Subprogram Contracts

University.adacore.com

Subprogram Contracts in Ada 2012 and SPARK 2014

Originate in Floyd-Hoare logic (1967-1969)

- a Hoare triple {P}C{Q}
- P is the precondition before executing command C
- Q is the postcondition after executing command C

Executable version by Meyer in Eiffel (1988)

- Called Design by Contract ™
- Precondition is checked dynamically before a routine starts
- Postcondition is checked dynamically when a routine returns

SPARK 2014 combines both views

- SPARK 2005 version was only logic, Ada version is only executable

Dynamic Execution of Subprogram Contracts

Contract on subprogram declaration

- Different from subprogram body in general (but not always)

Ada Reference Manual allows implementations choice

- Contract can be checked in the caller or in the callee
- GNAT's choice is to execute in the callee

GNAT introduces wrappers in some cases for contracts

- For an imported subprogram (e.g. from C) with a contract
- For cases where contracts on static call/dispatching are different

Contracts are not enabled by default

- Switch -gnata enables dynamic checking of contracts in GNAT

Dynamic Behavior when Subprogram Contracts Fail

Violation of contract raises an exception

- Standard exception Assertion_Error is raised (same as for pragma Assert and all other *assertions*)
- Exception cannot be caught by subprogram's own exception handler → implementation choice caller/callee has no effect
- Idiom allows to select another exception

```
function Sqrt (X : Float) return Float with
  Pre => X >= 0.0 or else raise Argument_Error;
```

Control over sequencing of checks

- Typical pre/post is a conjunction of Boolean conditions
- Use **and** when no possible RTE, **and then** otherwise (recommended for SPARK)

Precondition

Better alternative to defensive programming, compare

```
function Sqrt (X : Float) return Float with
  Pre => X >= 0.0 or else raise Argument_Error;
```

and

```
--  X should be non-negative or Argument_Error is raised
function Sqrt (X : Float) return Float;

function Sqrt (X : Float) return Float is
begin
   if X >= 0.0 then
      raise Argument_Error;
   end if;
```

Preconditions can be activated alone

```
pragma Assertion_Policy (Pre => Check);
```

Postcondition

Single place to check all return paths from the subprogram

- Avoids duplication of checks before each return statement
- Much more robust during maintenance
- Only applies to *normal* returns (not in exception, not on abort)

Can relate input and output values

- Special attribute X'Old for referring to input value of variable X
- Special attribute Func'Result for referring to result of function Func
- Special attribute Rec'Update or Arr'Update for referring to modified value of record Rec or array Arr → replaced by *delta aggregate* syntax in Ada 202X: (Rec **with delta** Comp => Value)

Contract Cases

Convenient syntax to express a contract by cases

- Cases must be disjoint and complete (forming a *partition)*
- Introduced in SPARK, planned for inclusion in Ada 202X
- Case is (guard => consequence) with ‘Old/’Result in consequence
- Can be used in combination with precondition/postcondition

```
function Sqrt (X : Float) return Float with
  Contract_Cases =>
     (X > 1.0              => Sqrt'Result <= X,
      X = 1.0              => Sqrt'Result = 1.0,
      X < 1.0 and X > 0.0  => Sqrt'Result >= X,
      X = 0.0              => Sqrt'Result = 0.0);
```

Both a precondition and a postcondition

- On subprogram entry, exactly one guard must hold
- On subprogram exit, the corresponding consequence must hold

Attribute 'Old

X'Old expresses the input value of X in postconditions

- Same as X when variable not modified in the subprogram
- Compiler inserts a copy of X on subprogram entry → if X is large, copy can be expensive in memory footprint!
- X can be a variable, a function call, a qualification (but not limited!)

```
procedure Extract (A : in out My_Array;
                   J : in     Integer;
                   V :    out Value)
with
  Post => (if J in A'Range then V = A(J)'Old and A(J) = 0);
```

Expr'Old is rejected in *potentially unevaluated context*

- Pragma Unevaluated_Use_Of_Old(Allow) allows it
- In Ada, user is responsible – in SPARK, user can rely on proof

Implication and Equivalence

If-expression can be used to express an implication

- (**if** A **then** B) expresses the logical implication $(A \rightarrow B)$
- (**if** A **then** B **else** C) expresses the formula $(A \rightarrow B) \ \wedge (\neg A \rightarrow C)$
- (**if** A **then** B **else** C) can also be used with B, C not of Boolean type
- (A <= B) should not be used for expressing implication (same dynamic semantics, but less readable, and harmful in SPARK)

Equality can be used to express an equivalence

- (A = B) expresses the logical equivalence $(A \leftrightarrow B)$
- A double implication should not be used for expressing equivalence (same semantics, but less readable and maintainable)

Reasoning by Cases

Case-expression can be used to reason by cases

- Case test only on values of expressions of discrete type
- Can sometimes be an alternative to contract cases

```
procedure Open (F : in out File; Success : out Boolean) with
  Post =>
     (case F.Mode'Old is
        when Open   => Success,
        when Active => not Success,
        when Closed => Success = (F.Mode = Open));
```

- Can sometimes be used at different levels in the expression

```
procedure Open (F : in out File; Success : out Boolean) with
  Post =>
     Success = (case F.Mode'Old is
                   when Open   => True,
                   when Active => False,
                   when Closed => F.Mode = Open);
```

Universal and Existential Quantification

Quantified expressions can be used to express a property over a collection of values

- (**for all** X **in** A .. B => C) expresses the universally quantified property $(\forall\, X.X \geq A \wedge X \leq B \rightarrow C)$
- (**for some** X **in** A .. B => C) expresses the universally quantified property $(\exists\, X.X \geq A \wedge X \leq B \wedge C)$

Quantified expressions translated as loops at run time

- Control exits the loop as soon as the condition becomes false (resp. true) for a universally (resp. existentially) quantified expression

Quantification forms over array and collection content

- Syntax uses (**for all/some** V **of** … => C)

Expression Functions

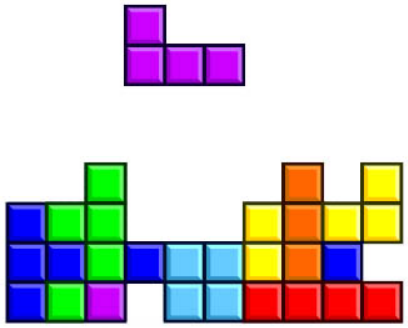
Without abstraction, contracts can become unreadable

- Also, use of quantifications can make them unprovable

Expression functions provide the means to abstract contracts

- Expression function is a function consisting in an expression
- Definition can complete a previous declaration
- Definition is allowed in a package spec! (crucial for proof with SPARK)

```
function Valid_Configuration return Boolean is
   (case Cur_State is
      when Piece_Falling | Piece_Blocked =>
        No_Overlap (Cur_Board, Cur_Piece),
      when Board_Before_Clean => True,
      when Board_After_Clean =>
        No_Complete_Lines (Cur_Board));
```



Quiz

Is this correct? 1/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
--  Fail systematically fails a precondition and catches the
--  resulting exception.

procedure Fail (Condition : Boolean) with
   Pre => Condition
is
   Bad_Condition : Boolean := False;
begin
   Fail (Bad_Condition);
exception
   when Assertion_Error => return;
end Fail;
```

Is this correct? 1/10

NO

```
--  Fail systematically fails a precondition and catches the
--  resulting exception.

procedure Fail (Condition : Boolean) with
   Pre => Condition
is
   Bad_Condition : Boolean := False;
begin
   Fail (Bad_Condition);
exception
   when Assertion_Error => return;
end Fail;
```

The exception from the recursive call is always caught in the handler, but not the exception raised if caller of Fail passes False as value for Condition.

Is this correct? 2/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
with Interfaces.C; use Interfaces.C;

procedure Memset
   (B   : in out char_array;
    Ch  : in     int;
    N   : in     size_t)
with
   Import,
   Pre  => N <= B'Length,
   Post => (for all Idx in B'Range =>
              (if Idx < B'First + N then
                 B (Idx) = Ch
               else
                 B (Idx) = B'Old (Idx)));
```

Is this correct? 2/10


```
with Interfaces.C; use Interfaces.C;

procedure Memset
   (B   : in out char_array;
    Ch  : in     int;
    N   : in     size_t)
with
   Import,
   Pre  => N <= B'Length,
   Post => (for all Idx in B'Range =>
              (if Idx < B'First + N then
                 B (Idx) = Ch
               else
                 B (Idx) = B'Old (Idx)));
```

GNAT will create a wrapper for checking the precondition and postcondition of Memset, calling the imported memset from libc.

Is this correct? 3/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
pragma Assertion_Policy (Pre => Ignore);
function Sqrt (X : Float) return Float with
  Pre => X >= 0.0;

pragma Assertion_Policy (Pre => Check);
function Sqrt (X : Float) return Float is
begin
   …
end Sqrt;
```

Is this correct? 3/10

NO

```
pragma Assertion_Policy (Pre => Ignore);
function Sqrt (X : Float) return Float with
  Pre => X >= 0.0;

pragma Assertion_Policy (Pre => Check);
function Sqrt (X : Float) return Float is
begin
   …
end Sqrt;
```

Although GNAT inserts precondition checks in the subprogram body instead of its caller, it is the value of Pre assertion policy at the declaration of the subprogram that decides if preconditions are activated.

Is this correct? 4/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
function Sqrt (X : Float) return Float with
  Pre => X >= 0.0;

function Sqrt (X : Float) return Float with
  Pre => X >= 0.0
is
begin
   …
end Sqrt;
```

Is this correct? 4/10

NO

```
function Sqrt (X : Float) return Float with
  Pre => X >= 0.0;

function Sqrt (X : Float) return Float with
  Pre => X >= 0.0
is
begin
   …
end Sqrt;
```

Contract is allowed only on the spec of a subprogram. Hence it is not allowed on the body when a separate spec is available.

Is this correct? 5/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
procedure Add (X, Y : Natural; Z : out Integer) with
  Contract_Cases =>
    (X <= Integer'Last - Y => Z = X + Y,
     others                => Z = 0)
is
begin
   Z := 0;
   Z := X + Y;
end Add;
```

Is this correct? 5/10

NO

```
procedure Add (X, Y : Natural; Z : out Integer) with
  Contract_Cases =>
    (X <= Integer'Last - Y => Z = X + Y,
     others                => Z = 0)
is
begin
   Z := 0;
   Z := X + Y;
end Add;
```

Postcondition is only relevant for normal returns.

Is this correct? 6/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
procedure Add (X, Y : Natural; Z : out Integer) with
  Post => Z = X + Y
is
begin
   Z := 0;
   Z := X + Y;
end Add;
```

Is this correct? 6/10


```
procedure Add (X, Y : Natural; Z : out Integer) with
  Post => Z = X + Y
is
begin
   Z := 0;
   Z := X + Y;
end Add;
```

Procedure may raise an exception, but postcondition correctly describes normal returns.

Is this correct? 7/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
procedure Add (X, Y : Natural; Z : out Integer) with
  Pre  => X <= Integer'Last - Y,
  Post => Z = X + Y
is
begin
   Z := X + Y;
end Add;
```

Is this correct? 7/10


```
procedure Add (X, Y : Natural; Z : out Integer) with
  Pre  => X <= Integer'Last - Y,
  Post => Z = X + Y
is
begin
   Z := X + Y;
end Add;
```

Precondition prevents exception inside Add.

Postcondition is always satisfied.

Is this correct? 8/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
procedure Memset
   (B   : in out String;
    Ch  : in     Character;
    N   : in     Natural)
with
   Pre  => N <= B'Length,
   Post => (for all Idx in B'Range =>
              (if Idx < B'First + N then
                 B (Idx) = Ch
               else
                 B (Idx) = B (Idx)'Old));
```

Is this correct? 8/10

NO

```
procedure Memset
    (B   : in out String;
     Ch  : in     Character;
     N   : in     Natural)
with
    Pre  => N <= B'Length,
    Post => (for all Idx in B'Range =>
               (if Idx < B'First + N then
                  B (Idx) = Ch
                else
                  B (Idx) = B (Idx)'Old));
```

'Old on expression including a quantified variable is not allowed.

Is this correct? 9/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
procedure Memset
   (B   : in out String;
    Ch  : in     Character;
    N   : in     Natural)
with
   Pre  => N <= B'Length - 1,
   Post => (for all Idx in 1 .. N => B (B'First + Idx - 1) = Ch)
     and then B (B'First + N) = B (B'First + N)'Old;
```

Is this correct? 9/10

NO

```
procedure Memset
   (B   : in out String;
    Ch  : in     Character;
    N   : in     Natural)
with
   Pre  => N <= B'Length - 1,
   Post => (for all Idx in 1 .. N => B (B'First + Idx - 1) = Ch)
     and then B (B'First + N) = B (B'First + N)'Old;
```

Expr'Old on potentially unevaluated expression is allowed only when Expr is a variable.

Is this correct? 10/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
procedure Memset
   (B  : in out String;
    Ch : in     Character;
    N  : in     Natural)
with
   Pre  => N <= B'Length - 1,
   Post => (for all Idx in 1 .. N => B (B'First + Idx - 1) = Ch)
     and B (B'First + N) = B (B'First + N)'Old;
```

Is this correct? 10/10


```
procedure Memset
   (B   : in out String;
    Ch  : in     Character;
    N   : in     Natural)
with
   Pre  => N <= B'Length - 1,
   Post => (for all Idx in 1 .. N => B (B'First + Idx - 1) = Ch)
     and B (B'First + N) = B (B'First + N)'Old;
```

Expr'Old does not appear anymore in a potentially unevaluated expression.

Another solution would have been to apply 'Old on B or to use pragma Unevaluated_Use_Of_Old(Allow);

university.adacore.com