SPARK 2014: State Abstraction

Presented by Claire Dross and Martyn Pike

University.adacore.com

State Abstraction – What is an Abstraction?

- **Two different views of the same object**
  - The abstraction captures what it does
  - The refinement provides more detail on how it is done
- **It is a well supported concept in Ada**
  - Packages and subprograms may have both a specification and an implementation
  - The specification and contracts are an abstraction of the body

```
procedure Increase (X : in out Integer) with
   Global => null,
   Pre    => X <= 100,
   Post   => X'Old < X;

procedure Increase (X : in out Integer) is
begin
   X := X + 1;
end Increase;
```

State Abstraction – Why is Abstraction Useful?

- **The specification summarizes what users may rely on**
  - Implementation of one abstraction should not depend on the implementation of another
- **Abstraction simplifies implementation and verification**
  - The users of an abstraction need only to understand its behavior
- **Abstraction simplifies maintenance and code reuse**
  - Changes to an object's implementation do not affect its users

```
procedure Increase (X : in out Integer) with
  Global => null,
  Pre    => X <= 100,
  Post   => X'Old < X;

while X <= 100 loop       --  The loop will terminate
  Increase (X);           --  Increase can be called safely
end loop;
pragma Assert (X = 101); --  Will this hold ?
```

State Abstraction – Abstraction of a Package's State

- **The variables declared in a package are part of its state**
- **The state of a package can be visible…**
  - Variables declared in the public part of the package's specification
- **… or hidden, allowing abstraction**
  - Variables declared in the private part of the package or in its body
  - They are typically accessed through subprogram calls
  - The implementation can be modified without updating client code

```
package Stack is
  procedure Pop  (E : out Element);
  procedure Push (E : in  Element);
end Stack;

package body Stack is
  Content : Element_Array (1 .. Max);
  Top     : Natural;
```

State Abstraction – Declaring a State Abstraction

- **A name, called *state abstraction*, can be introduced for the hidden state of a package using the Abstract_State aspect**
  - Several state abstractions can be introduced at once
  - A state abstraction is not a variable (it has no type), and cannot be used inside an expression

```
package Stack with
  Abstract_State => The_Stack
is
   ...

package Stack with
  Abstract_State => (Top_State, Content_State)
is
   ...

pragma Assert (Stack.Top_State = ...);
--  Compilation error: Top_State is not a variable
```

State Abstraction – Refining an Abstract State

- **Each state abstraction must be refined into its constituents using a Refined_State Aspect**
  - This aspect is specified in the package's body
  - It associates each state abstraction to the list of its constituents
  - Every hidden state, such as private variables, must be part of exactly one state abstraction
  - This refinement is mandatory, even if only one state abstraction is declared

```
package body Stack with
  Refined_State => (The_Stack => (Content, Top))
is
  Content : Element_Array (1 .. Max);
  Top     : Natural;
  --  Both Content and Top must be listed in the list of
  --  constituents of The_Stack
```

Sate Abstraction – Representing Private Variables

- **The private part of a package specification can be visible when its body is not**
- **In a package with an Abstract_State, private state must be associated to state abstractions at declaration**
  - It is done using the Part_Of aspect on its declaration
  - The hidden state must still be listed in the Refined_State aspect

```
package Stack with Abstract_State => The_Stack is

  procedure Pop  (E : out Element);
  procedure Push (E : in  Element);

private
  Content : Element_Array (...) with Part_Of => The_Stack;
  Top     : Natural             with Part_Of => The_Stack;
end Stack;

package body Stack with
  Refined_State => (The_Stack => (Content, Top))
```

State Abstraction – Additional State – Nested Packages

- **The state of a package nested inside a package P is a part of P's state**
  - If the nested package is hidden, its state is part of P's hidden state and must be listed in P's state refinement
  - If the nested package is public, its hidden state must be part of its own (public) state abstraction

```
package P with Abstract_State => State is
  package Visible_Nested with
    Abstract_State => Visible_State is
    ...
end P;

package body P with
  Refined_State => (State => Hidden_Nested.Hidden_State)
is
  package Hidden_Nested with
    Abstract_State => Hidden_State is
```

State Abstraction – Additional State – Constants with Variable Inputs

- **Constants with variable inputs are considered as variables in contracts**
  - A constant has variable inputs if its value depends on a variable, a parameter, or another constant with variable inputs
  - Constants with variable inputs participate to the flow of information between variables
  - Constants with variable inputs are part of the state of a package and must be listed in its state refinement

```
package body Stack with
  Refined_State => (The_Stack => (Content, Top, Max))
is
  Max     : constant Natural := External_Variable;
  Content : Element_Array (1 .. Max);
  Top     : Natural;
  --  Max has variable inputs. It must appear as a
  --  constituent of The_Stack
```

State Abstraction – Subprogram Contracts – Global and Depends

- **State abstractions can be used in Depends and Global contracts in place of the hidden state they represent**
  - State abstraction may induce loss in precision if a state abstraction aggregates several variables

```
package Stack with
  Abstract_State => (Top_State, Content_State) is

  procedure Pop  (E : out Element) with
    Global  => (Input  => Content_State,
                In_Out => Top_State),
    Depends => (Top_State => Top_State,
                E         => (Content, Top_State));
```

```
package Stack with
  Abstract_State => The_Stack is

  procedure Pop  (E : out Element) with
    Global  => (In_Out => The_Stack),
    Depends => ((The_Stack, E) => The_Stack);
```

State Abstraction – Subprogram Contracts – Global and Depends

- **Global and Depends contracts referring to state abstractions can be refined using the Refined_Global and Refined_Depends aspects**
  - The refined aspects should be associated to the subprogram's body, where the state refinement is visible
  - They refer directly to hidden state instead of state abstractions
  - Refined Global and Depends are used for internal calls
  - Refined Global and Depends contracts are optional

```
package body Stack
   ...
   procedure Pop  (E : out Element) with
     Refined_Global  => (Input  => Content,
                         In_Out => Top),
     Refined_Depends => (Top => Top,
                         E   => (Content, Top)) is
```

State Abstraction – Subprogram Contracts – Pre and Postconditions

- **Refinement in pre and postconditions is usually handled using expression functions**
  - Inside the package, the body of the expression function can be used for verification
  - Outside the package, the expression function is a black box

```
package Stack
   ...
   function Is_Empty return Boolean;
   function Is_Full  return Boolean;

   procedure Push (E : Element) with
     Pre  => not Is_Full,
     Post => not Is_Empty;

package body Stack
   ...
   function Is_Empty return Boolean is (Top = 0);
   function Is_Full  return Boolean is (Top = Max);
```

State Abstraction – Subprogram Contracts – Pre and Postconditions

- **The Refined_Post aspect can be used to strengthen a postcondition**
  - Like with expression functions, refined postconditions will only be available for internal calls
  - It must be stronger than the subprogram's postcondition
  - There are no counterparts for preconditions

```
package Stack
   ...
   procedure Push (E : Element) with
     Pre  => not Is_Full,
     Post => not Is_Empty;

package body Stack
   ...
   procedure Push (E : Element) with
     Refined_Post => not Is_Empty and E = Content (Top);
```

State Abstraction – Initialization of Local Variables

- **The Initializes aspect allows specifying the variables initialized during a package's elaboration**
  - It is optional, if not provided, an approximation of the set of initialized variables will be computed by the tool
  - If an Initializes aspect is provided, it must list all the state (both hidden and visible) initialized during the package's elaboration
  - Initializes refers to private variables using state abstractions
  - Note that in SPARK, only variables defined in the unit can be written at elaboration

```
package Stack with
  Abstract_State => The_Stack,
  Initializes    => The_Stack
is
--  Flow analysis will make sure both Top and Content are
--  initialized at package elaboration
```

State Abstraction – Initialization of Local Variables

- **If the initial value of a variable or state abstraction depends on an external variable, the relation must be stated in the Initializes aspect**
  - Like in Depends contracts, relations between variables are represented using an arrow
  - If an initialized state does not depend on any variable defined outside the package, the dependency can be omitted

```
package P with
  Initializes => (V1, V2 => External_Variable)
is
  V1 : Integer := 0;
  V2 : Integer := External_Variable;
end P;

--  The association for V1 is omitted, it does not depend
--  on any external state.
```

Quiz

Is this correct? 1/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
package Communication with
  Abstract_State => State,
  Initializes    => (State => External_Variable)
is
  ...
private
  package Ring_Buffer is
    Capacity : constant Natural := External_Variable;
    ...
  end Ring_Buffer;

  ...
end Communication;

package body Communication with
  Refined_State => (State => Ring_Buffer.Capacity) is
...
```

Is this correct? 1/10

NO

```
package Communication with
  Abstract_State => State,
  Initializes    => (State => External_Variable)
is
  ...
private
  package Ring_Buffer is
    Capacity : constant Natural := External_Variable;
    ...
  end Ring_Buffer;

  ...
end Communication;

package body Communication with
  Refined_State => (State => Ring_Buffer.Capacity) is
...
```

Here, Capacity is declared in the private part of Communication. Therefore, it should be linked to State at declaration using the Part_Of aspect.

Is this correct? 2/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
package Communication with
  Abstract_State => State,
  Initializes    => (State => External_Variable)
is
  ...
private
  package Ring_Buffer with
    Abstract_State => (B_State with Part_Of => State)
  is
    ...
  private
    Capacity : constant Natural := External_Variable with
      Part_Of => B_State;
    ...
  end Ring_Buffer;

  ...
end Communication;
```

Is this correct? 2/10

YES

```
package Communication with
  Abstract_State => State,
  Initializes    => (State => External_Variable)
is
  ...
private
  package Ring_Buffer with
    Abstract_State => (B_State with Part_Of => State)
  is
    ...
  private
    Capacity : constant Natural := External_Variable with
      Part_Of => B_State;
    ...
  end Ring_Buffer;

  ...
end Communication;
```

This program is correct and GNATprove will be able to verify it.

Is this correct? 3/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
package Counting with Abstract_State => State is
  procedure Reset_Black_Count;
  procedure Reset_Red_Count;
  ...
end Counting;

package body Counting with
  Refined_State => (State => (Black_Counter, Red_Counter))
is
  Black_Counter, Red_Counter : Natural;

  procedure Reset_Black_Count is
  begin
    Black_Counter := 0;
  end Reset_Black_Count;
  ...
end Counting;

procedure Main is
begin
   Reset_Black_Count;
   Reset_Red_Count;
   ...
end Main;
```

Is this correct? 3/10

YES

```
package Counting with Abstract_State => State is
  ...
end Counting;

package body Counting with
  Refined_State => (State => (Black_Counter, Red_Counter))
is
  Black_Counter, Red_Counter : Natural;

  procedure Reset_Black_Count is
  begin
    Black_Counter := 0;
  end Reset_Black_Count;
  ...
end Counting;

procedure Main is
begin
   Reset_Black_Count;
   ...
```

The generated Global contract for Reset_Black_Count is (In_Out => State)

This program does not read uninitialized data, but GNATprove will fail to verify this fact. As we have provided a state abstraction, flow analysis will compute subprogram's effects in terms of this state abstraction, and thus, will count the call to Reset_Black_Count as a read of State.

Is this correct? 4/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
package Counting is
  procedure Reset_Black_Count;
  procedure Reset_Red_Count;
  ...
end Counting;

package body Counting is
  Black_Counter, Red_Counter : Natural;

  procedure Reset_Black_Count is
  begin
    Black_Counter := 0;
  end Reset_Black_Count;
  ...
end Counting;

procedure Main is
begin
   Reset_Black_Count;
   Reset_Red_Count;
   ...
end Main;
```

Is this correct? 4/10

YES

```
package Counting is
  procedure Reset_Black_Count;
  procedure Reset_Red_Count;
  ...
end Counting;

package body Counting is
  Black_Counter, Red_Counter : Natural;

  procedure Reset_Black_Count is
  begin
    Black_Counter := 0;
  end Reset_Black_Count;
  ...
end Counting;

procedure Main is
begin
   Reset_Black_Count;
   Reset_Red_Count;
   ...
end Main;
```

Here, no state abstraction is provided. GNATprove will reason in terms of variables and will prove data initialization without any problem.

Is this correct? 5/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
package Counting with Abstract_State => State is
  procedure Reset_Black_Count with Global => (In_Out => State);
  procedure Reset_Red_Count   with Global => (In_Out => State);
  procedure Reset_All         with Global => (Output => State);
  ...
end Counting;

package body Counting with
  Refined_State => (State => (Black_Counter, Red_Counter))
is
  Black_Counter, Red_Counter : Natural;

  procedure Reset_Black_Count with
    Refined_Global => (Output => Black_Counter) is ...

  procedure Reset_Red_Count with
    Refined_Global => (Output => Red_Counter) is ...

  procedure Reset_All is
  begin
    Reset_Black_Count;
    Reset_Red_Count;
  end Reset_All;
end Counting;
```

Is this correct? 5/10

YES

```
package Counting with Abstract_State => State is
  procedure Reset_Black_Count with Global => (In_Out => State);
  procedure Reset_Red_Count   with Global => (In_Out => State);
  procedure Reset_All         with Global => (Output => State);
  ...
end Counting;

package body Counting with
  Refined_State => (State => (Black_Counter, Red_Counter))
is
  Black_Counter, Red_Counter : Natural;

  procedure Reset_Black_Count with
    Refined_Global => (Output => Black_Counter) is ...

  procedure Reset_Red_Count with
    Refined_Global => (Output => Red_Counter) is ...

  procedure Reset_All is
  begin
    Reset_Black_Count;
    Reset_Red_Count;
```

Flow analysis uses the refined version of Global contracts for internal calls and thus can verify that Reset_All indeed properly initializes State. Note that Refined_Global and Global annotations are not mandatory, they can also be computed by the tool.

Is this correct? 6/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
package Stack with Abstract_State => The_Stack is
  pragma Unevaluated_Use_Of_Old (Allow);

  type Element_Array is array (Positive range <>) of Element;
  Max : constant Natural := 100;
  subtype Length_Type is Natural range 0 .. Max;

  procedure Push (E : Element) with
    Post => not Is_Empty and
    (if Is_Full'Old then The_Stack = The_Stack'Old else Peek = E);

  function Peek     return Element with Pre => not Is_Empty;
  function Is_Full  return Boolean;
  function Is_Empty return Boolean;
end Stack;

package body Stack with
  Refined_State => (The_Stack => (Top, Content)) is
  Top     : Length_Type := 0;
  Content : Element_Array (1 .. Max);

  procedure Push (E : Element) is ...;
  function  Peek     return Element is (Content (Top));
  function  Is_Full  return Boolean is (Top >= Max);
  function  Is_Empty return Boolean is (Top = 0);
end Stack;
```

Is this correct? 6/10

NO

```
package Stack with Abstract_State => The_Stack is
  pragma Unevaluated_Use_Of_Old (Allow);

  type Element_Array is array (Positive range <>) of Element;
  Max : constant Natural := 100;
  subtype Length_Type is Natural range 0 .. Max;

  procedure Push (E : Element) with
    Post => not Is_Empty and
    (if Is_Full'Old then The_Stack = The_Stack'Old else Peek = E);

  function Peek     return Element with Pre => not Is_Empty;
  function Is_Full  return Boolean;
  function Is_Empty return Boolean;
end Stack;

package body Stack with
  Refined_State => (The_Stack => (Top, Content)) is
  Top     : Length_Type := 0;
  Content : Element_Array (1 .. Max);

  procedure Push (E : Element) is ...;
  ...
```

There is a compilation error in Push's postcondition. Indeed, The_Stack is a state abstraction and not a variable and cannot be mentioned in an expression.

Is this correct? 7/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
package Stack with Abstract_State => The_Stack is
   ...
   type Stack_Model is private;

   procedure Push (E : Element) with
     Post => not Is_Empty and
     (if Is_Full'Old then Get_Stack = Get_Stack'Old else Peek = E);

   function Peek       return Element with Pre => not Is_Empty;
   function Is_Full    return Boolean;
   function Is_Empty   return Boolean;
   function Get_Stack return Stack_Model;
private
   type Stack_Model is record
     Top      : Length_Type := 0;
     Content : Element_Array (1 .. Max);
   end record;
end Stack;

procedure Use_Stack (E : Element) with Pre => not Is_Empty is
    F : Element := Peek;
begin
    Push (E);
    pragma Assert (Peek = E or Peek = F);
end Use_Stack;
```

Is this correct? 7/10

YES

```
package Stack with Abstract_State => The_Stack is
   ...
   type Stack_Model is private;

   procedure Push (E : Element) with
     Post => not Is_Empty and
     (if Is_Full'Old then Get_Stack = Get_Stack'Old else Peek = E);

   function Peek      return Element with Pre => not Is_Empty;
   function Is_Full   return Boolean;
   function Is_Empty  return Boolean;
   function Get_Stack return Stack_Model;
private
   …
end Stack;

procedure Use_Stack (E : Element) with Pre => not Is_Empty is
    F : Element := Peek;
begin
    Push (E);
    pragma Assert (Peek = E or Peek = F);
end Use_Stack;
```

This program is correct, but GNATprove won't be able to verify the assertion in Use_Stack. Indeed, even if Get_Stack is an expression function, its body is not visible outside of Stack's body.

Is this correct? 8/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
package Stack with Abstract_State => The_Stack is
  ...
  procedure Push (E : Element) with
    Post => not Is_Empty and
    (if Is_Full'Old then Get_Stack = Get_Stack'Old else Peek = E);
  ...
private
  ...
  Top     : Length_Type := 0          with Part_Of => The_Stack;
  Content : Element_Array (1 .. Max) with Part_Of => The_Stack;

  function Peek      return Element     is (Content (Top));
  function Is_Full   return Boolean     is (Top >= Max);
  function Is_Empty  return Boolean     is (Top = 0);
  function Get_Stack return Stack_Model is ((Top, Content));
end Stack;

procedure Use_Stack (E : Element) with Pre => not Is_Empty is
   F : Element := Peek;
begin
   Push (E);
   pragma Assert (Peek = E or Peek = F);
end Use_Stack;
```

Is this correct? 8/10

YES

```
package Stack with Abstract_State => The_Stack is
   ...
   procedure Push (E : Element) with
     Post => not Is_Empty and
     (if Is_Full'Old then Get_Stack = Get_Stack'Old else Peek = E);
   ...
private
   ...
   Top     : Length_Type := 0            with Part_Of => The_Stack;
   Content : Element_Array (1 .. Max) with Part_Of => The_Stack;

   function Peek      return Element     is (Content (Top));
   function Is_Full   return Boolean     is (Top >= Max);
   function Is_Empty  return Boolean     is (Top = 0);
   function Get_Stack return Stack_Model is ((Top, Content));
end Stack;

procedure Use_Stack (E : Element) with Pre => not Is_Empty is
    F : Element := Peek;
begin
    Push (E);
    pragma Assert (Peek = E or Peek = F);
end Use_Stack;
```

GNATprove will be able to verify the assertion in Use_Stack since it has visibility of Get_Stack's body.

Is this correct? 9/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
package External_Interface with
  Abstract_State => File_System,
  Initializes    => File_System
is
  procedure Read_Data (File_Name : String; Data : out Data_Record)
  with Global => File_System;
end External_Interface;

package Data with Initializes => (Data_1, Data_2, Data_3) is
  pragma Elaborate_Body;
  Data_1 : Data_Type_1;
  Data_2 : Data_Type_2;
  ...
end Data;

pragma Elaborate_All (External_Interface);
package body Data is
begin
  declare
    Data_Read : Data_Record;
  begin
    Read_Data ("data_file_name", Data_Read);
    Data_1 := Data_Read.Field_1;
    ...
```

Is this correct? 9/10

NO

```
package External_Interface with
  Abstract_State => File_System,
  Initializes    => File_System
is
  procedure Read_Data (File_Name : String; Data : out Data_Record)
  with Global => File_System;
end External_Interface;

package Data with Initializes => (Data_1, Data_2, Data_3) is
  pragma Elaborate_Body;
  Data_1 : Data_Type_1;
  Data_2 : Data_Type_2;
  ...
end Data;

pragma Elaborate_All (External_Interface);
package body Data is
begin
  declare
    Data_Read : Data_Record;
  begin
    Read_Data ("data_file_name", Data_Read);
    Data_1 := Data_Read.Field_1;
```

The dependency between Data_1's initial value and File_System must be listed in Data's Initializes aspect.

Is this correct? 10/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
package Data is
  pragma Elaborate_Body;
  Data_1 : Data_Type_1;
  Data_2 : Data_Type_2;
  ...
end Data;

pragma Elaborate_All (External_Interface);
package body Data is
begin
  declare
    Data_Read : Data_Record;
  begin
    Read_Data ("data_file_name", Data_Read);
    Data_1 := Data_Read.Field_1;
    ...
  end;
end Data;

procedure Use_Data is
  X : Data_Type_1 := Data_1;
begin
  ...
```

Is this correct? 10/10

YES

```
package Data is
  pragma Elaborate_Body;
  Data_1 : Data_Type_1;
  Data_2 : Data_Type_2;
  ...
end Data;

pragma Elaborate_All (External_Interface);
package body Data is
begin
  declare
    Data_Read : Data_Record;
  begin
    Read_Data ("data_file_name", Data_Read);
    Data_1 := Data_Read.Field_1;
    ...
  end;
end Data;

procedure Use_Data is
  X : Data_Type_1 := Data_1;
begin
  ...
```

Since Data has no Initializes aspect, GNATprove will compute the set of variables initialized during its elaboration. Thereby, it can ensure that Data_1 is always initialized in Use_Data.

university.adacore.com