SPARK verification features

Paul Jackson

School of Informatics
University of Edinburgh

Formal Verification
Spring 2018

Adding specification information to programs

- Verification concerns checking whether some model (or program) has desired properties
- An assertion is a logical formula that is associated with a point in the control-flow of a program.
  It describes a property of the program state that is desired true at that point.
- Assertions usually expressed in the language of Boolean expressions provided by the programming language, sometimes extended with $\forall$ and $\exists$ quantifiers.
- FV approaches try to logically establish that assertions hold for all possible execution paths leading to them.

Assertion pragmas

```
if X > Y then
   Max := X;
else
   Max := Y;
end if;
```

```
pragma Assert (Max >= X and Max >= Y
               and (Max = X or Max = Y)
              );
```

Freedom from runtime exceptions

Common causes of runtime exceptions include

- arithmetic overflow
- divide by zero
- array index out of bounds
- subrange/subtype constraint violation

```
  subtype T1 is Integer range 1 .. 10;
  V  : T1 := 10;  -- OK
begin
  V := 1 + V - 1; -- OK
  V := 1 + V;     -- EXCEPTION THROWN
```

Assertions automatically inserted to check these never occur

Formal analysis simplified by not having to consider exception scenarios

Runtime errors example

Consider

```
A (I + J) := P / Q;
```

What runtime errors might occur?

Answer:

- `I+J` might overflow the base-type of the index ranges subtype
- `I+J` might be outside the index ranges subtype
- `P/Q` might overflow the base-type of the element type
- `P/Q` might be outside the element subtype
- `Q` might be zero

Preconditions

A precondition is an assertion attached to the start of a subprogram (a function or a procedure).

```
procedure Increment (X: in out Integer)
   with Pre => (X < Integer'Last)
is
begin
   X := X + 1;
end Increment;
```

- FV assumes subprogram preconditions hold when checking assertions within the subprogram
- FV checks preconditions hold at each subprogram invocation

Postconditions

A postcondition is an assertion attached to control-flow points of a subprogram where control flow exits the subprogram

```
function Total_Above_Threshold (Threshold : in Integer)
  return Boolean
with
  Post => Total_Above_Threshold'Result = Total > Threshold;

procedure Add_To_Total (Incr : in Integer) with
  Post => Total = Total'Old + Incr;
```

- When analysing a subprogram, FV checks all postconditions hold
- At each control flow point for the return of a call to a subprogram, FV assumes any subprogram postconditions hold

Combining preconditions and postconditions

```
procedure Increment (X: in out Integer)
   with Pre => (X < Integer'Last)
        Post => X = X'Old + 1;
```

```
procedure Sqrt (Input : in Integer; Res: out Integer)
  with
       Pre  => Input >= 0,
       Post => (Res * Res) <= Input and
               (Res + 1) * (Res + 1) > Input;
```

Design by contract

Preconditions and postconditions

- form a contract between subprogram users and the subprogram implementers.
- if rich enough, provide full documentation to users – insulate them from implementation details
- promote modular design
    - Extend the abstract data type (ADT) paradigm that inspired OO programming and the separation of package specifications and bodies in Ada.
- promote modular verification.

  Hence enable scaling of FV.

Contract use example

```
procedure Add2 (X : in out Integer)
   with Pre => (X <= Integer'Last - 2)
is
begin
   Increment (X);
   Increment (X);
end Add2;
```

Will pre-conditions of both `Increment` calls be verified?

Answer: yes if `Increment` contract is specified with a post-condition.

SPARK flow analysis

Considers two issues:

- Interaction between subprograms and global state – what global state is read from and written to.
- Dependence of outputs of subprograms on inputs
  - Inputs and outputs include both parameters and global variables

SPARK notation allows desired flows to be specified

Tools then check flow specifications met

- Specification properties might related to code security
- Checks identify uninitialised variables, unused variables, ineffective code.

Assertion checking by tools relies on flow analysis to check that all variables initialised.

Global flow contract examples

```
procedure Set_X_To_Y_Plus_Z with
  Global => (Input  => (Y, Z), -- reads values of Y and Z
             Output => X);     -- modifies value of X

procedure Set_X_To_X_Plus_Y with
  Global => (Input  => Y,  -- reads value of Y
             In_Out => X); -- modifies value of X
                           -- also reads its initial value
```

Sometimes known as data flow or just data dependencies in SPARK documentation.

Intra-subprogram flow contract examples

```
procedure Swap (X, Y : in out T) with
  Depends => (X => Y,    -- X depends on initial value of Y
              Y => X);   -- Y depends on initial value of X

procedure Set_X_To_Y_Plus_Z with
  Depends => (X => (Y, Z));   -- X depends on Y and Z
```

Sometimes known as information flow or just flow dependencies in SPARK documentation.

Statically checking an assertion

Involves considering all execution paths leading to it.

Branches and joins in execution paths due to conditionals are no problem.

```
if X > Y then
   Max := X;
else
   Max := Y;
end if;
pragma Assert (Max >= X and Max >= Y);
```

Loops are an issue

Execution paths involving loops

Full set of execution paths through a loop

- might not be fixed size – could be data dependent
- could be very large

```
subtype Natural is Integer range 0 .. Integer'Last;

procedure Increment_Loop (X : in out Integer;
                          N : in Natural) with
  Pre  => X <= Integer'Last - N,
  Post => X = X'Old + N
is
begin
   for I in 1 .. N loop
      X := X + 1;
   end loop;
end Increment_Loop;
```

Breaking loops with assertions

A Loop invariant is an assertion inserted into a loop to split execution paths into well-defined segments.

```
procedure Inc_Loop_Inv (X : in out Integer; N : Natural) with
  Pre  => X <= Integer'Last - N,
  Post => X = X'Old + N
is
begin
   for I in 1 .. N loop
      X := X + 1;
      pragma Loop_Invariant (X = X'Loop_Entry + I);
   end loop;
end Inc_Loop_Inv;
```

Segments are:

- `Pre` ⟶ `Loop_Invariant`
- `Loop_Invariant` ⟶ `Loop_Invariant`
- `Loop_Invariant` ⟶ `Post`
- `Pre` ⟶ `Post` for when $\mathtt{N} = 0$

Euclidean linear division

```
procedure Linear_Div (I : in Integer; J : in Integer;
                      Q : out Integer; R  : out Integer;)
with
  Pre  => I >= 0 and J > 0
  Post => Q >= 0 and R >= 0 and R < J and J * Q + R = I
is
begin
   Q := 0;
   R := I;
   while R >= J loop
      pragma Loop_Invariant
        (R >= 0 and Q >= 0 and J * Q + R = I);
      Q := Q + 1;
      R := R - J;
   end loop;
end Linear_Div;
```

Looping through an array

```
subtype Index_T is Positive range 1 .. 1000;
subtype Component_T is Natural;
type Arr_T is array (Index_T) of Component_T;

procedure Validate_Arr_Zero (A : Arr_T; Success : out Boolean)
with
  Post => Success = (for all J in A'Range => A(J) = 0)
is
begin
   for J in A'Range loop
      if A(J) /= 0 then
         Success := False;
         return;
      end if;
      pragma Loop_Invariant ???;
   end loop;

   Success := True;
end Validate_Arr_Zero;
```

Looping through an array, with a loop invariant

```
subtype Index_T is Positive range 1 .. 1000;
subtype Component_T is Natural;
type Arr_T is array (Index_T) of Component_T;

procedure Validate_Arr_Zero (A : Arr_T; Success : out Boolean)
with
  Post => Success = (for all J in A'Range => A(J) = 0)
is
begin
   for J in A'Range loop
      if A(J) /= 0 then
         Success := False;
         return;
      end if;
      pragma Loop_Invariant
            (for all K in A'First .. J => A(K) = 0);
   end loop;

   Success := True;
end Validate_Arr_Zero;
```

Discovery & inference of loop invariants

- Reasoning with loop invariants is very much like induction on naturals

$$\frac{P(0) \quad \forall n : \mathbb{N}.\, P(n) \Rightarrow P(n+1)}{\forall n : \mathbb{N}.\, P(n)}$$

  - Checking loop invariant holds on first iteration like base case of induction
  - Checking loop invariant holds on later iteration, given it holds on immediately previous one like step case of induction
- Loop invariants often discovered by generalising post-condition, just as proof by induction involves first generalising the statement to be proven.
- Automatic discovery of loop invariants is an active research field
- Some cases are easy
  - GNATprove tool does infer bounds on for-loop indexes.

Showing loops terminate

Let Σ be the set of possible program states,
$\langle W, < \rangle$ be a well-founded order.

To show a loop terminates:

1. define a function $v : \Sigma \to W$
2. show

   $$v(s') < v(s)$$

   whenever s is the state at some point in the loop and s' is the state at the same point one iteration on.

Function v is called a variant function.

In Spark

- W is most typically some bounded arithmetic type, e.g. `Integer`.
- $<$ is conventional order or converse
- Also can have W containing tuples of arithmetic values, lexicographically ordered

Loop termination example

```
subtype Index is Positive range 1 .. 1_000_000;
type Text is array (Index range <>) of Integer;

function LCP (A : Text; X, Y : Integer) return Natural with
   Pre  => X in A'Range and then Y in A'Range,
is
   L : Natural;
begin
   L := 0;
   while X + L <= A'Last
      and then Y + L <= A'Last
      and then A (X + L) = A (Y + L)
   loop
      pragma Loop_Variant (Increases => L);
      L := L + 1;
   end loop;

   return L;
end LCP;
```