Rigorous Methods for Software Engineering (F21RS-F20RS)
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.
- An introduction to SPARK contracts.

Contracts

- A contract allows a programmer to specify the intended behaviour of a subprogram, i.e. *what* the subprogram is intended to achieve without saying *how* it will be achieved.
- The notion of a contract has been around along time, i.e.
  Assigning Meanings to Programs
  Robert W. Floyd
  Proceedings of Symposium on Applied Mathematics, 1969.
- Contracts appear in a number programming languages today, e.g. Eiffel, C# (and Spec#), Frama-C, ...

Subprogram Contracts

- Contracts in SPARK may involve:
  - **data dependencies:** specifies what global data is read and written.
  - **flow dependencies:** specifies what subprogram outputs depend upon on subprogram inputs.
  - **preconditions:** specifies logical constraints on the caller of a subprogram.
  - **postconditions:** specifies logical constraints on the functional behaviour of a subprogram.
  - **contract cases:** a complementary way of specifying pre- and postconditions.

Subprogram Contracts

- A contract should be written before coding begins.
- The SPARK verification tools automatically check the correctness of the code with respect to its contract, i.e. static analysis.
- A contract is represented using the notion of an *aspect* – a construct of Ada 2012 that allows properties of an entity to be represented.

Data Dependency Contracts

- A data dependency contract describes what global data a subprogram accesses, and how the data is used, i.e. in (read), out (written), in out (read & written).
- Access to global data can easily go unnoticed by a programmer, thus potentially leading to errors.
- SPARK addresses this issue by forcing a programmer to be explicit about the use of global data within a contract.
- The `Global` aspect is used to represent a data dependency contract, i.e.

```
Global => (...,
           <mode> => <global_variable>,
           ...);
```

  where `<mode>` can be `Input`, `Output` or `In_Out`.
- When a subprogram has neither global inputs nor global outputs, it can be specified using the null data dependencies, i.e. `Global => null;`

The Global Aspect – Output

```
package Counter
is
   Count : Integer:= 0;
   procedure Reset(Value: in Integer)
   with
     Global => (Output => Count);
...
end Counter;
package body Counter
is
   procedure Reset(Value: in Integer) is
   begin
      Count:=Value;
   end Reset;
...
end Counter;
```

Note that the contract is associated with the package specification.

The Global Aspect – In_Out

```
package Counter
is
   Count : Integer:= 0;
   procedure Inc
   with
     Global => (In_Out => Count);
...
end Counter;
package body Counter
is
...
   procedure Inc is
   begin
      Count:=Count+1;
   end Inc;
...
end Counter;
```

The Global Aspect – Input

```
package Counter
is
   Count : Integer:= 0;
   function Value return Integer
   with
     Global => (Input => Count);
end Counter;
package body Counter
is
...
   function Value return Integer is
   begin
      return Count;
   end Value;

end Counter;
```

The `Global` Aspect – Summary of Modes

- Global mode `Input` indicates that the global variable should be completely assigned before calling the subprogram.
- Global mode `Output` indicates that the global variable should be completely assigned by the subprogram (procedure) call.
- Global mode `In_Out` indicates that the global variable should be completely assigned before calling the subprogram (procedure). And it can be assigned by the subprogram (procedure) call.

Note that if the global state (see `Abstract` and `Refined` Aspects below) is complex (i.e. multiple constitute parts) and an assignment is partial (i.e. not all constitute parts are assigned), then global mode `In_Out` should be used.

Flow Dependency Contracts (Procedures)

- Flow analysis is concerned with the relationships between values and computations.
- Variables that are used to communicate values into and out of procedures, i.e. imported and exported values.
- A flow dependency contract can be used to describe the relationship between imported and exported values.
- The `Depends` aspect is used to represent a flow dependency contract, i.e.

```
Depends => (...,
            <variable> => (..., <variable>, ...),
            ...);
```

When an output value does not depend on an input value corresponding flow dependency should use the null input list, i.e.

```
Depends => (...,
            <variable> => null,
            ...);
```

The Depends Aspect

```
package Switch
is
   procedure Int_Switch(X, Y: in out Integer)
   with
     Depends  => (X => Y, Y => X);
end Switch;
package body Switch
is
   procedure Int_Switch(X, Y: in out Integer)
   is
      T: Integer;
   begin
      T:=X; X:=Y; Y:=T;
   end Int_Switch;
end Switch;
```

Note: the exported value of X is derived from the imported value of Y and the exported value of Y is derived from the imported value of X.

Flow Dependency Contracts (Functions)

- A flow dependency contract can also be used to describe the relationship between the inputs to a function and the resulting value of the function.
- The general form of the `Depends` aspect for a functional contract is as follows:

```
Depends => (...,
            <func-id>'Result => (..., <variable>, ...),
            ...);
```

The Depends Aspect

```
package Inc_Value
is
   type T is range -128 .. 128;

   function Inc(X: T) return T
     with
     Depends => (Inc'Result => X);

end Inc_Value;
package body Inc_Value
is
   function Inc(X: T) return T
   is
   begin
      Return X+1;
   end Inc;
end Inc_Value;
```

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).

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.

Packages with Memory

```
package Int_Stack
is
   pragma Elaborate_Body;

   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;
   ...
end Int_Stack;
```

The order in which packages are **elaborated** can effect the meaning of an Ada program. The use of the **pragma** `Elaborate_Body` eliminates this potential for ambiguity and is enforced by SPARK when a declaration and the associated initialization are split across package specification and body.

Packages with Memory

```
package Int_Stack

is
   ...
   function Full return Boolean
     with
       Global  => (Input => Pointer),
       Depends => (Full'Result => Pointer);

   procedure Push(X: in Integer)
     with
       Global  => (In_Out  => (Data, Pointer)),
       Depends => (Data    => (Data, Pointer, X),
                   Pointer =>  Pointer);

end Int_Stack;
```

Packages with Memory

```
package body Int_Stack
is
   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(4):= 0;
```

- The flow analysis embodied within the SPARK tools 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 flow analysis spots that `Data(2)` is undefined.

Abstract Package Variables

- Note that the contract makes visible significant implementation details, *e.g.* `Data` and `Pointer`.
- This problem is addressed by allowing an **abstract package variable** at the specification level to be linked with a sequence of **concrete package 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 package variable then the corresponding subprogram body must have refined versions of the `Global` and `Depends` aspects.

An Example Abstract Package Variable – Abstract_State

```
package Int_Stack
  with
  Abstract_State => State
is
  function Full return Boolean
    with
    Global => (Input => State),
    Depends => (Full'Result => State);

  procedure Push(X: in Integer)
    with
      Global  => (In_Out => State),
      Depends => (State => (X, State));

end Int_Stack;
```

Note that the name of the abstract state here is `State`, but any legal variable identifier is allowable.

An Example Abstract Package Variable – Abstract_State

```
package Int_Stack
  with
  Abstract_State => State
is
  function Full return Boolean
    with
    Global => (Input => State),
    Depends => (Full'Result => State);

  procedure Push(X: in Integer)
    with
      Global  => (In_Out => State),
      Depends => (State => (X, State));

end Int_Stack;
```

Note also that the State depends on X and State. This is because the exported State corresponds to the imported State with the addition of X.

Refining Abstract Package Variables – Refined_State

```
package body Int_Stack
  with
  Refined_State => (State => (Data, Pointer))
is
   ...
   function Full return Boolean
     with
     Refined_Global => (Input => Pointer),
     Refined_Depends => (Full'Result => Pointer)
   is
   begin
      return Pointer = Stack_Size;
   end Full;
   ...
end Int_Stack;
```

Whenever an `Abstract_State` is used (`.ads`), an associated `Refined_State` is required (`.adb`).

Refining Abstract Package Variables – Refined_State

```
package body Int_Stack
  with
  Refined_State => (State => (Data, Pointer))
is
   ...
   procedure Push(X: in Integer)
     with
     Refined_Global => (In_Out => (Data, Pointer)),
     Refined_Depends => (Data => (X, Data, Pointer),
                         Pointer => Pointer)
   is
   begin
      Pointer:= Pointer + 1;
      Data(Pointer):= X;
   end Push;
   ...
end Int_Stack;
```

SPARK Tools – Checks on Abstract/Concrete State

- Each **Abstract Global Input** has at least one of its constituents mentioned by the **Concrete Global Inputs** (refined).
- Each **Abstract Global In_Out** has at least one of its constituents mentioned with mode Input and one with mode Output (or at least one constituent with mode In_Out) at the concrete (refined) level.
- Each **Abstract Global Output** has to have **ALL** its constituents mentioned by the **Concrete Global Outputs** (refined).

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` $<$ ...

Records

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

```
type 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
     with
     Depends => (Full'Result => S);

   procedure Init(S: out Stack)
     with
     Depends => (S => null);

   procedure Push(X: in Integer; S: in out Stack)
     with
     Depends => (S => (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;
      S.Data:=Vector'(Index_Range => 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' => Let_Cnt := Let_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`.
- Note that you can have multiple `exit` points.

Iteration: `for` statement

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

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

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

- In the following 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 – the variable **can not** also be declared in an outer block, *i.e.* eliminates risk of confusion that can arise with shadow variables.

Summary

- Contracts involving Global and Depends.
- Packages with memory.
- Abstract and concrete package variables.
- Record and enumeration types.
- Abstract data types via packages.
- Flow of control constructs.