A Language for Systems not Just Software

Peter Amey

Praxis Critical Systems
20, Manver Street
Bath, BA1 1PX, UK
+44 (0)1225 466991

pna@praxis-cs.co.uk

1. ABSTRACT

The *specification* and *implementation* of software-intensive systems have generally been viewed as separate processes with differing notations. There are good reasons for trying to use notations capable of bridging the gap between the two. The SPARK language was originally concerned solely with providing an unambiguous subset of Ada that was suitable for rigorous static analysis and formal verification. Evolution of SPARK's system of formal comments or *annotations* has resulted in a language which now provides parallel descriptions of required system behaviour and software implementation. Analyses performed by the SPARK Examiner bind these parallel descriptions together. The result, not foreseen by the original designers of SPARK, is a language that can be used to describe systems rather than just implement software.

1.1 Keywords

Programming languages, static analysis, specification, critical systems.

2. INTRODUCTION

Historically programming languages have largely been concerned with the *coding* of designs. The facilities of the language exist wholly in the software realm; expressing the behaviour of the system of which software forms a part is the role of some other notation such as UML, Z or HOOD [11]. This is inevitable given the genesis of programming languages which all grew of out of machine code, via assemblers, and were focussed on providing more powerful and abstract ways of expressing the desired behaviour of software.

There are strong reasons for wanting to bind system and software descriptions more closely together. The use of different notations for each creates a semantic gap that must be bridged as part of the verification process. Too large a gap means that bridging it is error prone and a small one suggests that the specification of the system has been written at a very concrete, code-like level. Ideally we need a single descriptive notation that captures both the desired behaviour of the system and implements it in software.

In parallel with the development of programming languages has come the development of program analysis tools. Originally these were concerned with the reverse engineering of legacy systems to try and understand and evaluate their behaviour. Examples of this approach are MALPAS [3] and SPADE [4] from the UK and some of the more recent ASIS-based [6] tools which have been developed. After producing SPADE, Program Validation Ltd (now wholly absorbed into Praxis Critical Systems) noted that experience with analysis tools indicated that the real issue was encouragement for good design in the first place. Well designed systems were straightforward to analyse whereas poorly designed ones defied even the most powerful analysis techniques. The answer was therefore not ever more powerful analysis tools, but early deployment of analysis as an integral part of the development process leading to "correctness by construction". This philosophy was embodied in the SPARK[1] language [2, 7, 8] and its analysis tool the SPARK Examiner.

The earlier deployment of static analysis has driven SPARK evolution to make it more suitable for describing systems rather than just software.

[1] **Note**: The SPARK programming language is not sponsored by or affiliated with SPARC International Inc. and is not based on SPARC™ architecture.

3. RATIONALE OF SPARK

Key goals for designers of SPARK were to provide *precise* static analysis and facilitate its *early* use. *Precision* required that the language definition should be completely free from ambiguity and implementation dependence (while remaining, from a compilation viewpoint, a true subset of Ada). *Early use* required that analysis was computationally efficient and could be performed on incomplete programs. Both objectives could be met by strengthening package and subprogram interface specifications by means of formal comments or *annotations*. The concept of using strong interface specifications is shared with the Eiffel [9] language and is sometimes termed "programming by contract". Ada's subprogram specifications are not sufficient alone because they are designed to provide enough information for *code generation* rather than for *code analysis*.

A subprogram's Ada signature together with its SPARK annotations provides a sufficiently complete description of its behaviour to allow analysis of calls to it without the need to inspect its implementation. Analysis therefore only requires the access to the *specifications* described by the annotations. The annotations are checked for accuracy when the subprogram bodies are themselves analysed.

Annotations also strengthen the language definition and allow detection of conditions that would lead to implementation dependency and ambiguous behaviour. For example, SPARK requires that direct or indirect use of non-local data is indicated by means of a global annotation; this can be viewed as an extension of a subprogram's parameter list. For example, consider a subprogram that increments its parameter but also increments a global variable called `CallCount`. In SPARK this would be expressed thus:

```
procedure Inc (X : in out Integer);
--# global in out CallCount; -- global annotation
-- note the way annotations are introduced with --# making
-- them  comments as far as the compiler is concerned
```

As well as describing the behaviour of `Inc` strongly enough to allow analysis of calls to it, the annotation allows computationally efficient detection of conditions that could lead to erroneous behaviour.

3.1 Function side effects

Allowing functions to have side effects introduces order-of-evaluation dependencies which undermine the design goals of SPARK. SPARK therefore prohibits functions from having such behaviour. The global annotation provides a very efficient way of detecting such abuses. If we attempt to write a function that returns the increment of its argument and implement it by using procedure `Inc`:

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

then the annotation of `Inc`, which shows `CallCount` being exported, clearly reveals function `AddOne` as having a side effect. The body of `Inc` is not required to detect the side effect and no analysis of the full call tree is required.

3.2 Aliasing

Aliasing, the simultaneous referencing of a single location in memory via more than one name, can introduce erroneous behaviour which depends on implementation decisions such as the choice of parameter passing mechanism. SPARK prevents aliasing by prohibiting the overlapping of exported global variables and parameters. Again annotations provide an efficient detection mechanism. Consider the call: `Inc (CallCount);` The global annotation makes it clear that this call involves aliasing between the parameter `X` and the global variable `CallCount`. Since both are scalar there is no implementation dependency in this particular case but the example shows the principle of alias detection via inspection of global annotations and parameter lists.

3.3 Own Variable Clauses

The principle of providing specifications strong enough to describe behaviour, without the need to access package bodies, required an annotation to indicate the presence of static "state" variables in package bodies. Consider an implementation of the increment procedure introduced above:

```
package body P
is
   CallCount : Integer := 0;

   procedure Inc (X : in out Integer)
   is
   begin
      X := X + 1;
      CallCount := CallCount + 1;
   end Inc;
end P;
```

The variable `CallCount` is, as we would expect, declared in the body of the package `P` where it can only be manipulated by the supplied subprograms. Unfortunately, our specification of `Inc` has already referenced `CallCount` in its global annotation so we have a reference before declaration which is contrary to the spirit of Ada. The SPARK solution to this dilemma was the introduction of the "own variable" annotation. This annotation states that the package owns a variable and forms an announcement that the variable will later be declared in the package body. It serves as a declaration *for annotation purposes only*, allowing use of the variable in annotation context but not in Ada context. The specification of package `P` is therefore:

```
package P
--# own  CallCount;          -- own variable clause
--# initializes CallCount;
-- indicates that the variable is initialized at elaboration
is

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

end P;
```

4. EVOLUTION OF SPARK ANNOTATIONS

It is clear from the outline above that the initial design objectives of SPARK could be met by the introduction of annotations which were very code-oriented. All the items appearing in the own variable and global annotations above can be found in parallel Ada declarations. Indeed, some early critics of SPARK [12] attacked it precisely because of the apparent repetition of information that could, given a whole program analysis, ultimately be deduced from the code itself. Such criticisms overlooked the considerable benefits arising from computationally efficient analysis of incomplete programs but nevertheless contained a germ of truth.

4.1 The concept of refinement

The specification of package `P` reveals that its body contains a variable called `CallCount`; this can be viewed as a weakening of the abstraction and as preventing the deferment of the design decision of how the call counter will be implemented. The problem becomes more serious when the package state is more complex than a simple integer variable. An abstract state machine stack package, for example, clearly contains some state and therefore requires an own variable annotation; however, we don't want to reveal the exact mechanism used to implement the stack. The implementation might be an array and a stack pointer indexing into that array or it might perhaps be a linked list; these are not valid concerns of users of the stack package and should remain internal to it. Fortunately, the design goals of SPARK only require that we are aware of the *presence* of state variables in a package, we do not require the exact representational details. This property was exploited by the introduction (nearly 10 years ago) of the idea of *abstract own variables*. These are names, chosen by the software designer, to describe the presence of state whose concrete details remain hidden in the package body. Abstract own variable annotations are syntactically identical to the concrete own variables described above. The distinction occurs in the package body where, instead of simply declaring the own variable as an Ada variable, a *refinement annotation* is provided to bind the actual package data variables to the abstract name. Thus a stack package can admit that it contains some state without revealing that it comprises an array and a pointer, or perhaps a linked list, which actually implements the stack. For example:

```
package Stack
--# own State;   -- abstract own variable declaration
is
   procedure Clear;
   --# global out State;
   -- annotations in terms of the abstract state

   procedure Push (X : in Integer);
   --# global in out State;

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

end Stack;
```

In the body we refine the abstract own variable `State` into the array and pointer that we use to implement the stack:

```
package body Stack
--# own State is Vector, Ptr;
-- refinement clause defining constituents of State
is
   MaxDepth : constant := 100;
   type Ptrs is range 0 .. MaxDepth;
   subtype Indexes is
     Ptrs range 1 .. MaxDepth;
   type Vectors is array (Indexes)
     of Integer;

   -- declaration of variables
   -- announced by refinement clause
   Ptr    : Ptrs;
   Vector : Vectors;

  ...
```

There is a traceable refinement path from the abstract own variable `State` right through to the Ada variables `Ptr` and `Vector` that it represents. Outside the stack package we know only about `Stack.State` and inside it we only concern ourselves with `Ptr` and `Vector`. Note that the refinement constituents can themselves be own variables of embedded or private child packages allowing the construction of complex hierarchies of state.

The introduction of abstract own variables is a crucial step in the evolution of a language which allows the description of systems rather than just software. For the first time we have, in an annotation, an entity which cannot be found anywhere in the compilable Ada statements of the program. Indeed, the rules of SPARK prohibit the use of the identifier `State` once it has been "refined away". `Stack.State` is a system design concept not a software implementation concept.

5. SPARK AND THE "REAL WORLD"

Real systems are not the elegant "terminating sequential processes" beloved of computing theoreticians. To accomplish anything useful a program must interact with its environment. For the kind of critical control system for which SPARK is often used this involves reading sensors and updating actuators. The concept of abstract own variables is well suited to describing such operations. Sensors and actuators are encapsulated in packages with abstract own variables providing a suitable name for the external source of the data being read or written. These packages form the boundary of the SPARK system and provide its interface to the external environment. The concept is very similar to that described in the Four-Variable Model of Parnas and Madey [10], see Figure 1. Environmental inputs are called *monitored variables* and the outputs are *controlled variables*. Devices on the periphery of the system transform monitored variables into *input data* items used by the control software to compute *output data* items which are themselves transformed by devices into the controlled variables that influence the environment.

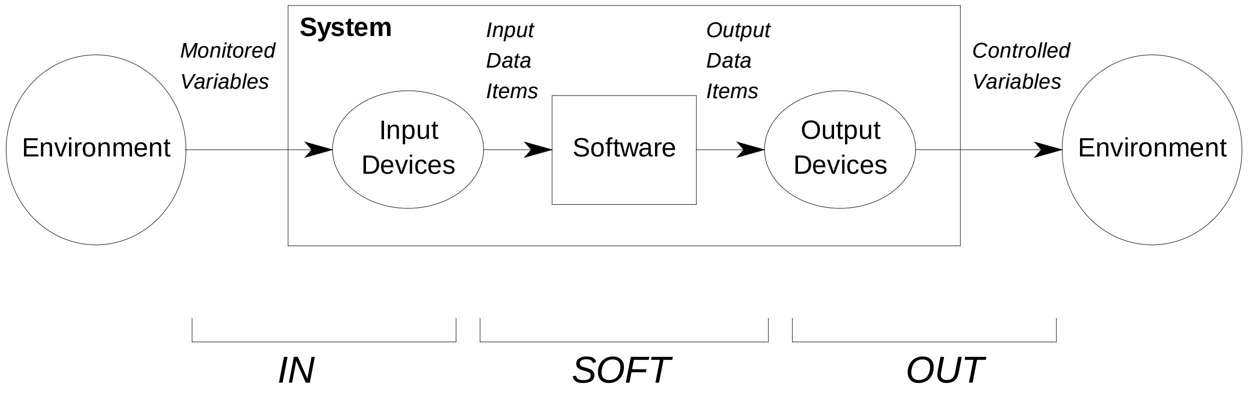

Figure 1

The composition of the *IN*, *SOFT* and *OUT* relations describes the overall effect of environmental inputs on environmental outputs.

In SPARK, the input devices of the above model become packages with abstract own variables representing the monitored variables. Subprograms exported by these packages provide the input data to the software. Similar packages represent the output devices and provide a means of writing to the controlled variables.

5.1 Volatility of system-level imports/exports

Although abstract own variables provide an effective way of naming external data sources and data destinations they fail to capture one important property of such entities: volatility. A characteristic of a monitored variable is that its value is generated not by the software or even the system of which the software is a part. Rather, it is generated by the external environment itself. This means that successive interrogations of such a variable may return different values even though the software has not updated it. This is quite different behaviour to the own variable `P.CallCount` and the abstract own variable `Stack.State` introduced earlier; these retain their values unless modified by a call to, for example, `Stack.Clear`.

SPARK users have traditionally exploited the way the Examiner accepts the correctness of annotations when analysing subprogram calls to deal with this volatility. Operations which read external devices are annotated so that they have an apparent side effect which modifies the monitored variable. To the Examiner it appears that each successive read returns a potentially different value from the last. In effect the monitored variable is treated as if it was an infinite sequence of possible future values. Reading the input returns the head of the sequence and has the side effect of removing it thus exposing a new value ready for the next read operation.

This approach is shown in the simple temperature sensor package that follows.

```
package Temperature
--# own Inputs;
--# initializes Inputs;
-- Inputs is an  abstract own variable representing the
-- monitored variable.
-- Values are provided (hence initialized) by the
-- environment.
is

   procedure Read (X : out Celsius);
   --# global in out Inputs;
   --# derives X      from Inputs &
   --#         Inputs from Inputs;

   -- The annotation shows that X is an input data item
   -- obtained from Temperature.Inputs and that there
   -- is a side effect on Temperature.Inputs giving the
   -- required volatility

end Temperature;
```

Although this is a serviceable approach that has been successfully used in many SPARK projects it is not without drawbacks. The principal problem is the counter-intuitive notion of procedure `Read` *updating* something which is conceptually an *input*. Even more difficult are the information flow couplings between such stream side-effects. For example, if we conditionally read a temperature depending on the value returned by a pressure sensor we find, unexpectedly, that `Temperature.Inputs` "depends on" `Pressure.Inputs`. An additional drawback is that we cannot analyse the body of package `Temperature` because we have deliberately "told lies" in its annotations to provide the desired input volatility. If `Read` obtained the value of parameter `X` by assignment from a memory-mapped port then static analysis would not be able to find the side effect described in `Read`'s annotation.

5.2 External "stream" variables

These considerations have led to the most recent change to the SPARK language and completed its evolution into a language that can be used to describe systems as well as their implementation in software.

The extension is to allow the addition of *modes* to own variable clauses. Like parameter (and global variable) modes these indicate a direction of data flow and directly identify an own variable as representing a system-level import or export. In the terms used by Parnas, an own variable of mode `in` is a *monitored variable* and one of mode `out` a *controlled variable*. The SPARK Examiner recognises the special characteristics of such variables and constructs the correct volatile behaviour for analysis purposes without the need for "unnatural" annotations. The result is that the unintuitive updating of inputs and reading of outputs is eliminated; stream side-effects do not become coupled together; and the bodies of external interface packages become amenable to analysis.

The temperature sensor presented earlier simplifies to:

```
package Temperature
--# own in Inputs;
-- mode in abstract own variable representing a
-- monitored variable
is
   procedure Read (X : out Celsius);
   --# global in Inputs;
   --# derives X from Inputs;
   -- X is an input data item obtained from
   -- Temperature.Inputs

end Temperature;
```

We now have all the tools we require to describe the behaviour of a software-intensive system that interacts with external devices. The executable Ada statements provide the operational semantics for a machine which we believe will have the desired behaviour. The SPARK annotations provide a parallel, abstract, system-oriented description of the system. The rules of SPARK and the analysis performed by the Examiner bind the two descriptions together. Nearer the leaves of the program call tree the annotations will be semantically closer to the code itself. Nearer the top of the call tree they will be more of an abstract description of the required system behaviour.

6. CASE STUDY

The case study is taken from [1] and takes the form of a water container with high and low level sensors together with fill and drain valves. A control system is required to keep the water level between the high and low levels. If too low the fill valve opens and if too high the drain valve opens. Integrators are used to prevent the valves "chattering" if the fluid is near the high or low level marks.

[1] pre-dates the introduction of moded external variables to SPARK and it is instructive to see how much simpler and clearer the example has become since their introduction.

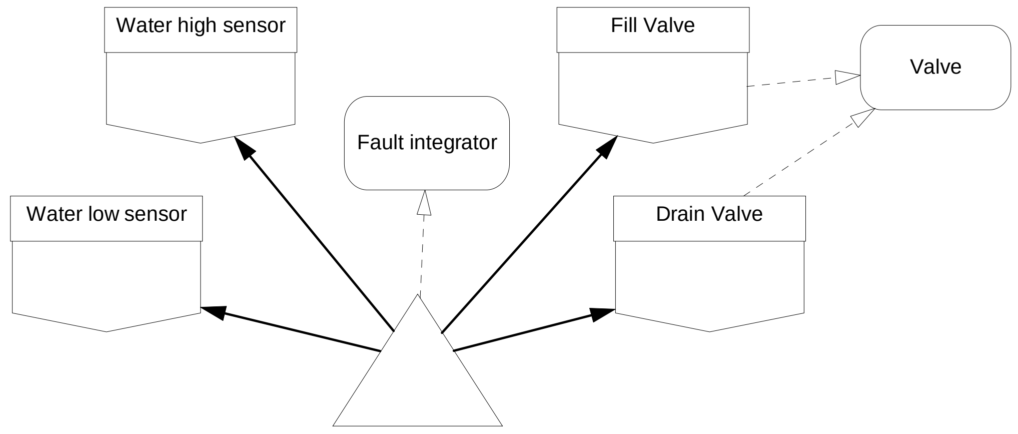

Figure 2

The structure of the proposed system is shown in Figure 2. The triangle is the main control program which uses "boundary packages" that read the sensors and update the valves. These are respectively the input devices and output devices of Parnas. Solid arrows represent strong data coupling between entities whereas the dashed arrows to the rounded boxes show weak coupling from the use of type declarations only.

The sensor packages follow the pattern established for the temperature sensor introduced earlier.

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

end WaterHighSensor;

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

end WaterLowSensor;
```

At this stage of the design we do not have to concern ourselves with the detail of how the sensor will be read. We may for example, take a majority vote of several sensors. The abstract description that we can obtain values from `WaterHighSensor.State`, a monitored variable, is adequate for us to implement the rest of the system.

The valve actuator packages are conceptually similar but the mode of the abstract own variables are, of course, `out` showing that they are controlled variables.

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

Package `Valve` simply defines a type shared by the fill and drain valve packages.

```
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;


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;
```

The fault integrator takes the form of an abstract data type. Its workings are not important for the purposes of this study but essentially it retains some state representing previous events and uses this, in conjunction with some instantaneous event, to decide whether an integrated event has occurred. It could, for example, be set so that the water full sensor did not trigger until a certain number of consecutive high signals had been received.

```
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;
```

Note that use has been made of another SPARK facility for the analysis of programs during their development: the hide directive. We do not have to commit ourselves to a representation of the fault integrator limited private type at this stage so we hide it from the Examiner. The program can be analysed but will not, of course, compile until the private part is completed.

We can now design the main control program. Noting that the fill valve and low water sensor are related as are the drain valve and the high water sensor, we use a separate control procedure for each pair. These are `ControlLow` and `ControlHigh` respectively. The main controller consists of a free running loop calling each in turn.

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

   procedure ControlHigh
   --# global in     WaterHighSensor.State;
   --#            out DrainValve.State;
   --#         in out HighIntegrator;
   --# derives DrainValve.State,
   --#         HighIntegrator from
   --#            HighIntegrator,
   --#            WaterHighSensor.State;
   is separate;

   procedure ControlLow
   --# global in     WaterLowSensor.State;
   --#            out FillValve.State;
   --#         in out LowIntegrator;
   --# derives FillValve.State,
   --#         LowIntegrator from
   --#            LowIntegrator,
   --#            WaterLowSensor.State;
   is separate;

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

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

   loop
      ControlHigh;
      ControlLow;
   end loop;
end Main;
```

Some observations can be made at this point. We have again deferred implementation, this time by making `ControlHigh` and `ControlLow` into subunits. Since their stubs are annotated we can still analyse the main control procedure and check that all the information flows are as expected. We could even *prove* properties of the main control loop if the annotations were strengthened with suitable pre and postconditions.

The top-level annotation of the main subprogram provides an entirely abstract description of the desired *system* behaviour:

```
--# derives FillValve.State  from
--#             WaterLowSensor.State  &
--#           DrainValve.State from
--#             WaterHighSensor.State;
```

which shows that the controlled variable `FillValve.State` depends only on the monitored variable `WaterLowSensor.State` and similarly for the other valve/sensor pair. Furthermore, the annotation shows that the valve/sensor pairs are independent of each other. Note that none of the items appearing in this description can be found anywhere in the program as Ada variables: it is a system-level description.

The slightly lower level procedures `ControlHigh` and `ControlLow` have annotations that contain a mix of system and software entities.

```
--# derives DrainValve.State,
--#         HighIntegrator from
--#            HighIntegrator,
--#            WaterHighSensor.State;
```

The monitored and controlled variables are present as before but we now also have `HighIntegrator` which is an Ada variable (albeit of an abstract type).

Having checked that the overall behaviour of the system is as required we can now implement `ControlHigh` and `ControlLow` and use the Examiner to check that their implementations match their annotations. Only `ControlHigh` is shown here.

```
separate (Main)
procedure ControlHigh
is
   RawFullEvent,
   TooFull : Boolean;
begin
   RawFullEvent :=
      WaterHighSensor.IsActive;
   FaultIntegrator.Test (HighIntegrator,
                         RawFullEvent,
                         -- to get
                         TooFull);
   if TooFull then
      DrainValve.SetTo (Valve.Open);
   else
      DrainValve.SetTo (Valve.Shut);
   end if;
end ControlHigh;
```

Finally, we can implement the bodies of the sensor and actuator packages and complete the fault integrator. A possible, rather simple implementation of the water high sensor might be.

```
with System.Storage_Elements;
package body WaterHighSensor
--# own State is in HighSensorPort;
-- refinement clause, maps our chosen monitored
-- variable name onto an Ada memory-mapped
-- variable of the same mode
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#);

   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 for sensor failure case: show
         -- water high
      end if;
      return Result;
   end IsActive;
end WaterHighSensor;
```

Note that at this low level in the subprogram calling hierarchy the annotation is entirely in software terms. It refers only to `HighSensorPort` which is a straightforward Ada variable.

Overall, however, we have complete traceability from low level implementation at the software level to high level specification at the system level:

`IsActive` *is obtained from* `HighSensorPort` *in the body of* `WaterHighSensor`

`HighSensorPort` *is a refinement of the abstract* `WaterHighSensor.State` *in the body of* `WaterHighSensor`

`IsActive` *is obtained from* `WaterHighSensor.State` *in the specification of* `WaterHighSensor`

and finally

`DrainValve.State` *is obtained from* `WaterHighSensor.State` *at the main program level.*

7. CONCLUSIONS

SPARK has its foundations in the analysis or reverse engineering of legacy software. Although it still embodies the fundamental analysis methods used for reverse engineering, it has evolved into something much more concerned with program construction rather than with program analysis. This migration has been guided by extensive experience of static analysis which clearly indicates that the maximum benefit accrues from its early deployment *during* program construction [5]. Such early deployment leads to error prevention rather than the more expensive error detection. It also allows the analysis to influence design. A crucial, but rather unsurprising, discovery is that well written programs are easy to analyse and test whereas badly written ones defy the best validation techniques available.

As SPARK has become more concerned with program construction rather than program analysis it has evolved notations to assist with describing desired system behaviour. The notion of, initially, abstract own variables and, latterly, the addition of own variable modes, has resulted in a programming language which can both *describe* and *implement* desired system behaviour. Furthermore, the rules of SPARK and the analysis performed by the Examiner, bind the system level description and software level together. The result is a language that describes systems, not just software.

8. REFERENCES

[1] Amey, Peter. The INFORMED Design Method for SPARK. Praxis Critical Systems[2] 1999.

[2] Barnes, John. High Integrity Ada - the SPARK Approach. Addison Wesley Longman, ISBN 0-201-17517-7.

[3] Bramson, B.D. Malvern's Program Analysers. RSRE Research Review 1984.

[4] Carré, Bernard. Program Analysis and Verification in High Integrity Software. Sennett, Chris (Ed). Pitman. ISBN 0-273-03158-9.

[5] Chapman, R.C. Industrial Experience of SPARK. Proceeding of SIGAda 2000. http://www.acm.org/sigada/conf/sigada2000/

[6] Cooper, C. Daniel. Ada Code Analysis: Technology, Experience, and Issues. Proceeding of SIGAda 2000. http://www.acm.org/sigada/conf/sigada2000/

[7] Finnie, Gavin et al: SPARK - The SPADE Ada Kernel. Edition 3.3, 1997, Praxis Critical Systems[2]

[8] Finnie, Gavin et al: SPARK 95 - The SPADE Ada 95 Kernel. 1999, Praxis Critical Systems[2]

[9] Meyer, Bertrand. Eiffel: The Language. Prentice Hall 1990. ISBN 0-13-247925-7.

[10] Parnas, D.L. and Madey, J. Functional Documentation for Computer Systems, in Science of Computer Programming. October 1995.

[11] Robinson, Peter J. HOOD - Hierarchical Object-Oriented Design. Prentice Hall, 1992.

[12] Rosskopf, Alfred. Use of a Static Analysis Tool for Safety-Critical Ada Applications. In Lecture Notes in Computer Science 1088, Reliable Software Technologies - Ada Europe 1996. Alfred Strohmeier (Ed). Springer 1996.

[2] Referenced Praxis Critical Systems documents may be obtained by sending an email request to `sparkinfo@praxis-cs.co.uk`