High Integrity Computing (F23PS2)

High Integrity Software Development (F29HD2)

The SPARK Approach (Part 2)

Andrew Ireland

Department of Computer Science
School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh

Overview

- More details on the SPARK language
- The SPARK Examiner and Ada compilation

Why Use Global Variables?

- The use of global variables is usually seen as **bad practice**.
- However, within the context of packages, global variables can have a useful role to play in terms of expressing state information.
- SPARK is typically used within embedded software systems – packages map onto physical devices, *e.g.* controllers, sensors, actuators, where each device has associated state.
- By having global variables within a package body, a package with "memory" can be created, *i.e.* a package with state.
- The procedures within the package can be used to change the state, and the state may in turn effect the behaviour of the subprograms within the package (finite state machines).

The `own` and `initializes` Clauses

- SPARK provides annotations in order to control the use of global variables.
- Within SPARK a variable that is global to a package is called an **own variable** and **must** be annotated as such.
- The initialization of own variables must also be included within the program annotations.
- Initialization may occur at the point of declaration, *e.g.*

```
Level: Integer:= 42;
```

  or via a statement sequence at the end of the package body or via a procedure call.

Packages with Memory

```
package Int_Stack
--# own Data, Pointer;
--# initializes Data, Pointer;
is
   function Full return Boolean;
   --# global in Pointer;

   procedure Push(X: in Integer);
   --# global in out Data, Pointer;
   --# derives Data from Data, Pointer, X &
   --#         Pointer from Pointer;

end Int_Stack;
```


Packages with Memory

```
package body Int_Stack
is
 Stack_Size: constant:=4;
 type Pointer_Range is range 0..Stack_Size;
 subtype Index_Range is Pointer_Range
                      range 1..Stack_Size;
 type Vector is array (Index_Range)
                               of Integer;
 Data: Vector;
 Pointer: Pointer_Range;

 function Full return Boolean is
 begin
    return Pointer = Stack_Size;
 end Full;

 procedure Push(X: in Integer) is
 begin
    Pointer:= Pointer + 1;
    Data(Pointer):= X;
 end Push;

begin
   Pointer:= 0;
   Data:= Vector'(Index_Range => 0);
end Int_Stack;
```

Array Initialization

- Simultaneous assignment via aggregate construct:

```
Data:= Vector'(Index_Range => 0);
```

- What is wrong with the following:

```
Data(1):= 0; Data(2):= 0; Data(3):= 0; Data(2):= 0;
```

- The Examiner's flow analysis treats an array as a single entity, therefore it requires all elements to be initialized simultaneously, *e.g.* when `Data(1)` is assigned the value `0`, the Examiner spots that `Data(2)` is undefined.
- For more details see:
  `http://www.macs.hw.ac.uk/~air/hic/notes/`
  `handoutArrayInit.pdf`

Abstract Own Variables

- Note that the annotations make visible significant implementation details, *e.g.* `Data` and `Pointer`.
- This problem is addressed by allowing an **abstract own variable** at the specification level to be linked with a sequence of **concrete own variables** within the associated body.
- Essentially the abstract package specification is refined within the concrete package body.
- Note that if a subprogram specification makes reference to an abstract own variable then the corresponding subprogram body must have refined versions of the `global` and `derives` clauses.

An Example Abstract Own Variable

```
package Int_Stack
--# own State;
--# initializes State;
is
   function Full return Boolean;
   --# global in State;

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

end Int_Stack;
```


Refining Abstract Own Variables

```
package body Int_Stack
--# own State is Data, Pointer;
is
 ...
 function Full return Boolean
 --# global in Pointer;
 is
 begin
   return Pointer = Stack_Size;
 end Full;

 procedure Push(X: in Integer)
 --# global in out Data, Pointer;
 --# derives Data from X, Data, Pointer &
 --#         Pointer from Pointer;
 is
 begin
    Pointer:= Pointer + 1;
    Data(Pointer):= X;
 end Push;

begin
   Pointer:= 0;
   Data:= Vector'(Index_Range => 0);
end Int_Stack;
```

Moded Own Variables

- Moded own variables **should** be used for external communication.
- Moded own variables are often referred to as **external variables**.
- Own variable modes:
  - **in**: input from an external source.
  - **out:** output to an external destination.

  Note that **in out** is not allowed.

Reading From An External Device (Example)

```
package P_Sensor
  --# own in Pressure;
  is
    procedure Read_P_Value(P: out Integer);
    --# global in Pressure;
    --# derives P from Pressure;
end P_Sensor;

package body P_Sensor is
  --# hide P_Sensor;
  --  Package body is not analysed by the SPARK Examiner
  procedure Read_P_Value(P: out Integer) is
  begin
     -- non-SPARK code that accesses pressure sensor
     -- hardware goes here.
  end Read_P_Value;
end P_Sensor;
```

The `hide` Clause

- Using the **hide** annotation, the following parts of a program can be hidden from the SPARK Examiner:
  - a subprogram body
  - the private part of a package (more later)
  - a package body (including initialization part)
- A **hide** clause must contain a subprogram/package name, *e.g.*

```
procedure Spooks
--# global out Secrets; in Spies;
--# derives Secrets from Spies;
is
  --# hide Spooks
  ... -- hidden text
end Spooks;
```

- Supports incremental development and analysis of code as well access to non-SPARK code, *e.g.* Ada, C, ...

Accessing Package Entities

- As illustrated earlier, the **with** clause declares the packages that the current program unit (procedure or package) requires access to, *i.e.*

```
with <name_1>, ..., <name_2>;
<current_program_unit>
```

- Note that SPARK forces all references to entities declared in other packages to be qualified with the package name, *e.g.*

```
with Int_Stack;
procedure Use_Int_Stack is
   ...
   Int_Stack.Push(42);
   ...
end Use_Int_Stack;
```

The `inherit` Clause

- As well as the `with` clause, SPARK forces the programmer to add an `inherit` clause.
- While the `with` clause relates to code visibility, the `inherit` clause relates to annotation visibility, *e.g.*

```
with Int_Stack;
--# inherit Int_Stack;
procedure Use_Int_Stack is
   ...
   Int_Stack.Push(42);
   ...
end Use_Int_Stack;
```

- But the `inherit` annotation does more than just duplicate the information provided by the `with` clause ...

More On The `inherit` Clause

```
package A is
   ...
end package A;

--#inherit A;
package B is
   ...
   ... A.V
   ...
end B;
```

```
package body A is
   V: Integer;
   --# inherit A;
   package B is
      ...
      ... A.V
      ...
   end B;
   ...
end A;
```

- Whether declared at the same level (above left) or nested (above right), the `inherit` clause must be used to indicate access to entities declared outside a package.
- Note that while the `with` clause may be attached to the package specification or the package body, the `inherit` clause is only attached to the package specification.

The main_program Annotation

- Within a SPARK program there will exist an unique main subprogram, where execution starts.
- The `main_program` annotation is used to indicate the main subprogram, *e.g.*

```
with Counter;
--# inherit Counter;
--# main_program;
procedure Use_Counter is
   ...
   Counter.Reset(42);
   ...
end Use_Counter;
```


Using The SPARK Examiner

- The SPARK Examiner analyses source files for conformance to the SPARK language rules.
- The SPARK Examiner requires certain basic information:
  - in order to analyse a package body the Examiner requires access to the package specification;
  - in order to access a package specification or main program that inherits auxiliary packages then the Examiner requires access to the specifications of the auxiliary packages.
- The SPARK Examiner provides feedback to the user via various output files.
- All source files are expected to be in the current working directory.

SPARK Command Line Interface

- The files to be analysed by the Examiner **must** be ordered on a need-to-know-basis, *e.g.*

  `spark int_stack.ads int_stack.adb test_int_stack.adb`

  where `int_stack.ads` and `int_stack.adb` are package specification and body respectively while `test_int_stack.adb` contains the main program (accesses `int_stack`).

- A **meta-file** (file extension `.smf`) can be used to simplify the command line (see next slide for an example).

Meta-file Example

- Given a file called `int_stack.smf` that contains:

  ```
  int_stack.ads
  int_stack.adb
  test_int_stack.adb
  ```

- Then the previous command line becomes:

  ```
  spark @int_stack.smf
  ```

The SPARK Examiner Output Files

- A report file is generated (default is `spark.rep`) that contains a list of the option settings used during the Examiner's analysis together with a summary of the analysis for each source file involved (see *SPARK Examiner User Manual*).
- Listing files detailing the analysis for each source file are also produced. By default the name of a listing file is generated by adding a `lst` extension to the given root file name, *e.g.* the listing file generated for `foo.adb` is by default `foo.lst`.
- Note that all report and listing files are written to the current working directory.

Some Argument & Command Options

- The argument option `listing_file` allows you to specify the name of a listing file, *e.g.*

```
spark foo.ads -listing_file=foo.1.lst ...
```

- The argument option `nolisting_file` allows you to suppress the generation of a listing file, *e.g.*

```
spark foo.ads foo.adb -nolisting_file
```

- The command option `listing_extension` allows you to specify the extension for a listing file, *e.g.*

```
spark -l=ls_ foo.ads foo.adb
```

  will generate `foo.lss` and `foo.lsb`, where `_` is a wild card character.

From Examination to Execution

- Ada compilation process:
  1. Sources files are **compiled** into object files, *e.g.* `foo.o` generated for `foo.adb`.
  2. Files are **bound** together, *i.e.* consistency checks, package elaboration, ... *etc*
  3. All relevant object files are **linked** to produce an executable file, *e.g.* `foo`.
- The GNAT Ada `gnatmake` utility:
  - Automates the above 3 steps.
  - Automatically determines which sources are required for compilation (if any).

Variables and Predefined Types

- SPARK predefined types are: `Integer`, `Float`, `Boolean`, `Character`, and `String`.
- Variables are declared by instantiating the following pattern:

```
<variable_seq> : <type> [:= <constant_expr>];
```

for example:

```
Count: Integer;
Found: Boolean:= False;
Weekly, Monthly: Float := 0;
```

Note that `:=` is assignment and is used here to initialize the variables `Found`, `Weekly` and `Monthly`.

Subtypes

- A **subtype** enables us to give a name to a subset of a type. Subtypes are useful for expressing values that lie within fixed bounds, *e.g.*

```
subtype Index_Type is Integer range 1 .. 10;
I, J, K: Index_Type;
```

- User defined types and subtypes cause problems when accessed out with their defining package. The problem is that:
  - SPARK excludes the Ada **use** clause – which makes everything within a package visible.
  - Accessing operators via the dot notation is messy, *e.g.* `P."+"` , `P."="` – **and SPARK syntax prohibits it!**
- To solve this problem SPARK includes the Ada `use type` clause that makes named types (subtypes) visible ...

A `use type` Example

```
package Foo is
   subtype Index_Type is Integer range 1 .. 10;
   ...
end package Foo;

with Foo; use type Foo.Index_Type;
--# inherit A;
--# main_program;
procedure Main
--# derives ... from ...;
is
X, Y, Z: Foo.Index_Type;
begin
   ...
   X:= Y+Z;
   ...
end Main;
```

Enumeration Type

- Enumeration types allow you to easily introduce meaningful names into your programs for collections of entities.
- For example, the days of the week can be represented by the enumeration type:

```
type Days is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
Today: Days;
...
Today:= Thu;
...
```

- The values within an enumeration type are ordered, *e.g.* `Mon` < `Tue` < `Wed` < `Thu` < ...

Constants & Attributes

- Constants are declared by instantiating the following pattern:

```
<variable_seq> : constant <type> := <constant_expr>;
```

  for example:

```
Maximum: constant Integer:= 100;
```

- Note that given a scalar (numeric & enumeration) subtype, *e.g.*

```
subtype Index_Type is Integer range 1 .. 10;
```

  then the first and last values (or attributes) can be accessed via `Index_Type'First` and `Index_Type'Last` respectively.

  Note that `Index_Type'First` is read as "index type tick first".

Arrays

- An array type is a random access composite type where the components are all of the same type (subtype).
- An example use of an array (constrained) type:

```
subtype Index is Integer range 1 .. 7;
type Readings is array (Index) of Float;
New, Old: Readings;
...
New(1) := 25.3;
New(2) := 22.3;
...
Old := New;
...
```

- Note that assignment, equality (`=`) and inequality (`/=`) are allowed with respect to arrays of the same type.

Records

- A record type is a composite type with named components.
- An example use of a record type:

```
subtype Days is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
type Reading is
   record
      Day: Days;
      Value: Float;
   end record;

A, B, C: Reading;
...
A.Day := Mon; A.Value := 32.2;
B.Day := Tue; B.Value := 23.4;
C := B;
```

- Note that assignment, equality (`=`) and inequality (`/=`) are allowed with respect to records of the same type.

Abstract Data Types via Packages

```
package Int_Stack is

   type Stack is private;

   function Full(S: Stack) return Boolean;

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

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

private
   ...

end Int_Stack;
```


Abstract Data Types via Packages

```
package Int_Stack is
   ...
private
   Stack_Size: constant:=4;
   type Pointer_Range is range 0..Stack_Size;
   subtype Index_Range is Pointer_Range
                            range 1..Stack_Size;
   type Vector is array (Index_Range) of Integer;
   type Stack is
      record
         Data: Vector;
         Pointer: Pointer_Range;
      end record;
end Int_Stack;
```

Note the private part is not accessible by the user, but must be provided within the specification so that the compiler can reserve appropriate storage.

Abstract Data Types via Packages

```
package body Int_Stack is

   function Full(S: Stack) return Boolean is
   begin
      return S.Pointer = Stack_Size;
   end Full;

   procedure Init(S: out Stack) is
   begin
      S.Pointer:=0;
   end Init;

   procedure Push(X: in Integer; S: in out Stack) is
   begin
      S.Pointer:=S.Pointer+1; S.Data(S.Pointer):=X;
   end Push;

end Int_Stack;
```


Conditional: `if` Statement

```
if X < Min then
   Min := X;
end if;
```

```
if X < Y then
   Min := X;
else
   Min := Y;
end if;
```

```
if X < Min then
   Pump := On;
   Value := Close;
elsif X > Max then
   Pump := Off;
   Value := Open;
elsif Pump = Off then
   Value := Close;
else
   Value := Open;
end if;
```

Note that the `elsif` form provides an alternative to the nesting of `if`-statements.

Conditional: case Statement

```
type Activity_Type is (Work, Gym, Party);
Activity: Activity_Type;
...
case Today is
   when Mon .. Thu => Activity := Work;
   when Fri => Activity := Gym;
   when Sat | Sun => Activity := Party;
end case;


Char: Character;
...
case Char is
   when 'a' .. 'z'| 'A' .. 'Z' =>
              Letter_Cnt := Letter_Cnt+1;
   when '0' .. '9' =>
              Digit_Cnt := Digit_Cnt+1;
   when others =>
              Unknown_Cnt := Unknown_Cnt+1;
end case;
```

Note that the `when others` clause is optional, but when present it comes last.

Iteration: loop and while statements

```
loop                              while <Condition> loop
   ...                               ...
   exit when <Condition>;            ...
   ...                               ...
end loop;                         end loop;
```

- Note that the `when` clause is optional, *i.e.* one can simply have an unconditional `exit`.
- If a `when` clause is present then it can **not** be nested within a conditional block, *i.e.* its closest block must be a loop.

Iteration: `for` statement

`for <Ident> in [reverse] <SubType> loop ... end loop;`

- Note that "`[...]`" denotes optional parts of the syntax.
- In the following `for`-loop the loop variable `I` counts up from 1 to `N`:

```
for I in Integer range 1 .. N loop
  ...
end loop;
```

- In the following `for`-loop the loop variable `I` counts down from `N` to 1:

```
for I in reverse Integer range 1 .. N loop
  ...
end loop;
```

Note that the loop variable (`<Ident>`) is local to the for-loop. Moreover, the variable **can not** also be declared within an outer block, *i.e.* eliminates risk of confusion that can arise with shadow variables.

Summary

- Packages with memory
- Own and initializes clauses
- Abstract own variables mechanism
- Moded own variables (external variables)
- With and inherits clauses
- Hide and main_program clauses
- Basic usage of SPARK Examiner & the GNAT Ada compiler
- Basic SPARK language features
- Abstract data types via packages