4.a Programming real-time systems (in Ada)

Where we see how program code may conform to the real-time systems theory, and then we build design and coding patterns that ensure conformance

From abstract to actual

- The real-time systems theory is of no use if no implementation technology can meet its requirements and assumptions
- We should like *precise pairing* between
  - The workload model *and* the static and dynamic components of the application program
    - What the program says it wants to do, and what it really does
  - The scheduling phenomena on paper *and* at run time
    - As assumed vs. as implemented

Means of enforcement

- Response-time analysis can be used to assert correct temporal behavior at *design time*
  - But that may not be robust if the specification claims err on the optimistic side
- Platform mechanisms can be used at *run time* to ensure that the application's temporal behavior stays within the asserted boundaries
  - We shall look at some example mechanisms
- Nicely complementary approaches, with very different costs
  - The cost of compile-time checking is paid once
  - The cost of run-time checking is paid during operation

Matching the workload model /1

- Static set of tasks
  - **Ada**: tasks declared at library level (the outermost scope), so that they have the longest lifetime
- Tasks issue jobs repeatedly
  - Duty cycle: activation, execution, wait-for-next-activation
    - To facilitate conformance, tasks should have a *single* source of activation (release event)
- Real-time attributes
  - Release time
    - Periodic: at every T time units
    - Sporadic: at least T between any two subsequent releases
  - Execution
    - Worst-case execution time C assumed to be known statically
  - Deadline: D time units after release

Matching the workload model /2

- Task interaction (collaboration)
  - Shared variables with mutually exclusive access
    - **Ada**: *protected objects* (PO) with procedures and functions
  - No avoidance synchronization: one's release must have a time bound
    - **Ada**: PO with a *single* entry, which fits sporadic tasks
- Scheduling model
  - Fixed-priority preemptive
    - **Ada**: FIFO within priorities
- Access protocol for shared objects
  - Ceiling priority protocol
    - **Ada**: Ceiling_Locking policy

Protected objects: exclusion synchronization

```
protected type Shared_Integer (Initial_Value : Integer) is
 function Read return Integer;
 procedure Write (Value : Integer);
private
 The_Integer : Integer := Initial_Value;
end Shared_Integer;
```

```
protected body Shared_Integer is
 function Read return Integer is
 begin
  return The_Integer;
 end Read;
 procedure Write (Value : Integer) is
 begin
  The_Integer := Value;
 end Write;
end Shared_Integer;
```

Concurrent Read

Mutually-exclusive Write

Protected objects: avoidance synchronization /1

```
Buffer_Size : constant Positive := 5;
type Index is mod Buffer_Size; -- modular type
subtype Count is Natural range 0 .. Buffer_Size;
type Buffer_T is array (Index) of Any_Type;

protected type Bounded_Buffer is
   entry Get (Item :     out Any_Type);
   entry Put (Item : in      Any_Type);
private
   First : Index := Index'First; -- 0
   Last  : Index := Index'Last;  -- 4
   In_Buffer : Count := 0;
   Buffer : Buffer_T;
end Bounded_Buffer;
```

- The PO interface exposes *entry* methods
  - Calls to PO entries are accepted only if given conditions hold within the functional logic of the PO

Protected objects: avoidance synchronization /2

```
protected body Bounded_Buffer is
   entry Get (Item :     out Any_Type)
      when In_Buffer > 0 is
   begin -- first read then move pointer
      Item := Buffer(First);
      First := First + 1; -- free from overflow
      In_Buffer := In_Buffer - 1;
   end Get;
   entry Put (Item : in      Any_Type)
      when In_Buffer < Buffer_Size is
   begin --  first move pointer then write
      Last := Last + 1; -- free from overflow
      Buffer(Last) := Item;
      In_Buffer := In_Buffer + 1;
   end Put;
end Bounded_Buffer;
```

- Any call on such entries is accepted *only* when the Boolean guard ("**when**" clause) is open
  - But this may not follow a predictable time pattern (no plausible worst-case)

Language profiling

- Any full language can hardly be restrictive
- Restricting needs language profiles
  - **Ada**: profile enforced by a *configuration* directive to the compiler

```
pragma Profile (Ravenscar);
```

  - Equivalent to a set of restrictions, plus three additional configuration pragmas

```
pragma Task_Dispatching_Policy (FIFO_Within_Priorities);
pragma Locking_Policy (Ceiling_Locking);
pragma Detect_Blocking;
```

- See "**Per approfondire 8**" on the static web page

Ravenscar profile restrictions

```
No_Abort_Statements,
No_Dynamic_Attachment,
No_Dynamic_Priorities,
No_Implicit_Heap_Allocations,
No_Local_Protected_Objects,
No_Local_Timing_Events,
No_Protected_Type_Allocators,
No_Relative_Delay,
No_Requeue_Statements,
No_Select_Statements,
No_Specific_Termination_Handlers,
No_Task_Allocators,
No_Task_Hierarchy,
No_Task_Termination,
Simple_Barriers,
Max_Entry_Queue_Length => 1,
Max_Protected_Entries => 1,
Max_Task_Entries => 0,
No_Dependence => Ada.Asynchronous_Task_Control,
No_Dependence => Ada.Calendar,
No_Dependence => Ada.Execution_Time.Group_Budget,
No_Dependence => Ada.Execution_Time.Timers,
No_Dependence => Ada.Task_Attributes
```


Restriction checking

- Almost all of the Ravenscar profile restrictions are checked at *compile time*
  - That's the key to a profile: the program's conformance is statically ascertained
- Those restrictions that would need full-program analysis to be decided, are left to run-time checking
  - Potentially blocking operations in PO bodies
  - Priority ceiling violation
  - Multiple call queued on a protected entry or suspension object
  - Task termination (real-time tasks do not terminate!)
- Let us consider these situations in more depth

Potentially blocking operations

- A call to a PO entry
- Delay until statement (explicit suspension)
- Transitively, call on a subprogram whose body contains a potentially blocking operation
- **Pragma Detect_Blocking** requires detection of potentially blocking operations
  - Exception **Program_Error** raised if detected
    - All bets are off …
  - Blocking need not be detected if it occurs in a call to a foreign language embedded in an Ada procedure

Other run-time checks

- Priority ceiling violation
- More than one call waiting on a protected entry or a suspension object
  - The release depends on the entry queuing policy and may incur priority inversion
  - **Program_Error** must be raised in any such case
- Task termination
  - Program behavior must be documented
  - Possible termination behaviors include
    - Silent termination
    - Holding the task in a pre-terminated state
    - Call of an application-defined termination handler defined with the **Ada.Task_Termination** package (C.7.3)

Other restrictions

- Some restrictions on the sequential part of the language may be useful in conjunction with the Ravenscar profile
  - **No_Dispatch**
  - **No_IO**
  - **No_Recursion**
  - **No_Unchecked_Access**
  - **No_Allocators**
  - **No_Local_Allocators**
- See ISO/IEC TR 15392, *Guide for the use of the Ada Programming Language in High Integrity Systems*

Outside of Ravenscar profile

- Real-time programming facilities of use when full static assurance is *not* possible
  - Execution-time measurement
  - Execution-time timers
  - Group budgets (for sporadic servers and other resource reservation policies)
  - Timing events
  - Additional dispatching policies

Execution-time measurement

- To monitor the CPU time consumed by individual tasks
- *Per-task* CPU clocks can be defined
  - Set at $t = 0$ before task activation
  - The clock value increases (notionally) as the task executes
    - Actual increments occur solely at dispatching points (sound) or at synchronous queries (silly)

Ada.Execution_Time

```
with Ada.Task_Identification;
with Ada.Real_Time; use Ada.Real_Time;
package Ada.Execution_Time is
   type CPU_Time is private;
   CPU_Time_First : constant CPU_Time;
   CPU_Time_Last  : constant CPU_Time;
   CPU_Time_Unit  : constant := implementation-defined-real-number;
   CPU_Tick       : constant Time_Span;
   function Clock
     (T : Ada.Task_Identification.Task_Id
          := Ada.Task_Identification.Current_Task)
     return CPU_Time;
   ...
end Ada.Execution_Time;
```


Execution-time timers

- To fire a user-defined event when a CPU clock reaches a specified value
  - A dedicated *handler* is invoked by the runtime when the corresponding event occurs
  - The handler is an (**access** to) a **protected procedure**
    - The educated equivalent of a callback
- Basic mechanism for execution-time monitoring

Ada.Execution_Time.Timers

```
with System;
package Ada.Execution_Time.Timers is
   type Timer (T : not null access constant
                   Ada.Task_Identification.Task_Id) is
      tagged limited private;
   type Timer_Handler is
      access protected procedure (TM : in out Timer);
   Min_Handler_Ceiling : constant System.Any_Priority
      := implementation-defined;
   procedure Set_Handler (TM      : in out Timer;
                          In_Time : in Time_Span;
                          Handler : in Timer_Handler);
   procedure Set_Handler (TM      : in out Timer;
                          At_Time : in CPU_Time;
                          Handler : in Timer_Handler);
   ...
end Ada.Execution_Time.Timers;
```


Group budgets

- To specify groups of tasks which share a global execution-time budget
  - Basic mechanism for server-based scheduling
    - As needed to serve aperiodic arrivals
  - Can be used to provide temporal isolation among task groups

Group budgets (spec)

```
with System;
package Ada.Execution_Time.Group_Budgets is
  type Group_Budget is tagged limited private;
  type Group_Budget_Handler is
    access protected procedure (GB : in out Group_Budget);
  ...
  Min_Handler_Ceiling : constant System.Any_Priority
    := implementation-defined;
  procedure Add_Task (GB : in out Group_Budget;
                      T  : in Ada.Task_Identification.Task_Id);
  ...
  procedure Replenish (GB : in out Group_Budget;
                       To : in Time_Span);
  procedure Add (GB       : in out Group_Budget;
                 Interval : in Time_Span);
  ...
  procedure Set_Handler (GB      : in out Group_Budget;
                         Handler : in Group_Budget_Handler);
  ...
end Ada.Execution_Time.Group_Budgets;
```


Timing events

- To execute user code at a specified time
  - A lightweight mechanism that does *not* require involving an application-level task
  - Similar in principle to interrupt handling
- The code is defined as an event handler
  - An (**access** to) a **protected procedure**
- Directly invoked by the runtime
  - Lowest possible latency

Ada.Real_Time.Timing events

```
package Ada.Real_Time.Timing_Events is
  type Timing_Event is tagged limited private;
  type Timing_Event_Handler is
    access protected procedure (Event : in out Timing_Event);
  procedure Set_Handler (Event   : in out Timing_Event;
                         At_Time : in Time;
                         Handler : in Timing_Event_Handler);
  ...
  procedure Cancel_Handler (Event     : in out Timing_Event;
                            Cancelled : out Boolean);
  ...
end Ada.Real_Time.Timing_Events;
```

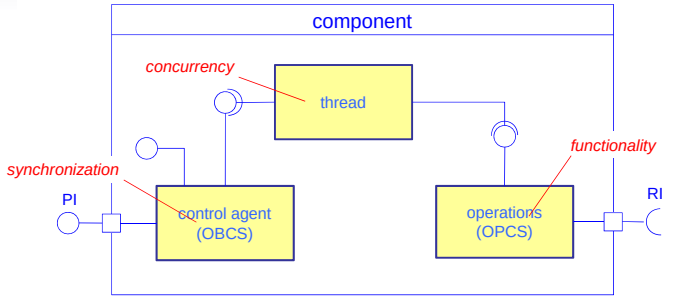

Dispatching policies

- A real-world real-time system may need other scheduling policies than just preemptive FPS
  - **Non preemptive**
    - With run-to-completion semantics in between explicit **yields**
  - **Round robin**
    - Within a specified band of priorities
    - Dispatch on quantum expiry is deferred until the end of protected action
  - **Earliest Deadline First**
    - Within a specified band of priorities
    - Relative and absolute “deadline”
    - EDF ordered ready queues
    - Resource locking used Deadline Floor policy

OOD for real-time systems

- Real-time components are design objects
  - Instances of classes
  - Hold abstract data types (internal state and operations on them), and expose interfaces
- Based on well-defined code patterns
  - Cyclic & sporadic tasks
  - Protected data
  - Passive data

Component structure



Component taxonomy

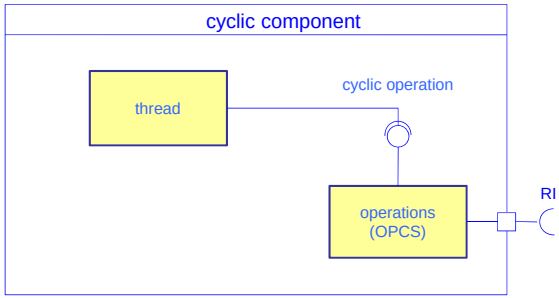
- Cyclic component
- Sporadic component
- Protected data component

- Under *inversion of control*
  - *What differentiates a framework from a library: the ability to enforce given design principles*

Cyclic component

- Clock-activated activity with fixed rate
- Attributes
  - Period
  - Deadline
  - Worst-case execution time
- The most basic cyclic code pattern does not need the synchronization agent
  - The system clock delivers the activation event
  - The component behavior is fixed and immutable

Cyclic component (basic)



Cyclic thread (spec)

```
task type Cyclic_Thread
   (Thread_Priority : Priority;
    Period          : Positive) is
  pragma Priority(Thread_Priority);
end Cyclic_Thread;
```

ms

Internally converted to `Time_Span` (the right type)

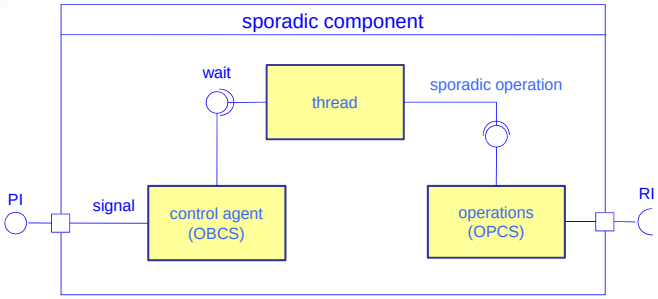
Cyclic thread (body)

```
task body Cyclic_Thread is
  Next_Time : Time := <Start_Time>; --  taken at elaboration time
                                    --+ higher up in the system
                                    --+ hierarchy
begin
  loop
    delay until Next_Time; -- so that all tasks start at T0
    OPCS.Cyclic_Operation; -- this is the job
                           --+ (this one, fixed and parameterless)
    Next_Time := Next_Time + Milliseconds(Period);
  end loop;
end Cyclic_Thread;
```


Sporadic component

- Activated by a software-activated or a non-clock interrupt event
  - Realized by signaling a waiting component
- Attributes
  - Minimum inter-arrival time
  - Deadline
  - Worst-case execution time
- The synchronization agent of the target component is used to signal the activation event
  - And to store-and-forward signal-related data (if any)

Sporadic component



Sporadic component (spec)

```
task type Sporadic_Thread(Thread_Priority : Priority) is
  pragma Priority(Thread_Priority);
end Sporadic_Thread;
```

```
protected type OBCS(Ceiling : Priority) is
  pragma Priority(Ceiling);
  procedure Signal;
  entry     Wait;
private
   Occurred : Boolean := False;
end OBCS;
```

A sporadic thread is activated by calling the `Signal` operation

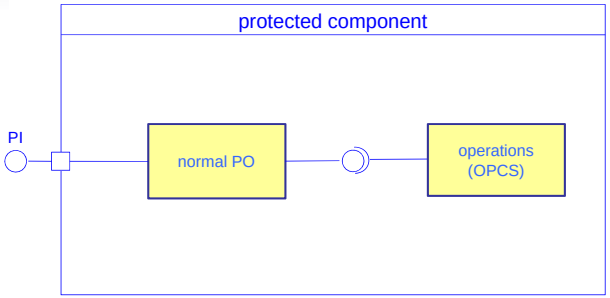
Sporadic thread (body)

```
task body Sporadic_Thread is
  Next_Time : Time := <Start_Time>;
begin
  delay until Next_Time; -- so that all tasks start at T0
  loop
    OBCS.Wait;
    OPCS.Sporadic_Operation;
    --  may take parameters if they were delivered by Signal
    --+ and retrieved by Wait
  end loop;
end Sporadic_Thread;
```


Sporadic control agent (body)

```
protected body OBCS is
  procedure Signal is
  begin
    Occurred := True;
  end Signal;
  entry Wait when Occurred is
  begin
     Occurred := False;
  end Wait;
end OBCS;
```

Protected component



Enforcing temporal properties

- The patterns we have seen just guarantee periodic or sporadic activation
- For stronger temporal guarantees at run time, such as
  - Minimum inter-arrival time for sporadic events
  - Deadline for all types of thread
  - WCET budgets for all types of thread
- Those patterns should be augmented

Minimum inter-arrival time /1

- Violations of the specified separation interval may cause increased interference on lower-priority tasks
- **Approach**: prevent sporadic thread from being activated *earlier* than stipulated
  - Compute earliest (absolute) allowable activation time
  - Withhold activation (if triggered) until that time

Sporadic thread with minimum separation (spec)

```
task type Sporadic_Thread
   (Thread_Priority : Priority;
    Separation      : Positive) is
  pragma Priority(Thread_Priority);
end Sporadic_Thread;
```

ms

Minimum inter-arrival time, internally converted to ms

Sporadic thread (body)

```
task body Sporadic_Thread is
  Release_Time : Time;
  Next_Release : Time := <Start_Time>;
begin
  loop
    delay until Next_Release;
    OBCS.Wait;
    Release_Time := Clock;
    OPCS.Sporadic_Operation;
    Next_Release := Release_Time + Milliseconds(Separation);
  end loop;
end Sporadic_Thread;
```

Still a single (logical) point of activation, but with nasty runtime overhead

Critique

- May incur temporal drift as the clock is read *after* task release
  - Preemption may hit just *after* the release but *before* reading the clock
  - Separation may become *larger* than required
- Better to read the clock at the place and time the task is released
  - Within the synchronization agent, which is protected hence less exposed to general interference

Minimum inter-arrival time /2

```
task body Sporadic_Thread is
  Release_Time : Time;
  Next_Release : Time := <Start_Time>;
begin
  loop
    delay until Next_Release;
    OBCS.Wait(Release_Time);
    OPCS.Sporadic_Operation;
    Next_Release := Release_Time + Milliseconds(Separation);
  end loop;
end Sporadic_Thread;
```


Recording release time /1

```
protected type OBCS(Ceiling : Priority) is
  pragma Priority(Ceiling);
  procedure Signal;
  entry Wait(Release_Time : out Time);
private
   Occurred : Boolean := False;
end OBCS;
```

Recording release time /2

```
protected body OBCS is
  procedure Signal is
  begin
    Occurred := True;
  end Signal;

  entry Wait(Release_Time : out Time) when Occurred is
  begin
    Release_Time := Clock;
    Occurred := False;
  end Wait;
end OBCS;
```


Deadline miss

- Two possible causes for it
  - Higher-priority tasks executing more often than expected
    - Can be prevented with inter-arrival time enforcement
  - Overruns in the same or higher-priority tasks
    - Programming error in the functional code
    - Inaccurate WCET analysis

Deadline-miss detection

- Can be done with the help of **timing events**
  - A mechanism for requiring some application-level action to be executed at a given time
- Timing events are statically allocated
  - Under the Ravenscar Profile, they can only exist at the library level
    - So that they never risk going out of scope!
- Minor optimization possible for periodic tasks
  - Which however breaks the symmetry of code patterns

Cyclic thread with deadline miss detection (spec)

```
task type Cyclic_Thread
   (Thread_Priority : Priority;
    Period          : Positive;
    Deadline        : Positive) is
  pragma Priority(Thread_Priority);
end Cyclic_Thread;
```

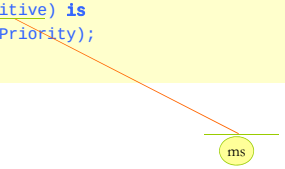
ms

Thread body

```
Deadline_Overrun : Timing_Event; -- static, local per component
task body Cyclic_Thread is
  Next_Time : Time := <Start_Time>;
  Canceled  : Boolean := False;
begin
  loop
    delay until Next_Time;
    Set_Handler(Deadline_Overrun,
                Next_Time + Milliseconds(Deadline),
                Deadline_Overrun_Handler); -- application-specific
    OPCS.Cyclic_Operation;
    Cancel_Handler(Deadline_Overrun, Canceled);
    Next_Time := Next_Time + Milliseconds(Period);
  end loop;
end Cyclic_Thread;
```


Sporadic thread with deadline miss detection (spec)

```
task type Sporadic_Thread
   (Thread_Priority : Priority;
    Separation      : Positive;
    Deadline        : Positive) is
  pragma Priority(Thread_Priority);
end Sporadic_Thread;
```



Thread body

```
Deadline_Overrun : Timing_Event; -- static, local per component
task body Sporadic_Thread is
  Release_Time    : Time;
  Next_Release    : Time := <Start_Time>;
  Canceled        : Boolean := False;
begin
  loop
    delay until Next_Release;
    OBCS.Wait(Release_Time);
    Set_Handler(Deadline_Overrun,
                Release_Time + Milliseconds(Deadline),
                Deadline_Overrun_Handler); -- application-specific
    OPCS.Sporadic_Operation;
    Cancel_Handler(Deadline_Overrun, Canceled);
    Next_Release := Release_Time + Milliseconds(Separation);
  end loop;
end Sporadic_Thread;
```


Execution-time overruns

- Tasks may execute for longer than stipulated, owing to
  - Programming errors in the functional code
  - Inaccurate WCET values used in feasibility analysis
    - Optimistic instead of conservative
- WCET overruns can be detected at run time with the help of **execution-time timers**
  - Not included in the Ravenscar profile because their implementation is costly
  - Included in extended profiles

Cyclic thread with WCET overrun detection (spec)

```
task type Cyclic_Thread
   (Thread_Priority : Priority;
    Period          : Positive;
    WCET_Budget     : Positive) is
  pragma Priority(Thread_Priority);
end Cyclic_Thread;
```

Thread body

```
task body Cyclic_Thread is
  Next_Time : Time := <Start_Time>;
  Id : aliased constant Task_ID := Current_Task;
  WCET_Timer : Timer(Id'access);
begin
  loop
    delay until Next_Time;
    Set_Handler(WCET_Timer,
                Milliseconds(WCET_Budget),
                WCET_Overrun_Handler); -- application-specific
    OPCS.Cyclic_Operation;
    Next_Time := Next_Time + Milliseconds(Period);
  end loop;
end Cyclic_Thread;
```


Observation

- WCET overruns in sporadic tasks can be detected similarly
- The timer should be set after release
- The timer is implicitly canceled when set again

Summary

- We have seen how one particular programming language is able to capture all design and execution aspects that descend from the real-time systems theory that we seen so far
- We have seen how design and code patterns could be used to make sure that the application program conforms with the required semantics

Selected readings

- Tullio Vardanega, Juan Zamorano, Juan Antonio de la Puente (**2005**), *On the Dynamic Semantics and the Timing Behavior of Ravenscar Kernels*
  DOI: 10.1023/B:TIME.0000048937.17571.2b