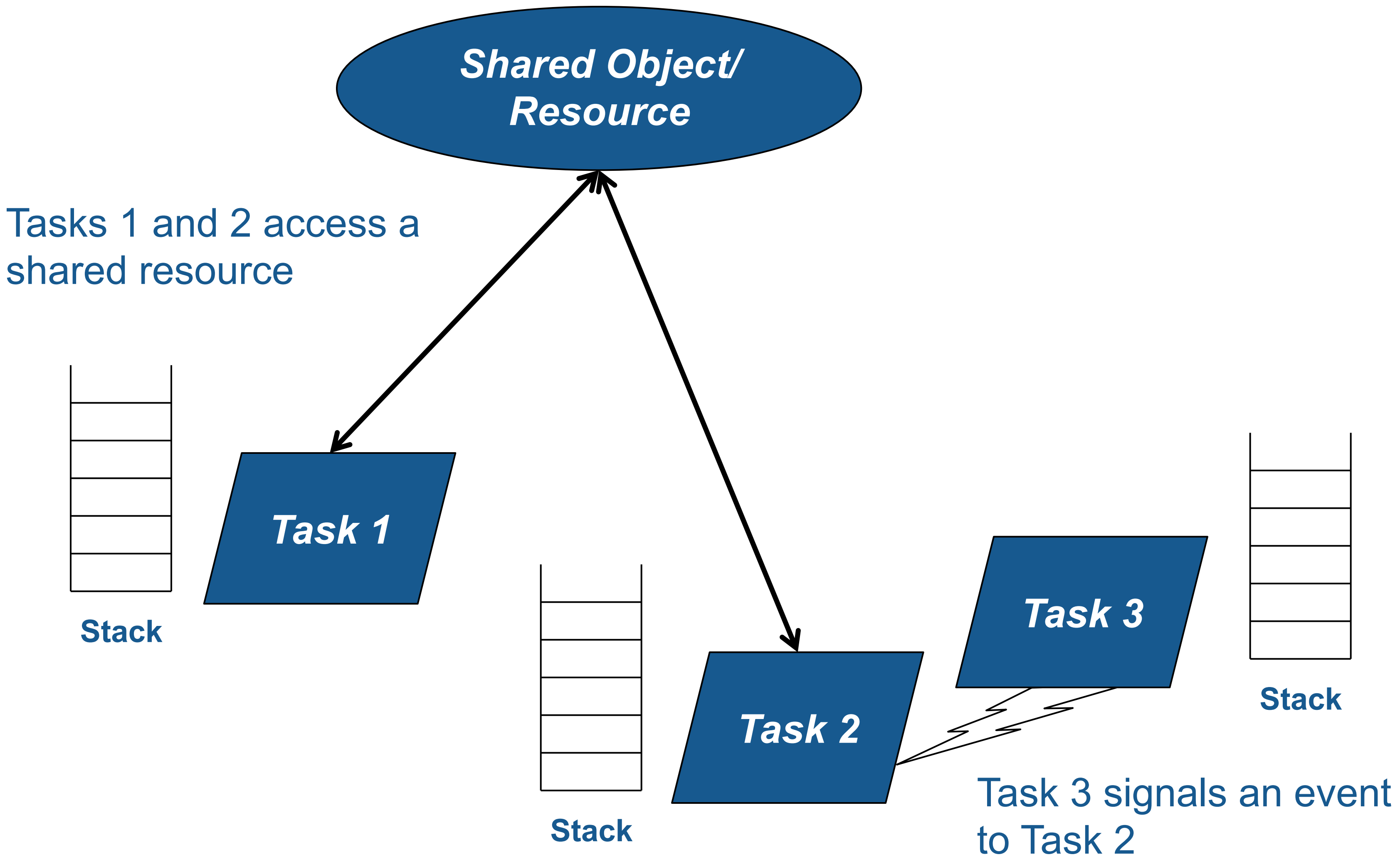
SPARK 2014: Concurrency

University.adacore.com

Concurrency ≠ Parallelism

- **Concurrency allows to create a well structured program**
- **Parallelism allows to create a high performance program**
- **Multiple cores/processors are…**
  - possible for concurrent programs
  - essential to parallelism
- **What about Ada and SPARK?**
  - GNAT runtimes for concurrency available on single core & multicore (for SMP platforms)
  - parallel features scheduled for inclusion in Ada and SPARK 202x

Concurrent Program Structure in Ada



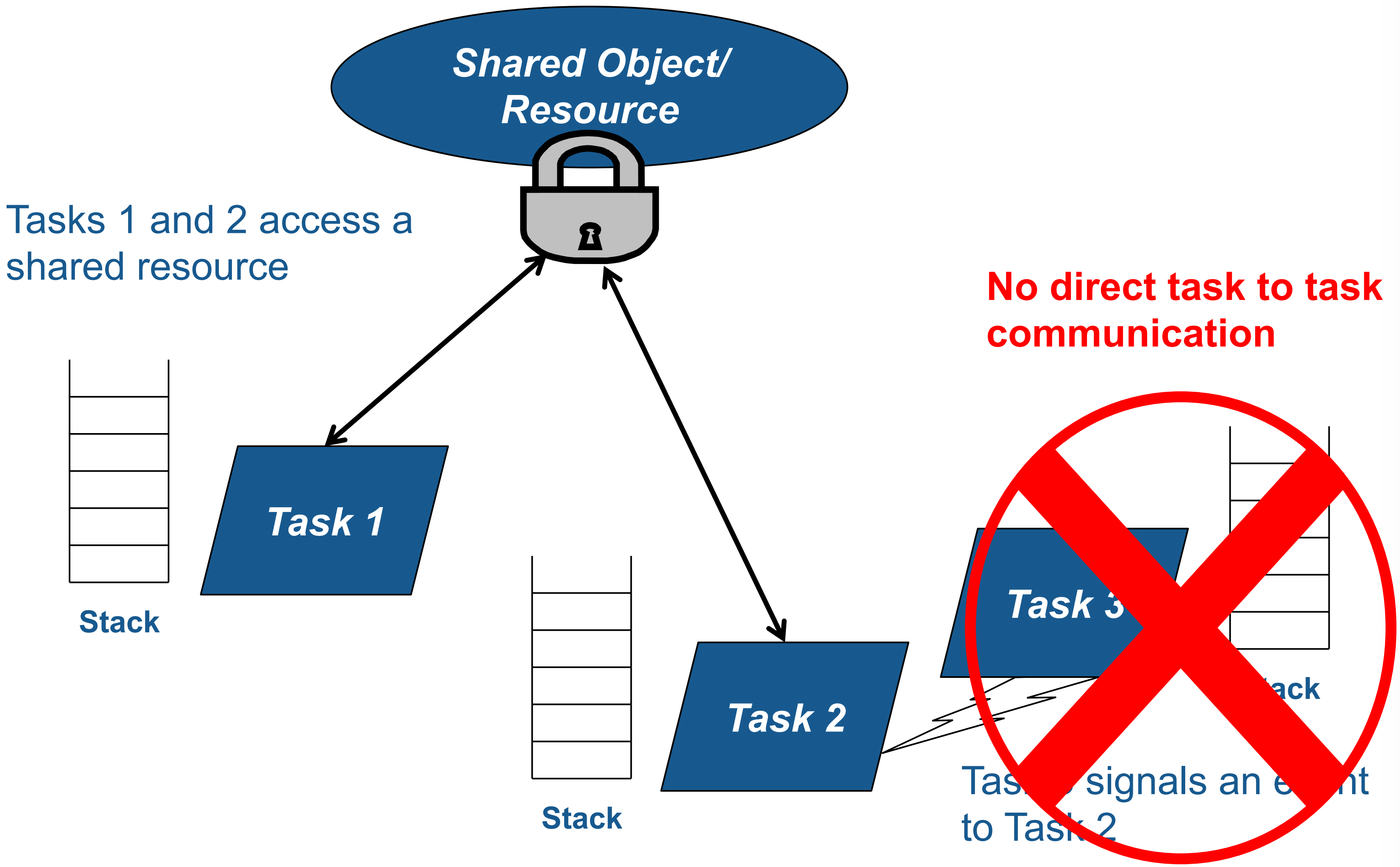
The problems with concurrency

- **Control and data flow become much more complex**
  - possibly nondeterministic even
  - actual behavior is one of many possible interleavings of tasks
- **Data may be corrupted by concurrent accesses**
  - so called *data races* or *race conditions*
- **Control may block indefinitely, or loop indefinitely**
  - so called *deadlocks* and *livelocks*
- **Scheduling and memory usage are harder to compute**

Ravenscar – the Ada solution to concurrency problems

- **Ravenscar profile restricts concurrency in Ada**
  - ensures deterministic behavior at every point in time
  - recommends use of protected objects to avoid data races
  - prevents deadlocks with Priority Ceiling Protocol
  - allows use of scheduling analysis techniques (RMA, RTA)
  - facilitates computation of memory usage with static tasks
- **GNAT Extended Ravenscar profile lifts some restrictions**
  - still same benefits as Ravenscar profile
  - removes painful restrictions for some applications

Concurrent Program Structure in Ravenscar



Ravenscar – the SPARK solution to concurrency problems

- **Ravenscar and Extended_Ravenscar profiles supported in SPARK**
- **Data races prevented by flow analysis**
  - ensures no problematic concurrent access to unprotected data
  - flow analysis also ensures non-termination of tasks
- **Run-time errors prevented by proof**
  - includes violations of the Priority Ceiling Protocol

Concurrency – A trivial example

```
task type T;

Id : Task_Id;

task body T is
begin
   loop
      Id := Current_Task;
   end loop;
end T;

T1, T2 : T;
```

Id can be written by T1 and T2 at the same time

Setup for using concurrency in SPARK

- **Any unit using concurrency features (tasks, protected objects, etc.) must set the profile**

```
pragma Profile (Ravenscar);
--  or
pragma Profile (GNAT_Extended_Ravenscar);
```

- **… plus an additional pragma**

```
pragma Partition_Elaboration_Policy (Sequential);
```

  – that ensures tasks start after the end of elaboration

- **… which are checked by GNAT partition-wide**

  – pragmas needed for *verification* even it not for *compilation*

Tasks in Ravenscar

- **A task can be either a singleton object or a type**

```
task T;
task type TT;
```

no declarations of entries for rendez-vous

- **… completed by a body**

```
task body T is
begin
   loop
      …
   end loop;
end T;
```

infinite loop to prevent termination

- **Tasks are declared at library-level**
- **… as standalone objects or inside records/arrays**

```
type TA is array (1 .. 3) of TT;
type TR is record
   A, B : TT;
end record;
```

Communication Between Tasks in Ravenscar

- **Tasks can communicate through protected objects**
- **A protected object is either a singleton object or a type**

```
protected (type) P is
   procedure Set (V : Natural);
   function Get return Natural;
private
   The_Data : Natural := 0;
end P;
```

all PO private data initialized by default in SPARK

- **… completed by a body**

```
protected body P is
   procedure Set (V : Natural) is
   begin
      The_Data := V;
   end Set;
   function Get return Natural is
     (The_Data);
end P;
```

Protected Objects in Ravenscar

- **Protected objects are declared at library-level**
- **… as standalone objects or inside records/arrays**

```
P : PT;

type PAT is array (1 .. 3) of PT;
PA : PAT;

type PRT is record
   A, B : PT;
end record with Volatile;
PR : PRT;
```

The record type needs to be volatile, as a non-volatile type cannot contain a volatile component.

The array type is implicitly volatile when its component type is volatile.

Protected Communication with Procedures & Functions

- **CREW enforced (Concurrent-Read-Exclusive-Write)**
  - procedures have exclusive read-write access to PO
  - functions have shared read-only access to PO
- **Actual mechanism depends on target platform**
  - scheduler enforces policy on single core
  - locks used on multicore (using CAS instructions)
  - lock-free transactions used for simple PO (again using CAS)
- **Mechanism is transparent to user**
  - user code simply calls procedures/functions
  - task may be queued until PO is released by another task

Blocking Communication with Entries

- **Only protected objects have entries in Ravenscar**
- **Entry = procedure with "entry" guard condition**
  - second level of queues, one for each entry, on a given PO
  - task may be queued until guard is True and PO is released

```
protected (type) P is
   entry Reset;
private
   Is_Not_Null : Boolean := False;
   …
```

at most one entry in Ravenscar

```
protected body P is
   entry Reset when Is_Not_Null is
   begin
      The_Data := 0;
   end Reset;
end P;
```

guard is a Boolean component of PO in Ravenscar

Relaxed Constraints on Entries with Extended Ravenscar

- **Proof limitations with Ravenscar**
  - not possible to relate guard to other components with invariant
- **GNAT Extended Ravenscar profile lifts these constraints**
  - and allows multiple tasks to call the same entry

```
protected type Mailbox is
   entry Publish;
   entry Retrieve;
private
   Num_Messages : Natural := 0;
   …
```

```
protected body Mailbox is
   entry Publish when Num_Messages < Max is …
   entry Retrieve when Num_Messages > 0 is …
end P;
```

Interrupt Handlers in Ravenscar

- **Interrupt handlers are parameterless procedures of PO**
  - with aspect Attach_Handler specifying the corresponding signal
  - with aspect Interrupt_Priority on the PO specifying the priority

```
protected P with
  Interrupt_Priority =>
    System.Interrupt_Priority'First
is
   procedure Signal with
     Attach_Handler => SIGHUP;
   …
```

- **Priority of the PO should be in System.Interrupt_Priority**
  - default is OK – in the range of System.Interrupt_Priority
  - checked by proof (default or value of Priority or Interrupt_Priority)

Other Communications Between Tasks in SPARK

- **Tasks must communicate through synchronized objects**
  - atomic objects
  - protected objects
  - suspension objects (standard “Boolean” protected objects)
- **Constants are considered as synchronized**
  - this includes variables constant after elaboration (specified with aspect Constant_After_Elaboration)
- **Single task or PO can access an unsynchronized object**
  - exclusive relation between object and task/PO must be specified with aspect Part_Of

Data and Flow Dependencies of Tasks

- **Input/output relation can be specified for a task**
  - as task never terminates, output is understood while task runs
  - task itself is both an input and an output

```
task T with
   Global => (Input => X,
              Output => Y,
              In_Out => Z),
   Depends => (T => T,
               Z => Y,
               Y => X)
is
   …
```

implicit In_Out => T

explicit dependency

State Abstraction over Synchronized Variables

- **Synchronized objects can be abstracted in synchronized abstract state with aspect Synchronous**

```
package P with
   Abstract_State => (State with Synchronous)
is …
```

```
package body P with
   Refined_State => (State => (A, P, T))
is
   A : Integer with Atomic, Async_Readers, Async_Writers;
   P : Protected_Type;
   T : Task_Type;
end P;
```

- **Synchronized state is a form of external state**
  – Synchronous same as External => (Async_Readers, Async_Writers)
  – tasks are not volatile and can be part of regular abstract state

Synchronized Abstract State in the Standard Library

- **Standard library maintains synchronized state**
  - the tasking runtime maintains state about running tasks
  - the real-time runtime maintains state about current time

```
package Ada.Task_Identification with
   SPARK_Mode,
   Abstract_State =>
     (Tasking_State with Synchronous,
        External => (Async_Readers, Async_Writers)),
   Initializes    => Tasking_State
```

```
package Ada.Real_Time with
   SPARK_Mode,
   Abstract_State =>
     (Clock_Time with Synchronous,
        External => (Async_Readers, Async_Writers)),
   Initializes    => Clock_Time
```

- **API of these units refer to Tasking_State and Clock_Time**

Quiz

Is this correct? 1/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
procedure Rendezvous is
   task T1 is
      entry Start;
   end T1;

   task body T1 is
   begin
      accept Start;
   end T1;

begin
   T1.Start;
end Rendezvous;
```

Is this correct? 1/10

NO

```
procedure Rendezvous is
   task T1 is
      entry Start;
   end T1;

   task body T1 is
   begin
      accept Start;
   end T1;

begin
   T1.Start;
end Rendezvous;
```

Task rendezvous is not allowed.
violation of restriction
"Max_Task_Entries = 0"

Local task is not allowed.
violation of restriction
"No_Task_Hierarchy"

Is this correct? 2/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
protected P is
   entry Reset;
end P;

Data : Boolean := False;

protected body P is
   entry Reset when Data is
   begin
      null;
   end Reset;
end P;
```

Is this correct? 2/10

NO

```
protected P is
   entry Reset;
end P;

Data : Boolean := False;

protected body P is
   entry Reset when Data is
   begin
      null;
   end Reset;
end P;
```

Global data in entry guard is not allowed.
violation of restriction "Simple_Barriers" (for Ravenscar) or "Pure_Barriers" (for Extended Ravenscar)

Is this correct? 3/10

YES (click on the check icon)

NO (click on the error location(s))

```
protected P is
   procedure Set (Value : Integer);
end P;
task type TT;
T1, T2 : TT;
```

```
Data : Integer := 0;
protected body P is
   procedure Set (Value : Integer) is
   begin
      Data := Value;
   end Set;
end P;
task body TT is
   Local : Integer := 0;
begin
   loop
     Local := (Local + 1) mod 100;
      P.Set (Local);
   end loop;
end TT;
```

Is this correct? 3/10

NO

```
protected P is
   procedure Set (Value : Integer);
end P;
task type TT;
T1, T2 : TT;
```

```
Data : Integer := 0;
protected body P is
   procedure Set (Value : Integer) is
   begin
      Data := Value;
   end Set;
end P;
task body TT is
   Local : Integer := 0;
begin
   loop
      Local := (Local + 1) mod 100;
       P.Set (Local);
   end loop;
end TT;
```

Global unprotected data accessed in protected object shared between tasks

Is this correct? 4/10

YES (click on the check icon)

NO (click on the error location(s))

```
protected P is
   procedure Set (Value : Integer);
end P;
Data : Integer := 0 with Part_Of => P;
task type TT;
T1, T2 : TT;
```

```
protected body P is
   procedure Set (Value : Integer) is
   begin
      Data := Value;
   end Set;
end P;
task body TT is
   Local : Integer := 0;
begin
   loop
      Local := (Local + 1) mod 100;
      P.Set (Local);
   end loop;
end TT;
```

Is this correct? 4/10

YES

```
protected P is
   procedure Set (Value : Integer);
end P;
Data : Integer := 0 with Part_Of => P;
task type TT;
T1, T2 : TT;
```

```
protected body P is
   procedure Set (Value : Integer) is
   begin
      Data := Value;
   end Set;
end P;
task body TT is
   Local : Integer := 0;
begin
   loop
     Local := (Local + 1) mod 100;
      P.Set (Local);
   end loop;
end TT;
```

Data is part of the protected object state. The only accesses to Data are through P.

Is this correct? 5/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
protected P1 with Priority => 3 is
   procedure Set (Value : Integer);
end P1;

protected P2 with Priority => 2 is
   procedure Set (Value : Integer);
end P2;

task type TT with Priority => 1;
T1, T2 : TT;

protected body P2 is
   procedure Set (Value : Integer) is
   begin
      P1.Set (Value);
   end Set;
end P2;

task body TT is
begin
   loop
      P2.Set (Local);
   end loop;
end TT;
```

Is this correct? 5/10

YES

```
protected P1 with Priority => 3 is
   procedure Set (Value : Integer);
end P1;
```

```
protected P2 with Priority => 2 is
   procedure Set (Value : Integer);
end P2;
```

```
task type TT with Priority => 1;
T1, T2 : TT;
```

```
protected body P2 is
   procedure Set (Value : Integer) is
   begin
      P1.Set (Value);
   end Set;
end P2;
```

```
task body TT is
begin
   loop
      P2.Set (0);
   end loop;
end TT;
```

Ceiling_Priority policy is respected. Task never accesses a protected object with lower priority than its active priority.

Note that PO can call procedure or function from another PO, but not an entry (possibly blocking).

Is this correct? 6/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
protected type Mailbox is
   entry Publish;
   entry Retrieve;
private
   Not_Empty    : Boolean := True;
   Not_Full     : Boolean := False;
   Num_Messages : Natural := 0;
end Mailbox;
```

```
Max : constant := 100;
protected body Mailbox is
   entry Publish when Not_Full is
   begin
      Num_Messages := Num_Messages + 1;
      Not_Empty := True;
      if Num_Messages = Max then
         Not_Full := False;
      end if;
   end Publish;
   entry Retrieve when Not_Empty is …
end Mailbox;
```

Is this correct? 6/10

NO

```
protected type Mailbox is
   entry Publish;
   entry Retrieve;
private
   Not_Empty    : Boolean := True;
   Not_Full     : Boolean := False;
   Num_Messages : Natural := 0;
end Mailbox;
```

```
Max : constant := 100;
protected body Mailbox is
   entry Publish when Not_Full is
   begin
      Num_Messages := Num_Messages + 1;
      Not_Empty := True;
      if Num_Messages = Max then
         Not_Full := False;
      end if;
   end Publish;
   entry Retrieve when Not_Empty is …
end Mailbox;
```

integer range cannot be proved correct

Is this correct? 7/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
protected type Mailbox is
   entry Publish;
   entry Retrieve;
private
   Num_Messages : Natural := 0;
end Mailbox;
```

```
Max : constant := 100;
protected body Mailbox is
   entry Publish when Num_Messages < Max is
   begin
      Num_Messages := Num_Messages + 1;
   end Publish;

   entry Retrieve when Num_Messages > 0 is
   begin
      Num_Messages := Num_Messages - 1;
   end Retrieve;
end Mailbox;
```

Is this correct? 7/10

YES

```
protected type Mailbox is
   entry Publish;
   entry Retrieve;
private
   Num_Messages : Natural := 0;
end Mailbox;
```

```
Max : constant := 100;
protected body Mailbox is
   entry Publish when Num_Messages < Max is
   begin
      Num_Messages := Num_Messages + 1;
   end Publish;

   entry Retrieve when Num_Messages > 0 is
   begin
      Num_Messages := Num_Messages - 1;
   end Retrieve;
end Mailbox;
```

Precise range obtained from entry guards allows to prove checks.

Is this correct? 8/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
type Content is record
   Not_Empty … Not_Full … Num_Messages …
end record with Predicate =>
   Num_Messages in 0 .. Max
   and Not_Empty = (Num_Messages > 0)
   and Not_Full = (Num_Messages < Max);
protected type Mailbox is
   … C : Content;
end Mailbox;
```

```
protected body Mailbox is
   entry Publish when C.Not_Full is
      Not_Full     : Boolean := C.Not_Full;
      Num_Messages : Natural := C.Num_Messages;
   begin
      Num_Messages := Num_Messages + 1;
      if Num_Messages = Max then
         Not_Full := False;
      end if;
      C := (True, Not_Full, Num_Messages);
   end Publish;
```

Is this correct? 8/10

YES

```
type Content is record
   Not_Empty … Not_Full … Num_Messages …
end record with Predicate =>
   Num_Messages in 0 .. Max
   and Not_Empty = (Num_Messages > 0)
   and Not_Full = (Num_Messages < Max);
protected type Mailbox is
   … C : Content;
end Mailbox;
```

```
protected body Mailbox is
   entry Publish when C.Not_Full is
      Not_Full      : Boolean := C.Not_Full;
      Num_Messages : Natural := C.Num_Messages;
   begin
      Num_Messages := Num_Messages + 1;
      if Num_Messages = Max then
         Not_Full := False;
      end if;
      C := (True, Not_Full, Num_Messages);
   end Publish;
```

Precise range obtained from predicate allows to prove checks. Predicate is preserved.

Is this correct? 9/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
package Service with
  Abstract_State => (State with External)
is
   procedure Extract (Data : out Integer) with
     Global => (In_Out => State);
```

```
task type T;
T1, T2 : T;

task body T is
   X : Integer;
begin
   loop
      Extract (X);
   end loop;
end T;
```

Is this correct? 9/10

NO

```
package Service with
  Abstract_State => (State with External)
is
   procedure Extract (Data : out Integer) with
     Global => (In_Out => State);
```

```
task type T;
T1, T2 : T;

task body T is
   X : Integer;
begin
   loop
      Extract (X);
   end loop;
end T;
```

Unsynchronized state cannot be accessed from multiple tasks or protected objects.

Is this correct? 10/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
package Service with
  Abstract_State => (State with Synchronous, External)
is
   procedure Extract (Data : out Integer) with
     Global => (In_Out => State);
```

```
task type T;
T1, T2 : T;

task body T is
   X : Integer;
begin
   loop
      Extract (X);
   end loop;
end T;
```

Is this correct? 10/10

YES

```
package Service with
  Abstract_State => (State with Synchronous, External)
is
   procedure Extract (Data : out Integer) with
     Global => (In_Out => State);
```

```
task type T;
T1, T2 : T;

task body T is
   X : Integer;
begin
   loop
      Extract (X);
   end loop;
end T;
```

Abstract state is synchronized, hence can be accessed from multiple tasks and protected objects.

university.adacore.com