Rigorous Methods for Software Engineering (F21RS-F20RS)
The SPARK Approach: Part 1

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

Overview

- Basic building blocks of a SPARK program.
- Basic analysis: checking compliance with the SPARK subset of Ada.

Basic Program Unit

- The basic building block of a program is the **procedure**.
- A procedure consists of a **declaration** part and a **statement** part:

```
procedure <program_name> (<parameter_list>) is
   <declarations>
begin
   <statements>
end <program_name>;
```

Note the use of `<...>` to denote a place-holder for actual program code.

Hello World – Revisited

```
with Text_IO;
-- My first program (this is a comment)!
procedure Hello is
begin
   Text_IO.Put_Line("Hello WORLD!");
end Hello;
```

- Note that an empty parameter list **does not** contain brackets, *i.e.* `()`, as is the case with Java, C, C++.
- A program will typically use a existing library resource, *e.g.* such as the `Put_Line` procedure from the `Text_IO` library resource (more details later).
- Such library resources will typically contain many procedures, types, *etc* and therefore require a higher level of structuring ...

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 := 0.0;
```

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

Integer Types and Subtypes

- Specifying the valid range of an `Integer` type can help to eliminate certain kinds of errors and makes it easier to reason about the correctness of the code, e.g.

```
type A_grade is range 70 .. 100;
type B_grade is range 60 .. 69;
```

- However, combining variables of different types in an expression requires explicit type conversion.
- A neater approach is to use a **subtype**, *e.g.*

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

  The **base type** of the `subtype` `Index_Type` is `Integer`.
- Variables that have the same base type can occur within an expression and without the need for type conversion.

Constants

- Constants are declared by instantiating the following pattern:

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

  for example:

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

- There is an alternative form of the constant declaration:

  <variable_seq> : constant := <static_expr>;

  A <static_expr> is limited to expressions of a scalar type and String type.

- Note that the type of <static_expr> is implicit.

Constants

- The implict typing of a `<static_expr>` is useful when defining a type. For example, consider the type declarations:

```
Stack_Size: constant := 4;
type Pointer_Range is range 0..Stack_Size;
subtype Index_Range is
        Pointer_Range range 1..Stack_Size;
```

The above would not be legal SPARK if `Stack_Size` is declared as follows:

```
Stack_Size: constant Integer := 4;
```

That is, if `Stack_Size` is declared to be an `Integer` then it cannot be used in the declaration of `Index_Range`. This is because `Index_Range` is a subtype of `Pointer_Range`, and the type `Pointer_Range` is distinct from the type `Integer`.

Attributes

- 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".

Formal Parameters To Procedures

- in : the value of the **actual** parameter is copied to the **formal** parameter. The **formal** parameter is treated as a constant. Note if not specified then **in** is the default mode.
- in out : the **actual** parameter must be a variable that has a value at the time the procedure is called. The value can be used or changed within the procedure. At the end of the procedure call the value of the **formal** parameter is copied back to the **actual** parameter.
- out : the **actual** parameter must be a variable. The value of the **actual** parameter at the time the procedure is called is ignored. At the end of the procedure call the value of the **formal** parameter is copied back to the **actual** parameter.

A Simple Procedure

```
procedure Int_Switch(X, Y: in out Integer)
is
   T: Integer;
begin
   T:=X;
   X:=Y;
   Y:=T;
end Int_Switch;
```

- `X` and `Y` are (formal) parameters of type `Integer`.
- `T` is a local variable of type `Integer`.
- The effective of the call `Int_Switch(A, B)` is to switch the values of the actual parameters `A` and `B`.

A Simple Function

```
function Int_Min(X, Y: in Integer) return Integer
is
begin
   if X > Y then
      return(Y);
   else
      return(X);
   end if;
end Int_Min;
```

- SPARK functions **can not** have side-effects:
  - function parameters are constrained to have **in** mode
  - modification of global state variables is prohibited (more details later).
- Note that a SPARK function can have multiple `return` statements. This represents a change from SPARK 2005 which allowed only a single return statement.

Building Larger Program Units

- This higher level of structuring, or abstraction, is provided by **packages**.
- A large number of so called library packages, such as `Text_IO` exist, however, the notion of a package provides a general mechanism for structuring large software systems.
- While procedures support programming-in-the-small, packages can be seen as supporting programming-in-the-large.
- A package has a **specification** and a **body**:
  - A package specification provides an interface, it tells a user **what** resources a package provides.
  - A package body defines **how** the resources are implemented, and is not visible to the user.

Package Specification

```
package <package_name> is
   <declarations>
end <package_name>;
```

- Here the declarations provide enough information to use a resource (subprogram) without revealing its implementation details.
- Note that package specifications and bodies can be defined within the same file, but "best practice" suggests keeping them in separate files, *i.e.* `foo.ads` for the package specification of `foo` and `foo.adb` for the package body of `foo`.
- Note also that the file names need to be in **lower case**.

Package Body

```
package body <package_name> is
   <declarations>
end <package_name>;
```

- Here the declarations provide the implementation details of the resource (subprogram).
- Any constants, types, variables declared within the package body are not accessible from outside the package.

Package Body

- An optional statement part can be included within the package body, *i.e.*

```
package body <package_name> is
   <declarations>
begin
   <statements>
end <package_name>;
```

- If included, then the statement part is executed once when the program using the package starts. This process is called **elaboration**.

Different Roles for Packages

- Packages can be used to group together types and constants.
- Packages can be used to group together logically related subprograms (procedures and functions).
- Packages can have “memory” and can therefore represent objects with state.
- Packages can be used to construct **abstract data types**.

Packages of Types and Constants

```
package Distances is
    subtype Dist is Integer range 1..Integer'Last;

    Edin_Glas: constant Dist := 42;
    Glas_Stir: constant Dist := 30;
    Stir_Edin: constant Dist := 36;
end Distances;
```

Packages of Subprograms

```
package Volumes is

   Pi: constant Float := 3.14159;

   function Box_Car_Vol(L: Float;
                        W: Float;
                        H :Float) return Float;

   function Tank_Car_Vol(L: Float;
                         R: Float) return Float;

end Volumes;
```

Packages of Subprograms

```
package body Volumes is

   function Box_Car_Vol(L: Float;
                        W: Float;
                        H :Float) return Float is
   begin
      return L * W * H;
   end Box_Car_Vol;

   function Tank_Car_Vol(L: Float;
                         R: Float) return Float is
   begin
      return Pi * R * R * L;
   end Tank_Car_Vol;

end Volumes;
```

A Simple Procedure (specification)

```
package Switch
is
   procedure Int_Switch(X, Y: in out Integer);

end Switch;
```

- `X` and `Y` are (formal) parameters of type `Integer`.
- The intended effective of the call `Int_Switch(A, B)` is to switch the values of the actual parameters `A` and `B`.

A Simple Procedure (body)

```
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;
```

- `X` and `Y` are (formal) parameters of type `Integer`.
- `T` is a local variable of type `Integer`.
- The effective of the call `Int_Switch(A, B)` is to switch the values of the actual parameters `A` and `B`.

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 Switch;
package body Foo is
  procedure Use_Int_Switch is
    ...
    Switch.Int_Switch(A, B);
    ...
  end Use_Int_Switch;
end Foo;
```

The `use type` clause

- When we declare a type, a set of operations is also defined.
- For example, consider the declaration of `Index_Type` within a package A:

  ```
  type Index_Type is range 1 .. 10;
  ```

  `Index_Type` is associated with a distinct set of arithmetic operators, e.g. $+$, $-$, $=$, $>$ etc.
- When accessing a type from another package we must:
  - Prefix the operators with the name of the package in which they are defined, OR
  - Include a **use type clause** which gives direct access to the operators associated with the specified type, e.g. `use type A.Index_Type`

A **use clause** is also available, which gives access to all resources within a package.

A use type Example

```
package A is
   type Index_Type is range 1 .. 10;
   ...
end A;

with A;
use type A.Index_Type;
package B
is
   procedure Inc (X: in out A.Index_Type)
   ...
end B;
```

The code for this example is available via
http://www.macs.hw.ac.uk/~air/rmse/code/AB/

Identifying SPARK Code

- A software system maybe constructed using a combination of SPARK and full blown Ada 2012, i.e. where the safety and security critical code is written in SPARK.
- One can specify the SPARK (and non-SPARK) parts using the Ada `pragma` construct, i.e.

```
pragma SPARK_Mode (On)
package Switch
is
   procedure Int_Switch(X, Y: in out Integer);

end Switch;
```

Note that a `pragma` denotes a directive that is used here by the SPARK verification tools. We will encounter other uses of the Ada `pragma` construct in time.

Identifying SPARK Code

- Note that pragma `SPARK_Mode (Off)` switches off the SPARK analysis mode.
- Alternatively, one can use the *aspect* construct to allow for finer control over the SPARK mode, i.e.

```
package Switch
is
   procedure Int_Switch(X, Y: in out Integer)
   with
     SPARK_Mode => On;

end Switch;
```

Verifying Software Written in SPARK

- The SPARK tools can be run either via the **command-line** or using the **GNAT Programming Studio** (GPS). For the purposes of these notes we will focus on the command-line interface.
- GNATprove is the SPARK Verification Tool.
- GNATprove supports 3 level of analysis:
  - Language compliance – checks that the correct subset of Ada has been used.
  - Flow analysis – checks data flow (e.g. initialization of variables) and information flow (e.g. consistency of code with dependency relations).
  - Formal proof – formally verifies code with respect to the: i) absence of run-time errors and ii) assertions (e.g. pre- and postconditions).

Running GNATprove at the Command Line

```
gnatprove -P myproject.gpr --mode=check -u switch.ads switch.adb
```

- `-P` flag specifies the project file, e.g. `myproject.gpr`,
- `-u` flag specifies specification files to analyse.
- `-U` flag indicates that all files in the search path (i.e. defined within the project file) should be analysed.
- Specifying level of analysis:
  - `--mode=check` indicates that code should be verified with respect to the SPARK subet of Ada.
  - `--mode=flow` indicates that code should be verified with respect flow contracts.
  - `--mode=prove` indicates that code should be verified with respect to the i) absence of run-time errors and ii) assertions (e.g. pre- and postconditions).

GNATprove: Summary of SPARK Analysis

```
Summary of SPARK analysis
=========================

-------------------------------------------------------------------------------------------------------------
SPARK Analysis results        Total        Flow   Interval   CodePeer   Provers   Justified   Unproved
-------------------------------------------------------------------------------------------------------------
Data Dependencies                 1           1          .          .         .           .          .
Flow Dependencies                 1           1          .          .         .           .          .
Initialization                    1           1          .          .         .           .          .
Non-Aliasing                      .           .          .          .         .           .          .
Run-time Checks                   .           .          .          .         .           .          .
Assertions                        .           .          .          .         .           .          .
Functional Contracts              .           .          .          .         .           .          .
LSP Verification                  .           .          .          .         .           .          .
-------------------------------------------------------------------------------------------------------------
Total                             3    3 (100%)          .          .         .           .          .


max steps used for successful proof: 0

Analyzed 1 unit
in unit switch, 2 subprograms and packages out of 2 analyzed
  Switch at switch.ads:4 flow analyzed (0 errors, 0 checks and 0 warnings)
  Switch.Int_Switch at switch.ads:7 flow analyzed (0 errors, 0 checks and 0 warnings)
```

- For **flow** and **prove** modes a summary table is generated, i.e. `./gnatprove/gnatprove.out`
- The above summary table was generated for the switch example (i.e. `mode=flow`).
- Note: flow analysis is the focus of the next lecture.

Summary

Learning outcomes:

- Understand the basic building blocks of a SPARK program.
- Understand programming in the large via packages.
- Understand how to check SPARK compliance, i.e. the SPARK subset of Ada.

Summary

Recommended reading:

- "Building High Integrity Applications with SPARK" Mccormick, J.W. and Chapin, P.C. Cambridge University Press, 2015.
- "High Integrity Software: The SPARK Approach to Safety and Security" Barnes, J. Addison-Wesley, 2003.
- AdaCore SPARK resources:
  - SPARK 2014 User's Guide: https://docs.adacore.com/spark2014-docs/html/ug/
  - AdaCore: SPARK Pro: https://www.adacore.com/sparkpro
- Related approaches to high integrity software engineering:
  - Frama-C: https://frama-c.com/index.html
  - eCv: http://eschertech.com/products/ecv.php
  - B-Method http://www.systerel.fr
  - Eiffel: https://www.eiffel.com
  - Spec#: https://www.microsoft.com/en-us/research/project/spec/