CIS 890: Safety Critical Systems

Lecture:
SPARK – *Functional Contract Notation*

Objectives

- Understand the motivation for SPARK's procedure contract language
- Be able to use correctly SPARK's annotations for
  - specifying pre- and post- conditions
  - specifying ranges
  - specifying universal and existential quantification
- Be aware of potential pitfalls associated with specifying procedural contracts

Outline

- Simple assertions
- Simple pre/post-conditions
- Return annotations for SPARK functions
- Quantification
- Update notation for arrays and records

Assertions

Assertions are specified in SPARK using the **`--# assert`** annotation, and can appear in the implementation of a procedure or function

```
--# assert C >= 0;
```

Boolean expression

Assertions

Example – using an assertion in a body of a loop to specify a loop invariant

```
procedure Divide(M, N : in Integer; Q, R : out Integer);
--# derives Q, R from M,N;
--# …
is
begin
   Q := 0;
   R := M;
   loop
      --# assert (M = Q * N + R) and (R >= 0);
      exit when R < N;
      Q := Q + 1;
      R := R - N;
   end loop;
end Divide;
```

Constraints on integer variables

Example File: ex0305a

Assertions

Example – using assertions to capture combinations of constraints including range constraints

```
function Value_Present (A: AType; X : Integer) return Boolean is
     Result : Boolean;
  begin
     Result := False;
     for I in Index loop
        if A(I) = X then
           Result := True; exit;
        end if;
        --# assert I in Index and not Result and
        --#        (for all M in Index range Index'First .. I
        --#                     => (A(M) /= X));
     end loop;
     return Result;
  end Value_Present;
```

`I`*'s value lies within the range* `Index`

Constraint on current version of return value

Universal quantification on values up to current value of loop variable

Procedure Contracts

Post-conditions describe functional properties of outputs & guarantees made to clients

```
procedure Add(X: in Integer);
--# global in out Total;
--# post Total = Total~ + X;
```

Variable modes indicate that Total has a value in both the pre-state and post-state

Final value of `Total` *equals initial value plus* `X` *(recall* `X` *cannot be changed in procedure)*

Tilde (~) indicates value of `Total` *in the procedure pre-state (before execution)*

Terminology

- Pre-state - value of all variables just before the method begins execution
- Post-state - value of all variables just after the method completes execution

Procedure Contracts

Pre-conditions state requirements/assumptions on clients

```
procedure Add(X: in Integer);
--# global in out Total;
--# pre X > 0;
--# post Total = Total~ + X;
```

Constraints on `X` *that the client must satisfy before calling procedure* `Add`*. Avoids errors in which clients call procedures with incorrect assumptions about parameter restrictions.*

Procedure Contracts

Another example of a simple *relational* post-condition that relates input and output values

```
procedure Exchange(X, Y : in out Float);
--# derives X from Y &
--#         Y from X;
--# post X = Y~ and Y = X~;
is
  T : Float;
begin
  T := X; X := Y; Y := T;
end Exchange;
```

Simple post condition stating that parameter values have been swapped.

Position of Tilde

One must be careful about the position of the tilde in some cases…

- `A~(I)`
  - Refers to position I (current value of I) in original value of A
  - Here, A is both imported/exported (not necessarily for I)
- `A~(I~)`
  - Refers to position I (original value of I) in the original value of A
  - Here both A and I are imported/exported
- `R~.C`
  - Refers to the initial value of component C for some record R
- `P.V~` (not `P~.V`)
  - Refers to the initial value of global variable V from some package P (`P~` doesn't make sense because a package does not change state)

Procedure Contracts

Building on our previous example…

```
procedure CAB(A, B, C : in out Float);
--# derives A from C &
--#         B from A &
--#         C from B;
--# post A = C~ and B = A~ and C = B~;
is
begin
   Exchange(A,B);
   Exchange(A,C);
end CAB;

procedure Exchange(X, Y : in out Float);
--# derives X from Y &
--#         Y from X;
--# post X = Y~ and Y = X~;
end Exchange;
```


Example File: ex0306b

Location of SPARK Contracts

Recall that a SPARK package includes both a package specifications (the public interface) and a package body

Package Specification

```
package MyPackage
--# own G1, G2;
is

type MyPublicType is…

procedure P(in X, out Y);
--# global in G1; out G2;
--# derives Y from X,
--#      & G2 from G1;
--# post Y = X + 1;

end MyPackage;
```

Contract is stated in package spec for a public procedure/ function

Package Body

```
package body MyPackage
is
G1: …
G2: …

type MyPrivateType is…

procedure P(in X, out Y) is
begin
   …P implementation…
end P;
end MyPackage;
```

- In the absence of data (own variable) refinement, we have two simple rules…
  - If a procedure/function is public (appears in the package spec), the contract is stated in the package spec. This exposes the contract to clients of the package.
  - If a procedure/function is private (appears only in the package body), the contract is stated in the package body

For You To Do

Pause the lecture and complete the following exercise (FYTD #1)…

- For the program below (fytd-01.ada), add appropriate pre/post-conditions to package to enforce the following constraints:
  - Pre-condition: X, Y, Z are unique values (e.g., X /= Y, etc.)
  - Post-condition: The final value of Max contains the greatest value held in one of the input variables X, Y, Z
- Use the Examiner to check that the syntax of your contract is correct

```
package P
is
   procedure FYTD01(
     X: in Integer;
     Y: in Integer;
     Z: in Integer;
     Max: out Integer);
end P;

package body P
is
  procedure FYTD01(X: in Integer;
                   Y: in Integer;
                   Z: in Integer;
                   Max: out Integer) is
  begin
    if (X > Y) and (X > Z) then
       Max := X;
    elsif (Y > X) and (Y > Z) then
       Max := Y;
    else
       Max := Z;
    end if;
  end FYTD01;
end P;
```


Return Annotation

SPARK functions do not update globals or parameters – they only provide a return value. Thus, "post-conditions" in functions are specified using `return` annotations.

```
function Inc(X: Integer) return Integer
--# return X + 1;
is
begin
 return X + 1;
end Inc;
```

An expression that describes the result of a function.

Return Annotation

Return annotations can introduce a specific name, e.g., `M`, to refer to the turn value

```
function Max (X, Y : Integer) return Integer
  --# return M => (X > Y -> M = X) and
  --#             (Y > X -> M = Y);
  is
     Result : Integer;
  begin
     if X > Y then
       Result := X;
     else
       Result := Y;
     end if;
     return Result;
  end Max;
```

Return M where is such that if X > Y then M equals X and if Y > X then M equals Y

Return Annotations

Impact of non-exhaustive cases in return annotation

```
function Max (X, Y : Integer) return Integer
  --# return M => (X > Y -> M = X) and
  --#             (Y > X -> M = Y);
  is
     Result : Integer;
  begin
     if X > Y then
       Result := X;
     elsif Y > X then
       Result := Y;
     else
       Result := 210;
     end if;
     return Result;
  end Max;
```

Cases for X,Y are not exhaustive, so there is a "hole" in the post-condition where the return value is not specified: when X=Y, and value can be returned as the result.

Any integer value that we choose to place here would satisfy the function contract.

Return Annotations

Care is needed to avoid incomplete specifications. One approach is to use the following form…

```
--# return M => (A -> M = X) and
--#             (B -> M = Y) and
--#             (C -> M = Z) and
--#             (A or B or C);
```

Guarantees that one of the antecedents is always true.

Return Annotations

Another pitfall: Overlapping ranges in return annotation…

```
function Max (X, Y : Integer) return Integer
   --# return M => (X >= Y -> M = X) and
   --#             (Y >= X -> M = Y);
   is
      Result : Integer;
   begin
      if X > Y then
        Result := X;
      else
         Result := Y;
      end if;
      return Result;
   end Max;
```

Ranges overlap. But we are OK for this example because in the range of overlap (X=Y) the return value (either X or Y) will be the same.

For You To Do

Pause the lecture and complete the following exercise (FYTD #2)…

- For the program below (fytd-02.ada) which corresponds to the previous FYTD example restated as a function, add appropriate pre-conditions and return value constraints to package to enforce the following constraints:
  - Pre-condition: X, Y, Z are unique values (e.g., X /= Y, etc.)
  - Return value constraint: The return value is the greatest value held in one of the input variables X, Y, Z
- Use the Examiner to check that the syntax of your contract is correct

```
package P
is
   function FYTD02(
     X: in Integer;
     Y: in Integer;
     Z: in Integer);
end P;

package body P
is
   function FYTD02(X: in Integer;
                   Y: in Integer;
                   Z: in Integer) is
      Result : Integer;
   begin
     if (X > Y) and (X > Z) then
         Result := X;
      elsif (Y > X) and (Y > Z) then
         Result := Y;
      else
         Result := Z;
      end if;
      return Result;
   end FYTD02;
end P;
```


Proof Contexts

Expressions in proof contexts (assert, check, pre/post) are written in SPARK with a few extensions…

- The expression in a return annotation can take the form of some identifier followed by `=>` and an extended expression as illustrated in previous example.
- The operators `->` (implies) and `<->` (is equivalent to) are available. Their precedence is the same as **`and`**, **`or`** and **`xor`**. They similarly cannot be mixed in an expression without the use of parentheses.
- Predicates can be quantified using **`for all`** and **`for some`**
  - e.g., `for all M in Index => (A(M) = 0)`
- There are forms for representing composite objects with some components changed.
  - `A[I => A(J); J => A(I)]`
    - …denotes the array `A` with components `I` and `J` interchanged. Similarly `D [Year => 1994]` denotes the record `D` of type `Date` with the `Year` component changed.

Ranges

Ranges are a common feature of SPARK contracts – especially those dealing with arrays

Consider the declarations…

```
subtype Index is Integer range 1 .. 100;
```

The range `Index` can be referenced in a number of ways…

- `I in Index`
- `I in Index range Index'First .. 50;`
- `I in Index range Index'First .. Index'Last – 1;`
- `I in Index range Index'First + 1 .. Index'Last – 49;`

The same notation can be used when referencing ranges in specifying the bounds of `for`-loops and when describing the ranges of quantified variables.

Ranges

Example of referring to the last value of a range…

```
procedure Inc(X: in out T)
--# derives X from X;
--# pre X < T'Last;
is
begin
  X := X + 1;
end Inc;
```

Max value of range type (annotation pointing to `T'Last`)

Example File: ex0304e

Quantification in SPARK

Quantification notation must reference a previously declared range…

```
subtype Index is Integer range 1 .. 10;
type AType is array (Index) of Integer;

function Value_Present (A: AType; X : Integer) return Boolean;
--# return for some M in Index => (A(M) = X);

function Value_Present (A: AType; X : Integer) return Boolean is
      Result : Boolean;
   begin
      Result := False;
      for I in Index loop
         if A(I) = X then
            Result := True; exit;
         end if;
         --# assert I in Index and not Result and
         --#        (for all M in Index range Index'First .. I
         --#                           => (A(M) /= X));
      end loop;
      return Result;
   end Value_Present;
```

Existential quantification

Universal quantification

Quantification in SPARK

Quantification notation must reference a previously declared range…

```
subtype Index is Integer range 1 .. 10;
type AType is array (Index) of Integer;

function Value_Present (A: AType; X : Integer) return Boolean;
--# return for some M in Index => (A(M) = X);

function Value_Present (A: AType; X : Integer) return Boolean is
      Result : Boolean;
   begin
      Result := False;
      for I in Index loop
         if A(I) = X then
            Result := True; exit;
         end if;
         --# assert I in Index and not Result and
         --#        (for all M in Index range Index'First .. I
         --#                           => (A(M) /= X));
      end loop;
      return Result;
   end Value_Present;
```

In this case, we only need to refer to the range name **Index** *to define the range of the quantified variable* **M***.*

In this case, we further constrain the value of **M** *by specifying a subrange of the named range.*

Loop Invariant Declaration

Asserts need to be properly placed to be recognized by the Examiner as loop invariants…

```
while Some_Condition
--# assert …
loop


for I in Some_Type range Some_Range
--# assert …
loop …
```


Update Notation

SPARK provides notation for representing composite objects (arrays, records) with some components changed.

Updating an array at a specific index position…

```
procedure Write( I: in Index; V: in Value) is
    --# global in out A;
    --# derives A from *, I, V;
    --# post A = A~[I => V];
begin
    A(I) := V;
end Write;
```

The post-value of **A** *is equal to the pre-value of* **A** (**A~**) *except for the value at position* **I** *which maps to* **V**

Update Notation

Swapping elements of two arrays at given positions…

```
type Index is range 1 .. 10;
type AType is array (Index) of Integer;

procedure Swap_Elements(I, J : in Index;
                        A : in out AType)
 --# derives A from A, I, J;
 --# post A = A~[I => A~(J); J => A~(I)];
is
 Temp : Integer;
begin
 Temp := A(I); A(I) := A(J); A(J) := Temp;
end Swap_Elements;
```

The post-value of **A** *is equal to the pre-value of* **A** *(***A~***) except for the value at position* **I** *maps to the value at index position* **J** *in the original value of the array, etc.*

Update Notation

Updating an record field…

```
type Point is
record
  X_Coord, Y_Coord: Real;
end record;

procedure UpdateX(P : in out Point, NewX: in Real)
 --# derives P from P, NewX;
 --# post P = P~[X_Coord => NewX];
is
begin
 P.X_Coord := NewX;
end UpdateX;
```

The post-value of **P** *is equal to the pre-value of* **P** *(***P~***) except for the value at field X_Coord maps to the value NewX.*

For You To Do

Pause the lecture and complete the following exercise (FYTD #3)…

- Write a procedure with the following signature and an accompanying contract (post-condition) that sets each component of the array to its index value.
  - `procedure Idarr(A : out Atype);`
- Write a function with the following signature to return the value of the maximum component of an array
  - `function Max_Value(A : Atype) return Integer;`
- Using the `Point` type on the previous slide, write a procedure and an accompanying contract (post-condition) that swaps the components of a `Point` value.

For each of your solutions, use the Examiner to check the syntax of your program/contracts.

Summary

SPARK's contract language allows assertions and pre/post-conditions to be specified for procedures and functions

- Assertions are boolean expressions that express a programmer's intention (constraints/properties that a programmer intends to hold) at a particular point in the program
- Assertions and pre/post conditions are based on SPARK boolean expressions plus…
  - Universal and existential quantification
  - Update notation for arrays and records
- Post-conditions support special notation (~) that allows values from the pre-state to be referenced
  - Allows *relational* post-conditions to be specified

Acknowledgements

- The material in this lecture is based on Chapters 3 and 11 from…
  - *High Integrity Software: The SPARK Approach to Safety and Security,* John Barnes, Addison Wesley, 2003.