Ada 2005 code patterns for provable real-time programming

Anno accademico 2012/13
Laurea magistrale in informatica
Università di Padova
Tullio Vardanega

Workload model – 1

- Static set of tasks
  - **Ada**: tasks declared at library level (outermost scope)
- Tasks issue jobs repeatedly
  - Task cycle: activation, execution, suspension
    - Single activation source per task
- Real-time attributes
  - Release time
    - Periodic: at every T time units
    - Sporadic: at least T time units between any two subsequent releases
  - Execution
    - Worst case execution time (WCET) assumed to be known
    - Deadline: D time units after release

Workload model – 2

- Task communication
  - Shared variables with mutually exclusive access
    - **Ada**: protected objects (PO) with procedures and functions
  - No conditional synchronization
    - Other than for sporadic task activation
    - **Ada**: PO with a single entry
- Scheduling model
  - Fixed-priority pre-emptive
    - **Ada**: FIFO within priorities
- Access protocol for shared objects
  - Ceiling priority protocol (base version)
    - **Ada**: Ceiling_Locking policy

Language profile

- Enforced by means of a configuration pragma

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

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

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

Ravenscar 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 restrictions can be checked at compile time
- A few can only be checked at run time
  - Potentially blocking operations in protected operation bodies
  - Priority ceiling violation
  - More than one call queued on a protected entry or a suspension object
  - Task termination

Potentially blocking operations

- Protected entry call statement
- Delay until statement
- Call on a subprogram whose body contains a potentially blocking operation
- **Pragma Detect_Blocking** requires detection of potentially blocking operations
  - Exception **Program_Error** must be raised if detected at run-time
  - Blocking need not be detected if it occurs in the domain of a foreign language (e.g. C)

Other run-time checks

- Priority ceiling violation
- More than one call waiting on a protected entry or a suspension object
  - **Program_Error** must be raised in both cases
- 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 15942, *Guide for the use of the Ada Programming Language in High Integrity Systems*, for details

Execution-time measurement

- The CPU time consumed by tasks can be monitored
- Per-task CPU clocks can be defined
  – Set at 0 before task activation
  – The clock value increases as the task executes

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

- A user-defined event can be fired when a CPU clock reaches a specified value
  – An event handler is automatically invoked by the runtime
  – The handler is an (**access** to) a **protected procedure**
- 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

- Groups of tasks with a global execution-time budget can be defined
  - Basic mechanism for server-based scheduling
  - Can be used to provide temporal isolation among groups of tasks

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

- Lightweight mechanism for defining code to be executed at a specified time
  - Does not require an application-level task
  - Analogous to interrupt handling
- The code is defined as an event handler
  - An (**access** to) a **protected procedure**
- Directly invoked by the runtime

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

- Additional dispatching policies
  - **Non preemptive** (explicit yield)
    - Run-to-completion semantics (per partition)
  - **Round robin**
    - Within specified priority band
    - Dispatch on quantum expiry deferred until end of protected action
  - **Earliest Deadline First**
    - Within specified priority band
    - Relative and absolute “deadline”
    - EDF ordered ready queues
    - Guaranteed form of resource locking (preemption level + deadline)

Priority-band dispatching

- Mixed policies can coexist within a single partition
  - Priority specific dispatching policy can be set by configuration
  - Protected objects can be used for tasks to communicate across different policies
  - Tasks do not move across bands

An object-oriented approach

- Real-time components are objects
  - Instances of predefined classes
  - Internal state + interfaces
- Based on well-defined code patterns
  - Cyclic & sporadic tasks
  - Protected data
  - Passive data

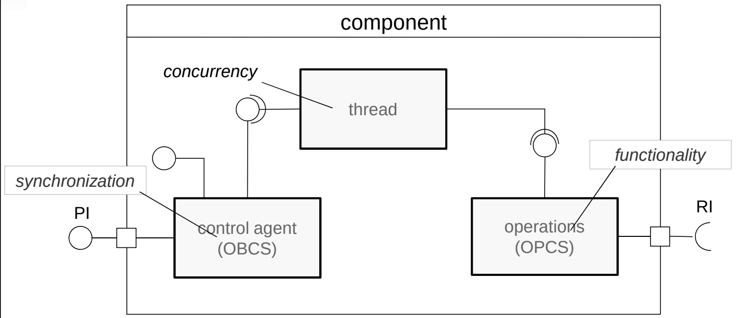
Enforce intentions

- Static WCET analysis and response-time analysis can be used to assert correct temporal behavior at *design time*
- Platform mechanisms can be used at *run time* to ensure that temporal behavior stays within the asserted boundaries
  – Clocks, timers, timing events, …
- Conveniently complementary approaches

Run-time services

- The execution environment must be capable of preserving properties asserted at model level
  – Real-time clocks & timers
  – Execution-time clocks & timers
  – Predictable scheduling
- We assume an execution environment implementing the Ravenscar model
  – Ada 2005 with the Ravenscar profile
  – Augmented with (restricted) execution-time timers

Component structure



Component taxonomy

- Cyclic component
- Sporadic component
- Protected data component
- Passive component

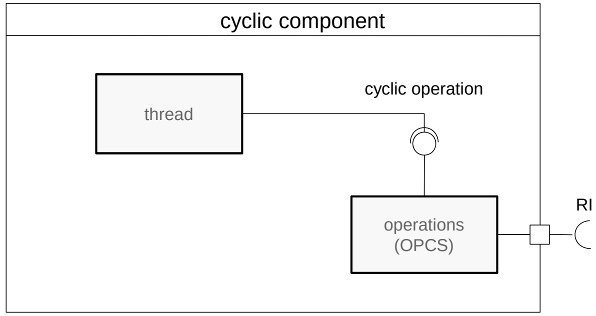
- Under *inversion of control*

6

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

cannot be `Time_Span`!

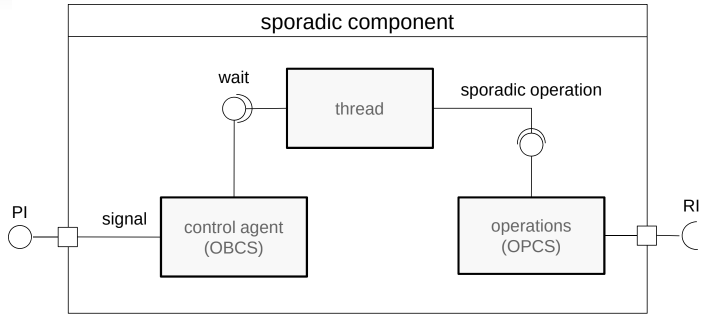
Cyclic thread (body)

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

Sporadic component

- Activated by a software-mediated event
  - Signaled by software or hardware interrupts
- 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;
```


Other components

- **Protected component**
  - No thread, only synchronization and operations
  - Straightforward direct implementation with protected object
- **Passive component**
  - Purely functional behavior, neither thread nor synchronization
  - Straightforward direct implementation with functional package

Temporal properties

- Basic patterns only guarantee periodic or sporadic activation
- They can be augmented to guarantee additional temporal properties at run time
  - Minimum inter-arrival time for sporadic events
  - Deadline for all types of thread
  - WCET budgets for all types of thread

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 expressed in 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 point of activation

Critique

- May incur some 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 and thus 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

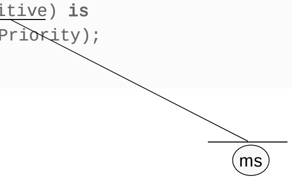
- May result from
  - Higher priority tasks executing more often than expected
    - Can be prevented with inter-arrival time enforcement
  - Overruns in the same or higher priority tasks
    - Programming error in the functional code
    - Inaccurate WCET analysis

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
  - Under the Ravenscar Profile timing events can only exist at library level
- Timing events are statically allocated
- 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;
```



Thread body

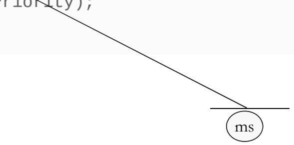
```
Deadline_Overrun : Timing_Event; -- static, local per component
task body Cyclic_Thread is
  Next_Time : Time := <Start_Time>;
  Canceled  : Boolean := False;
begin
  loop
    delay until Next_Time;
    Set_Handler(Deadline_Overrun,
                Next_Time + Milliseconds(Deadline),
           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


Thread body (streamlined)

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

Can't streamline as the deadline cannot be computed until returning from Wait

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 vs. pessimistic
- WCET overruns can be detected at run time with the help of **execution-time timers**
  - Not included in Ravenscar
  - Extended profile

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;
```

ms

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;
```

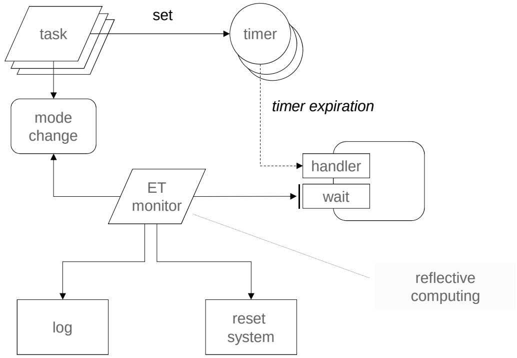
Observation

- WCET overruns in sporadic tasks can be detected similarly
- The timer should be set after the activation
- No need for timer cancellation

Fault handling strategies

- Error logging
  – Only for low-criticality tasks
- Second chance
  – Use slack time and try to complete
- Mode change
  – Switch to safe mode
    - For fail safe or fail soft behavior

Fault handling scheme



Multiple jobs per task

- Cyclic and sporadic objects may have **modifier** operations
  – For mode change, or occasional behavior modifications
- Asynchronous Transfer of Control are not allowed in Ravenscar
  – Modifier requests are queued in the OBCS
    - OBCS needed for cyclic components as well

Cyclic thread with modifier

```
task body Cyclic_Thread is
  Next_Release_Time : Time := <Start_Time>;
  Request : Request_Type;
begin
  loop
    delay until Next_Release_Time;
    OBCS.Get_Request(Request); -- may include operation parameters
    case Request is
      when NO_REQ  => OPCS.Periodic_Activity;
      when ATC_REQ => -- may take parameters
                      OPCS.Modifier_Operation;
    end case;
    Next_Release_Time := Next_Release_Time + Period;
  end loop;
end Cyclic_Thread;
```


Synchronization agent – 1

```
-- for cyclic thread
protected type OBCS (Ceiling: Priority) is
  pragma Priority(Ceiling);
  procedure Put_Request(Request : Request_Type);
  procedure Get_Request(out Request : Request_Type);
private
  Buffer : Request_Buffer; -- bounded queue
end OBCS;
```


Synchronization agent – 2

```
-- for cyclic thread
protected body OBCS(Ceiling : Priority) is
  procedure Put_Request(Request : Request_Type) is
  begin
    Buffer.Put(Request);
  end Put_Request;
  procedure Get_Request(out Request : Request_Type) is
  begin
    if Buffer.Empty then
      Request := NO_REQ;
    else
      Buffer.Get(Request);
    end if;
  end Get_Request;
end OBCS;
```