sensor. The physical outputs are the actuator switches for each individual heater.

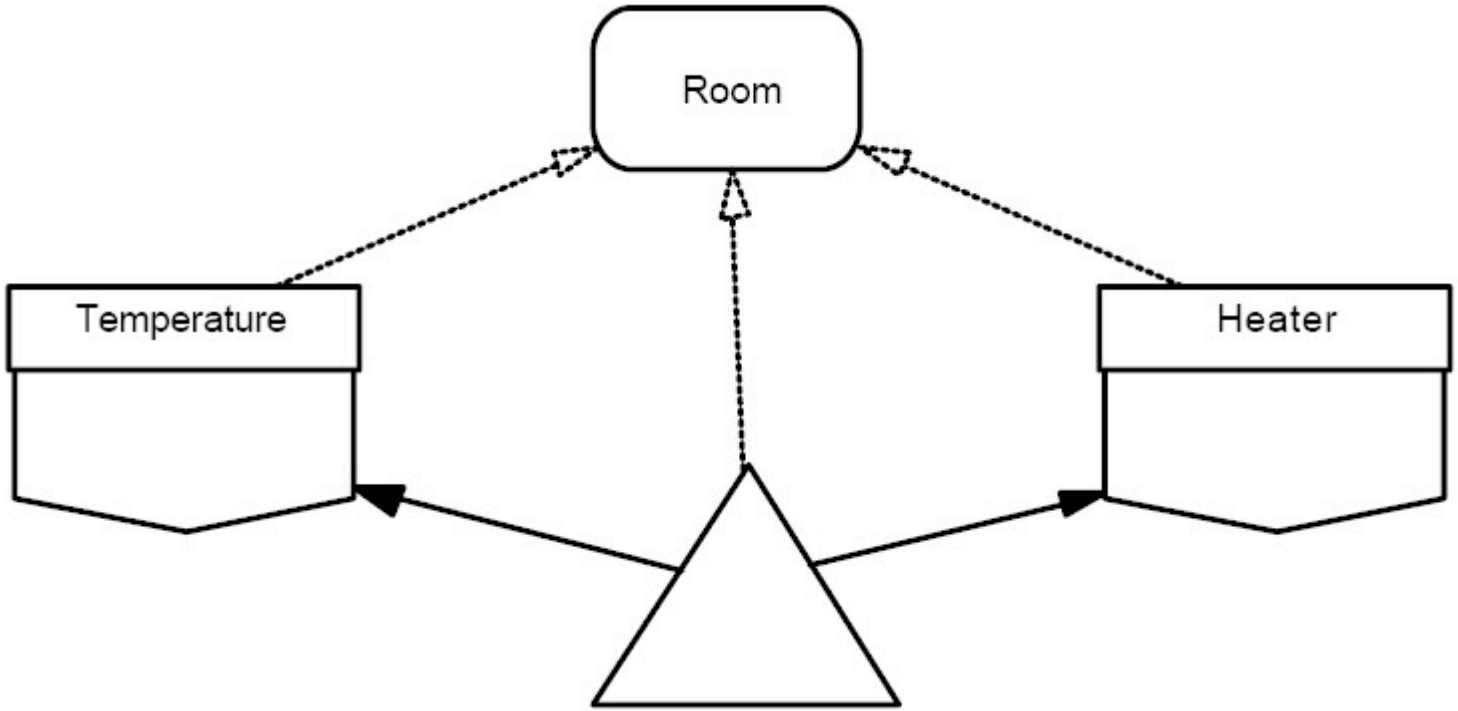
Design Steps

(Step 2) Identification of SPARK Boundary

- We need to be able to ask for the abstracted temperature (which may be the mean of several sensors) of any room and set a logical heat signal (which may turn on more than one heater) for any room.
- This suggests a boundary variable for "room temperatures" and one for "room heaters".
- Since both need to select the room they are dealing with they clearly share a "room" object which appears to be a type package.

Building Heating System

Architecture



Design Steps

(Step 3) Identification & Localization of System State

- The very simple control required by this system means that it does not have to retain any "state" variables; each room is checked in turn and its associated heater turned on or off as appropriate.
- We might choose to use "fault integration" as in the previous example but this has been omitted in this case for simplicity.
- We can already represent this design using INFORMED design element templates and flow analyse it.

Building Heating System

Initial Design

```
package Room
is
type T is (One, TheRest);
                    -- deferred decision on number of rooms
end Room;
```

Building Heating System

Initial Design

```
with Room;
--# inherit Room;
package Temperature
--# own State;
    -- represents both sensors and whatever internal state proves to be needed
--# initializes State;
is
type T is range -30 .. 60; -- Celsius
procedure Poll;
--# global in out State;
--# derives State from State;
function Reading(TheRoom : Room.T) return T;
--# global State;
end Temperature;
```

Building Heating System

Initial Design

```
with Room;
--# inherit Room;
package Heater
--# own State;
--# initializes State;
is
type T is (On, Off);
procedure SetSwitch(TheRoom : in Room.T;
                    Setting : in T);
--# global in out State;
--# derives State
--#           from State,
--#                TheRoom,
--#                Setting;
end Heater;
```

Building Heating System

Initial Design

```
with Room,Temperature, Heater;
use type Temperature.T;
--# inherit Room,
--# Temperature,
--# Heater;
--# main_program;
procedure Main
--# global in out Temperature.State,
--# Heater.State;
--# derives Heater.State,
--#                Temperature.State
--#     from *,
--#                Temperature.State;
is
```

Building Heating System

Initial Design

```
procedure Control
--# global in out Temperature.State, Heater.State;
--# derives Heater.State, Temperature.State
--#     from *, Temperature.State;
is
Desired : constant Temperature.T := 18;
begin -- Control
   Temperature.Poll;
   for TheRoom in Room.T loop
     if Temperature.Reading(TheRoom) < Desired then
       Heater.SetSwitch(TheRoom, Heater.On);
     else
       Heater.SetSwitch(TheRoom, Heater.Off);
     end if;
   end loop;
end Control;
```

Building Heating System

Initial Design

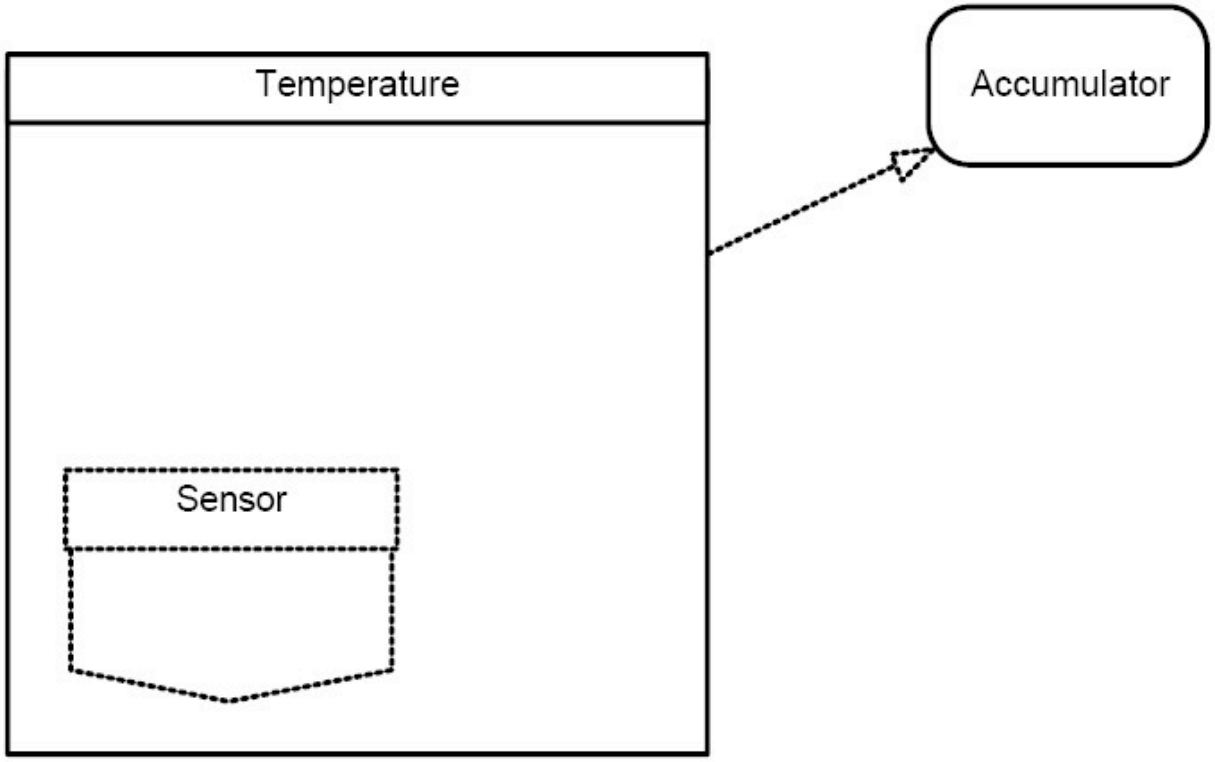
```
begin -- Main
   loop
      Control;
   end loop;
end Main;
```

Building Heating System

Assessment

- Although this is a complete and analysable outline design for the heating control system it ignores important details that need to be addressed in a complete implementation.
- In particular, it ignores the mapping of physical sensors and heaters to the rooms in which they are situated.
- We could decide that all this detail resided in the body of packages Temperature and Heater and therefore lay outside the SPARK boundary; however, this would unnecessarily restrict the portion of code which could be written in SPARK and analysed.
- Instead we decide to treat packages Temperature and Heater as boundary abstraction layers and *refine* them in terms of embedded (or private child) boundary variables.
- In the interests of flexibility we would clearly prefer the mapping of physical sensors/heaters to rooms not to be achieved by hard-coded control flow but rather by being treated as data.

Building Heating System



Building Heating System

We need to share notion of "temperature" between accumulator and Temperature package, so make it into a type package

```
package Celsius
is
type T is range -30 .. 60;
end Celsius;
```

Building Heating System

Define Accumulator

```
with Celsius;
--# inherit Celsius;
package Accumulator
is
   type T is private;
   Clear : constant T;
   procedure Add(TheAccumulator : in out T;
                 Value : in Celsius.T);
    --# derives TheAccumulator from *, Value;
   function Mean(TheAccumulator : T) return Celsius.T;
private
--# hide Accumulator;
end Accumulator;
```

Building Heating System

Use refinement when constructing Temperature implementation

```
with Accumulator;
package body Temperature
--# own State is in Sensor.State, AccumulatedValue; -- refinement clause
--  the Sensor.State component is an external variable
is
   PhysicalSensorCount : constant := 10;
   type SensorNumber is range 1..PhysicalSensorCount;

-- define look up table for physical sensors to rooms
type SensorMapping is array(SensorNumber) of Room.T;
SensorToRoom : constant SensorMapping :=
     SensorMapping'(SensorNumber => Room.TheRest); -- mapping details deferred
-
- create store for accumulated values for each room (must be initialized because
-- abstract state is initialized)
type AccumulatedValues is array(Room.T) of Accumulator.T;
AccumulatedValue : AccumulatedValues :=
     AccumulatedValues'(Room.T => Accumulator.Clear);
```

Building Heating System

Embedded Sensor boundary variable package

```
-- embedded boundary variable to read physical sensors – could also be private child
--# inherit Temperature, Celsius;
package Sensor
--# own in State;
is
function Read (TheSensor : Temperature.SensorNumber)
                                                    return Celsius.T;
--# global State;
end Sensor;
package body Sensor is separate; -- and outside SPARK boundary
```

Building Heating Example

Temperature Poll Procedure Implementation

```
procedure Poll
--# global in  Sensor.State;
--#         out AccumulatedValue;
--# derives AccumulatedValue from Sensor.State;
is
  LocalAccu : Accumulator.T;
  AssociatedRoom : Room.T;
  SensorReading : Celsius.T;
begin
  AccumulatedValue := AccumulatedValues'(
      Room.T => Accumulator.Clear);
  for CurrentSensor in SensorNumber loop
      AssociatedRoom := SensorToRoom(CurrentSensor);
      LocalAccu := AccumulatedValue (AssociatedRoom);
      SensorReading := Sensor.Read (CurrentSensor);
      Accumulator.Add (LocalAccu, SensorReading);
      AccumulatedValue (AssociatedRoom) := LocalAccu;
end loop;
end Poll;
```

Building Heating System

Function to read mean of temperatures in room

```
function Reading(TheRoom : Room.T) return Celsius.T
--# global AccumulatedValue;
is
  begin
    return Accumulator.Mean(AccumulatedValue(TheRoom));
  end Reading;
end Temperature;
```

Building Heating System

Accumulator type (the declarations below go in the private part of the Accumulator package specification)

```
private
type AccumulatedTemp is range -300 .. 600;
-- allows for up to 10 sensors per room. Code would be more robust if the type bounds
-- were calculated in some way from a constant representing the maximum number of
-- sensors a room could have.
type T is record
   Count : Natural;
   Total : AccumulatedTemp;
end record;

   Clear : constant T := T'(0, 0);

end Accumulator;
```

Building Heating System

Accumulator Implementation

```
with Celsius; -- second with clause needed to allow the use type which follows
use type Celsius.T;
package body Accumulator is
   procedure Add(TheAccumulator : in out T; Value : in Celsius.T) is
     begin
       TheAccumulator.Count := TheAccumulator.Count + 1;
       TheAccumulator.Total := TheAccumulator.Total +
                               AccumulatedTemp(Value);
     end Add;
   function Mean(TheAccumulator : T) return Celsius.T is
       begin
         return Celsius.T(TheAccumulator.Total /
                          AccumulatedTemp(TheAccumulator.Count));
         -- to avoid a divide by zero we need to ensure that Add is always called at least once
         -- before Mean; otherwise we need some defensive programming here
end Mean;
end Accumulator;
```

Full Slide Graphic

Room	Number of sensors	Sensor number allocated
One	1	1
Two	2	2, 3
Three	1	4
Four	3	5, 6, 7
Five	1	8

Table 2: Heating example configuration data

Graphic explanation

…points of interest highlighted in red dashed boxes and supporting annotation is linked via a dashed arrow.

Title

Sub-title

```
package Room
is
type T is (One, Two, Three, Four, Five);
end Room;
...
PhysicalSensorCount :constant := 8;
...
SensorToRoom : constant SensorMapping :=
SensorMapping'(1 => Room.One,
2 | 3 => Room.Two,
4 => Room.Three,
5 | 6 | 7 => Room.Four,
8 => Room.Five);
```

Full Slide Graphic