CSC313 High Integrity Systems/CSCM13 Critical Systems

CSC313 High Integrity Systems/ CSCM13 Critical Systems

Course Notes
Chapter 2: SPARK Ada
Section (a)

Anton Setzer
Dept. of Computer Science, Swansea University

http://www.cs.swan.ac.uk/∼csetzer/lectures/critsys/current/index.html

February 22, 2019

Remark

Sections 2 (a) - (g) will be highly based on John Barnes: High Integrity Software. The SPARK approach. [Ba03].

2 (a) Introduction into Ada

2 (b) Architecture of SPARK Ada

2 (c) Language Restrictions in SPARK Ada

2 (d) Data Flow Analysis

2 (e) Information Flow Analysis

2 (f) Verification Conditions (Simple Programs)

2 (g) Verification Cond. (Range, Loops, Procedures, Functions)

2 (h) Example: A railway interlocking system

2 (a) Introduction into Ada

2 (b) Architecture of SPARK Ada

2 (c) Language Restrictions in SPARK Ada

2 (d) Data Flow Analysis

2 (e) Information Flow Analysis

2 (f) Verification Conditions (Simple Programs)

2 (g) Verification Cond. (Range, Loops, Procedures, Functions)

2 (h) Example: A railway interlocking system

Ada Lovelace (1815 – 1852)

- Ada Lovelace wrote programs for Charles' Babbage analytical machine.
  - The first general purpose computer (mechanical; Turing complete).
- Therefore often regarded as the first computer programmer.

Ada Lovelace (1815 – 1852)

A Basic Introduction into Ada

- In this subsection we introduce the basic features from Ada used in our introduction into SPARK Ada.
- This is not a thorough introduction into Ada.
- It is not necessary to have a complete understanding of Ada for this module.
  You only need to be able to understand the SPARK Ada code used.
- There is no need to write full Ada programs.

Motivation for Developing Ada

- Original problem of Department of Defence in USA (DOD):
  - Too many languages used and created for military applications (>450).
    - Languages largely incompatible and not portable.
    - Often minimal software available.
    - Competition restricted, since often only one vendor of compilers and other tools for one language.
  - Existing languages too primitive.
    - No modularity.
    - Hard to reuse.
  - Problems particularly severe in the area of embedded systems.
    - 56% of the software cost of DOD in 1973 for embedded systems.
  - Most money spent on maintaining software, not developing it.

Ada

- Decision by DOD: Development of new standard programming language for military applications.
- First release: Ada 83
  (1983 – same year C++ was released).
- Ada 95: Revision of Ada, integration of object-orientation.
- Ada 2012: Revision of Ada, including many features of SPARK Ada into the core language.
  - Allows therefore what in Eiffel is called **Design by Contract (TM)**.

Imperative Programming Language

- Ada can be seen as the culmination of the development of imperative programming languages.
- It was developed at a time when object oriented languages started to be used widely.
- Syntax of object orientation added to Ada was not as easy to use as in other object-oriented languages.

Reasons for Using Ada in Critical Systems

- Ada is the often **required standard for military applications in the US**, and has therefore adopted by the Western military industry.
  - Software for **military applications forms a large portion of critical systems**.
  - Therefore lots of **tools for critical systems support Ada**.
- Ada was developed taking into account its use in **real-time critical systems:**
  - It is possible to write efficient code and code close to assembler code, while still using high level abstractions.
  - Features were built in which prevent errors (e.g. array bound checks, no writing to arbitrary memory locations).

Basic Imperative Features

Ada is an imperative language.

- There are global variables.
- Instead of methods we have functions and procedures.
  - procedures are like methods in Java with return type void (i.e. no return type),
  - functions are like methods in Java with a genuine return type (not void).

Not Case Sensitive

Ada is **not** case sensitive.

- So x and X are the same,
- HelloWorld, Helloworld and helloworld are the same.

SPARKAda will replace in generated code all variables by their lower case versions.

Compilation

- One can compile ada programs using the gps system, which will be discussed later.
- One can compile Ada as well on the command line using the gcc-ada compiler (available under Linux).
  The command is

  ```
  gnatmake file
  ```

  This will automatically compile the file and all packages referred to in the file.

Division into Specification and Implementation

All units in Ada (programs, functions, packages) have two parts

- The specification.
  - It declares the interface visible to the outside.
  - E.g. for a function the specification declares its arguments and return type:

    ```
    function Myfunction (A : Integer) return Integer;
    ```
- The implementation.
  - E.g. for a function it would be

    ```
    function Myfunction (A : Integer) return Integer is
    begin
       return A ** 3;
    end Myfunction;
    ```
  - ** means here exponentiation.
  - Note that that the implementation repeats the specification.

(**sect2a/example2a-1/**)

Examples

- References such as **sect2a/example2a-1/** in the slides refer to the examples.
  These can be found at the git repository

  https://bitbucket.org/anton_setzer/
  criticalhighintegritysystems/src/master/

  which you can clone by running from the command shell under Linux (all in one line omit the line break)

  git clone https://anton_setzer@bitbucket.org/anton_setzer/
  criticalhighintegritysystems.git

Division into Specification and Implementation

- Usually specification and implementation are separated into two files:
  - **file.ads** contains the specification.
  - **file.adb** contains the implementation.

Basic Syntax

- **Typing** is written as in Haskell
  - X : A stands for "X has type A" .
    - In Java or C this is written as "A X".
- -- introduces a comment.

Enumeration Types

- We have enumeration types. E.g.
  - **type** Genders **is** (Male,Female);

  introduces a new type having elements Male and Female.
  (**sect2a/example2a-2**)

Ranges in Ada

- Ada allows to define ranges.

  subtype Year **is** Integer **range** 0 .. 2100;

  Myyear : Year := 3010;
  results in a compiler error.
  - If an out of range error is detected at compile time, the compiler will report an error.
  - If it isn't detected at compile time we get a run time error:

    Get(Myyear);
    If you enter at runtime 2101 we get:

    raised CONSTRAINT_ERROR : example5.adb:15 range check failed

  (**sect2a/example2a-3/**)

Ranges in Ada

- We have as well subranges of enumerations:

  type Day **is** (Mon,Tue,Wed,Thu,Fri,Sat,Sun);

  subtype Weekday **is** Day **range** Mon .. Fri;
  (**sect2a/example2a-3/**)

Arrays

Arrays are declared and used as follows:

type myArrayRange **is new** Integer **range** 0 .. 100;

type Day **is** (Mon, Tue, Wed, Thu, Fri, Sat, Sun);

type myArrayType **is array** (myArrayRange) **of** Day;

myArray : myArrayType;
myArray(17) := Sun;

One can have as well anonymous arrays (index type is not defined separately):

type Vector **is array** (0 .. 100) **of** Integer;

(**sect2a/example2a-4/**)

Conversion between Types

- To convert from one type to another one uses the notation Mytype(X), which converts a variable X of some type into Mytype.
- You can convert from a type to a subtype and back.
- For instance, if
  X : Integer
  then
  Year(X) : Year
  and if
  Y : Year
  then
  Integer(Y) : Integer
- Conversion to a subtype is generally unsafe.
  - SPARK Ada will generate verification conditions for that.

(**sect2a/example2a-4b/**)

If Clauses

- The syntax for if and nested ifs is as follows:

```
if Answer = 'y' then
   Put_Line ("You said yes");
elsif Answer = 'n' then
   Put_Line ("You said no");
else
   Put_Line ("You must type y or n");
end if;
```

(**sect2a/example2a-5/**)

Loops

- The basic loop statement is an infinite loop with an exit statement. E.g.
  -
```
Answer : Character;
loop
   Put("Answer yes (y) or no (n)");
   Get(Answer);
   if Answer = 'y' then
     Put_Line ("You said yes");
     exit;
   elsif Answer = 'n' then
     Put_Line ("You said no");
     exit;
   else
     Put_Line ("You must type y or n");
   end if;
end loop;
```

(**sect2a/example2a-6/**)

For and While Loops

- "For loops" are written as follows:
  -
```
for N in 1 .. 20 loop
   Put ("-");
end loop;
```
- A while loop is written as follows:
  -
```
while (X > 0.0) loop
   Y := Y + 1.0;
   exit when Y > 5.0;
   X := X - 1.0;
end loop;
```

(**sect2a/example2a-7/**)

Nested Loops can be Labelled

- One can label loops.
  This allows to exit not only to the currently innermost loop.
  Example:

```
Outer_Loop:
for J in 1 .. M loop
  ...
  for K in 1 .. M loop
      <SomeCode>
      exit Outer_loop when A = 0;
      <SomeCode>
      end loop;
    ...
end loop Outer_Loop;
-- when exit statement is executed,
-- we continue here
```

(**sect2a/example2a-8/**)

In - out - parameters

- In Ada all parameters have to be labelled as
  - input parameters; symbol: **in**,
    - Value parameters.
      The same as a **non-var** parameter in Pascal.
      All parameters in Java are **in** parameters.
  - output parameters; symbol: **out**,
  - input/output parameters; symbol: **in out**.
    - Reference parameters.
      The same as a **var** parameter in Pascal
      In Java non-constant instance variables of object parameters behave essentially as **in out** parameters.
- Example:

```
procedure ABC(A: in Float;
              B: out Integer;
              C: in out Colour)
```

(**sect2a/example2a-9/**)

In – Out – In Out Parameters

- **out** and **in out** parameters can only by instantiated by variables.
  - Assume for instance a procedure

```
procedure Myproc(B: out Integer)
  is begin
  B:= 0;
  end Myproc;
```

  - It doesn't make sense to make a call
    Myproc(0)
    (Compiler error)
    it only makes sense to call
    Myproc(C)
    where C is a variable that can be changed.
- We see as well that the variable we instantiate it with cannot be an **in** parameter itself, because then it cannot be changed.
  (**sect2a/example2a-10/**)

In – Out – In Out Parameters

- **in** parameters can be instantiated by arbitrary terms.
  - Consider:

```
procedure Myproc(B: in Integer,C: out Integer)
  is begin
  C:= B;
  end Myproc;
```

  - It makes sense to call
    Myproc(0,D)
    where D is a variable that can be changed.
  - It makes as well sense to call
    Myproc(3 + 5,D)
    or
    Myproc(f(E,F),D)
    where f is a function with result type Integer.

(**sect2a/example2a-11/**)

In – Out – In Out Parameters

- In **Java**
  - Parameters are always passed by **value**, and behave like **in** parameters.
  - The value of an object which is passed as a parameter is the **pointer** to that object.
    - A method will use a **copy** of that pointer to that object.
      Changing the pointer will have no effect outside of the method.
    - However, we can change the **instance variables** of an object (possibly by invoking methods of the object)
      and such changes are visible to the outside.
    - So instance variables in objects are passed by **reference**.
  - This is different from Ada where, if a record is passed on as an **in** parameter, no fields can be changed.

(**sect2a/example2a-12-java/, sect2a/example2a-13/**)

Example Parameters in Java

```
public static void exchange(int a, int b){
   int tmp = a;
   a = b;
   b = tmp;}
-- The above doesn't exchange a and b
-- since integers are passed by value
-- so the method works only on a copy of those integers

-- Define a wrapper class for integers
class myint{
   public int theint ;
   public myint(int theint){
      this.theint = theint;}
}
```


Example Parameters in Java

```
public static void exchange(myint a, myint b){
   myint tmp = a;
   a = b;
   b = tmp;}
-- The above doesn't exchange a and b
-- since the references to the objects are passed by value
-- so the method works only on a copy of those references

public static void exchange2(myint a, myint b){
   int tmp = a.theint;
   a.theint = b.theint;
   b.theint = tmp;}
-- The above does exchange a and b
```


Packages

- Programs are divided into packages.
- A package is a collection of types, global variables (with their types), functions and procedures.
- Packages are specified (in the .ads file) as follows:

```
package Mypackage is
-- Specification of the types, functions and procedures
-- of this package
end Mypackage;
```


Packages

- The implementation (in the .adb file) of packages is given as follows:

```
package body Mypackage is
-- Implementation of all types functions and procedures
-- of this package
begin
   -- Initialisation part of the packages
   -- Initialises global variables
end Mypackage;
```

Types Defined in Packages

- Types defined in different packages are different.
  - If you want to use the same type in several packages you need to import it from a common package.
  - For instance, you could have a package only containing definitions of types for that purpose.

Initialisation of Variables

Variables in a package can be initialised

- In their declaration e.g.

  Sum: Integer := 0;

  declares a new variable Sum of type Integer which is initialised to 0.

Initialisation of Variables

Variables in a package can be initialised

- Or in the initialisation part of the procedure, e.g.

```
package body MyPackage is
X : Integer range 0 .. 1000;

-- Declaration of procedures

begin
   X := 0;
end MyPackage
```

(**sect2a/example2a-14/**)

Using other Packages

If one wants to refer to another package, one has to write before the procedure/package:

with nameOfPackageToBeUsed;

Then no identifiers are directly visible, they have to be used in the form

nameOfPackageToBeUsed.X

If one adds in addition

use nameOfPackageToBeUsed;

then the identifier X can be used directly.

Record Types

- A record is essentially a class without methods.
- Syntax in Ada:

```
type Person is
  record
    Yearofbirth : Integer;
    Gender : Genders;
  end record
```

- This example declares a type Person which has two fields Yearofbirth and Gender.
- If a: Person, then we have that a.Yearofbirth : Integer and a.Gender : Genders.
- One can make assignments like a.Gender := Male.
  (**sect2a/example2a-14/**)

Record Types

- If a : Person one can give a value to its fields by saying
  - a := (1975,Male);
  - or
    a := (Yearofbirth => 1975,Gender => Male);
- There are as well Variant Records, where the type of other fields depends on one field. See additional material.

Polymorphism

- Ada allows to introduce a new name for a type.
  - Then functions for the new type inherit the functions from the old type.
  - However functions can be **overridden**.

Example for Use of Polymorphism

```
type Integer is <SomeCode>; -- Some definition.

function "+" (X, Y : Integer) return Integer;
-- + can be used infix

type Length is new Integer;
-- We can use a + b : Length for a,b : Length

type Angle is new Integer;
function "+" (X, Y : Angle) return Angle;
-- Now we have overridden + for Angle by a new function.
```

(Example taken from S. Barbey, M. Kempe, and A. Strohmeier: Object-Oriented Programming with Ada 9X. `http://www.adahome.com/9X/OOP-Ada9X.html`)

Polymorphism - Deciding Which Function is Used

- Note in case of polymorphism, which function to be chosen, can be decided at compile time, since it only depends on the (fixed) type of the argument.

Executable File

- In order to create an exectuable file one needs to have a file "filename.adb" with one procedure having name "filename".
- One can compile it
  - From the command line by using gnatmake filename.adb
  - From gps by clicking on the current file and running Build → Project → Build <current file>
- This will create an executable file with "filename" which can be executed from a shell.
- For pure Ada files one can use from libraries Ada.Text_Io and Ada.Integer_Text_IO
  - "Put" for printing a string or integer on the console
  - "Get" for getting as input a string or integer on the console.
  - "New_Line" for a printing a line break.

Example Executable Program: example2.adb

- The following example program will demontrate how to use IO.
- It will raise if one inputs something which is not a number on the console an error, because Get(Number) requires that the input is a number.
- Later, when using SPARK Ada we will use a wrapper class, which avoids this problem.

Example Executable Program: example2.adb

```
with Ada.Text_Io, Ada.Integer_Text_IO, Myfunction;
use Ada.Text_Io, Ada.Integer_Text_IO;

procedure example2 is
   Number : Integer;
begin
   Get(Number);
   Put( "Test " );
   Put("f(x)=");
   Put(Myfunction(Number));
   New_Line(1);
end Example2
```

Example Executable Program: example2.ads

```
procedure Example2;
```


Example Executable Program: myfunction.adb

```
function Myfunction (A : Integer) return Integer is
begin
  return A ** 3;
end Myfunction;
```


Example Executable Program: myfunction.ads

```
function Myfunction (A : Integer) return Integer;
```

(**sect2a/example2a-1/**)

Input of Strings from Console

- In order to obtain a user input from the console, you need to declare a variable of type String.
  - Ada requires to determine the maximum length of the strings.
  - This is done by writing (taking as length 20):
    S : String(1 .. 20);
  - You can initialize a string to have only blanks by using the following:
    S : String(1 .. 20):= (others => ' ');
  - A string is an array of characters and the instruction (others => ' ') means that all positions in this array are set to the character for blank.
- Once you have a variable User_Input of type string one can obtain a user input by using
  Get_Line(User_Input, Last);
  - The variable Last needs to be of type Integer or Natural (Natural are the non-negative natural numbers)
  - In the variable Last the length of the input by the user is stored.
  - Any characters occurring after position Last in User_Input will be unchanged by the instruction Get_Line.

Input of Strings from Console

- After an input of an integer from console, there are still some characters in the input buffer, which would be used by a Get_Line command.
  - So before having a proper Get_Line command, you need to clear the input bufffer by using a Get_Line command and ignoring its output:

    Get_Line(User_Input, Last);
- An exampe can be found in the lecture examples collection at **misc/conversion24hrClock12hrAMPMClock /vers2OnlyAdaWithIOIntegersNoLibrary/**

Functions vs Procedures

- Functions are like Java methods with return type not being void.
  - Usually they shouldn't have side effects
  - In fact in previous versions of SPARK Ada it was not allowed to have functions with side effects.
- Procedures are like Java methods with return type void.
  - So they only have side effects and no return value.

Example example13.adb

```
with Ada.Text_Io, Ada.Integer_Text_IO;
use Ada.Text_Io, Ada.Integer_Text_Io;

package body Example13 is

procedure Myproc( B : in Integer;
                  C :out Integer) is
begin
   Put("B=");
   Put(B);
   New_Line;
   C := B;
   Put("C=");
   Put(C);
   New_Line;
end MyProc;
```


Example example13.adb (Cont.)

```
function F(B,C : in Integer) return Integer is
begin
  return B * C;
end F;
```

Example example13.ads (Cont.)

```
package Example13 is

procedure Myproc(B : in Integer;
                 C : out Integer);

function F(B,C : in Integer) return Integer;

end Example13;
```

(**sect2a/example2a-11/**)

Sections 2 (b) - (h) will be distributed later