AdaCore
The GNAT Pro Company

Ada 2005 for Real-Time, Embedded and High-Integrity Systems

Jose F. Ruiz <ruiz@adacore.com>
Senior Software Engineer

Ada Deutschland Software Workshop 2008
Karlsruhe, 24th January, 2008

Outline of the presentation

- **Ada**
  - For embedded high-integrity real-time systems

- **Ada 2005**
  - The Ravenscar tasking profile
  - Flexible real-time scheduling algorithms
  - CPU clocks and timers
  - Timing events
  - Flexible object-oriented features

Ada for High-Integrity Applications

- **Ada promotes safety / reliability**
  - Readability
  - Compile-time checking (strong typing)
  - Encapsulation and data abstraction
  - Deterministic language semantics (ISO Standard)
    - Implementation must document effect where language semantics offers flexibility

- **Support modern software engineering techniques**
  - High abstraction level constructions integrated within the language
    - tasking, OOP, templates, modularity, data abstraction and encapsulation, …
  - General design philosophy promotes sound software engineering

- **Specific features**
  - Real-Time and High-Integrity Annexes
  - Language subsets

- **Guidelines documents**
  - *Guide for Ada in High-Integrity Systems* (an ISO Technical Report)
  - *Guide for Ada Ravenscar Profile in High-Integrity Systems*

Ada for Real-Time Systems

- **Concurrency**
  - Within the language
    - Avoid error-prone low-level constructions
  - Well-defined semantics for scheduling
  - Safe / efficient mutual exclusion
    - Avoidance of unbounded priority inversion
  - Ravenscar profile
    - Restricted set of tasking features amenable to schedulability analysis and certification

- **Asynchrony**
  - Asynchronous events / event handlers
    - Connection with interrupts
  - Asynchronous Transfer of Control
    - Timeout
    - Task termination
  - Preemptive task abortion
  - Asynchronous task control

- **Time**
  - Support for high-resolution monotonic clock and absolute and relative delays

Ada for Embedded Systems

- **Specific Annex for low-level support**
  - Access to hardware-specific features
- **Access to machine operations**
  - Assembly and intrinsic subprograms
- **Representation support**
  - Address, alignment, size, layout
- **Shared variable control**
  - Atomic, volatile,…
- **Storage management**
  - Specific storage pools
    - User-defined managers that may be placed in specific memory regions, and that may be suitable for real-time systems because they can be made predictable

What is N
ew in Ada 2005?

- **The Ravenscar profile**

- **Task elaboration and finalization**
  - Partition elaboration policy for high-integrity systems (atomic elaboration)
  - Task termination procedures

- **Restriction pragmas**
  - *No_Relative_Delay*, *Max_Entry_Queue_Length*,…

- **Time and clocks**
  - Timing events
  - Execution time clocks
  - Execution time budgets
    - for task groups also

- **Scheduling**
  - New dispatching policies
    - Non-preemptive, round robin, Earliest Deadline First (EDF)
  - Dynamic ceiling priorities
  - Priority Specific dispatching

- **Object-Oriented Programming**
  - Interfaces
  - Object notation

The Ravenscar Profile

- **A subset of the Ada tasking model**

- **Defined to meet safety-critical real-time requirements**
  - Determinism
  - Schedulability analysis
  - Memory-boundedness
  - Execution efficiency and small footprint
  - Suitability for certification

- **State-of-the-art concurrency constructs**
  - Adequate for most types of real-time software

AdaCore
The GNAT Pro Company

The Ravenscar Profile (II)

- **Set of tasks / interrupts to be analyzed is fixed and has static properties**
  - Tasks, protected objects only at library level
    - No dynamic allocation of tasks or protected objects
  - Each task is infinite loop
    - single “triggering” action (delay or event)

- **Memory usage is deterministic**
  - Tasks descriptors and stacks are statically created at compile time
    - No implicit heap usage

- **Program execution is deterministic**
  - Simple protected objects
    - at most one entry, at most one caller queued
  - Task creation and activation is very simple and deterministic
    - Tasks created at initialization, then activated and executed according to their priority

The Ravenscar Tasking Model

- **A single processor**
- **A fixed number of tasks**
- **Single invocation event per task**
  - Time-triggered or event-triggered
- **Task interaction using shared data**
  - Mutual exclusive access
- **Remove constructions difficult to analyse**
  - No asynchronous control, no abort, …

Most violations detected at compile time

The Ravenscar Tasking Model (II)

- **Scheduling policy**
  - Preemptive fixed-priorities
- **Locking policy**
  - Ceiling priority for bounding priority inversion
- **Remove non-deterministic constructions**
  - No relative delays, no task termination, no abort, ...

Supports sound real-time development techniques, such as Rate Monotonic Analysis and Response Time Analysis

Example: Cyclic tasks

```
task body Cyclic is
   Period : constant Time_Span := Seconds (1);
   Next_Activation : Time := Clock;
begin
   loop
      delay until Next_Activation;
      --  Do something
      Next_Activation := Next_Activation + Period;
   end loop;
end Cyclic;
```

```
task body Cyclic_With_Deadline is
   Period : constant Time_Span := Seconds (1);
   Next_Activation : Time := Clock;
begin
   loop
      delay until Next_Activation;
      Next_Activation := Next_Activation + Period;
      select
         delay until Next_Activation;
         --  Notify missed deadline
      then abort
         --  Do something
      end select;
   end loop;
end Cyclic_With_Deadline;
```

Real-Time Scheduling

- **Ada 95 provides**
  - Complete and well defined set of language primitives for Fixed Priority Scheduling

- **Ada 2005 allows new schemes**
  - Non-preemptive
  - Round Robin
  - Earliest Deadline First (EDF)
  - Mixed policies within a partition

Timing Events

- **A means of defining code that is executed at a future point in time**
  - Efficient stand-alone timer
- **Does not need a task**
  - Executed directly in the context of the interrupt handler
  - Reduce the number of
    - tasks in a program
    - Context switches
- **Similar in notion to interrupt handling**
  - Time itself generates the interrupt
- **Useful for**
  - Short time-triggered procedures
  - Imprecise computation

Example: Task deadlines with timing events

```ada
protected Watchdog is
   pragma Interrupt_Priority (Interrupt_Priority'Last);
   procedure Timeout (Event : in out Timing_Event);
   entry Is_OK;
private
   Panic : Boolean := False;
end Watchdog;

protected body Watchdog is
   procedure Timeout (Event : in out Timing_Event) is
   begin
      --  Alarm !!!
      Panic := True;
   end Timeout;
   entry Is_OK when Panic is
   begin
      --  Panic mode activated
      Panic := False;
   end Is_OK;
end Watchdog;
```

```ada
task body Cyclic_With_Deadline is
   Period : constant Time_Span := Seconds (1);
   Next_Activation : Time := Clock;
   Deadline_Event : Timing_Event;
   Alarm_Cancelled : Boolean;
begin
   loop
      delay until Next_Activation;
      Next_Activation := Next_Activation + Period;
      Set_Handler
         (Event     => Deadline_Event,
          At_Time  => Next_Activation,
          Handler  => Watchdog.Timeout'Access);
      select
         Watchdog.Is_OK;
         --  Notify missed deadline
      then abort
         --  Do something
         --  Notify end of computation
         Cancel_Handler
            (Deadline_Event, Alarm_Cancelled);
      end select;
   end loop;
end Cyclic_With_Deadline;
```

Execution Time Support

- **Monitor and control task execution time**
  - Every task has an execution time clock
  - Fire an event when a task execution time reaches a specified value
  - Useful in high-integrity (fault tolerant) applications for detecting
    - Wrong WCET estimations
    - Software errors

- **Allocate and support bugdets for groups of tasks**
  - Useful for some scheduling policies, such as those for aperiodic servers

Example: Iterative computation

```
task body Iterative_Task is
  Stop_Time : CPU_Time :=
    Ada.Execution_Time.Clock + Milliseconds (10);
begin
  while
    Ada.Execution_Time.Clock < Stop_Time
  loop
    --  Do something
  end loop;
end Iterative_Task;
```

```
task body Iterative_Task_2 is
  ID : aliased Task_ID := Current_Task;
  Budget_Manager : Timer (ID'Access);
begin
  Set_Handler
    (Budget_Manager, Milliseconds (10),
     Overrun.Timeout'Access);
  select
    Overrun.Stop_Task;
  then abort
    loop
      --  Do something
    end loop;
  end select;
end Iterative_Task_2;
```

```
protected Overrun is
  entry Stop_Task;
  procedure Timeout (TM : in out Timer);
private
  Budget_Overrun : Boolean := False;
end Overrun;

protected body Overrun is
  entry Stop_Task when Budget_Overrun is
  begin
    --  Budget overrun
    Budget_Overrun := False;
  end Stop_Task;
  procedure Timeout (TM : in out Timer) is
  begin
    --  Stop computation
    Budget_Overrun := True;
  end Timeout;
end Overrun;
```

Example: Budgets

```
protected Overrun is
   entry Stop_Task;
   procedure Handler (TM : in out Timer);
private
   Budget_Overrun : Boolean := False;
end Overrun;

protected body Overrun is
   entry Stop_Task when Budget_Overrun is
   begin
      --  Budget overrun
      Budget_Overrun := False;
   end Stop_Task;
   procedure Handler (TM : in out Timer) is
   begin
      --  We have a problem
      Budget_Overrun := True;
   end Handler;
end Overrun;
```

```
task body Cyclic_With_Budget is
   Period : constant Time_Span := Seconds (1);
   Next_Activation : Time := Clock;
   ID : aliased Task_ID := Current_Task;
   Budget_Manager : Timer (ID'Access);
   Alarm_Cancelled : Boolean;
begin
   loop
      delay until Next_Activation;
      Next_Activation := Next_Activation + Period;
      Set_Handler
         (TM       => Budget_Manager,
          At_Time => Next_Activation,
          Handler  => Overrun.Handler'Access);
      select
         Overrun.Stop_Task;
         --  Notify missed deadline
      then abort
         --  Do something
         --  Notify end of computation
         Cancel_Handler
            (Budget_Manager, Alarm_Cancelled);
      end select;
   end loop;
end Cyclic_With_Budget;
```

Safe Object Oriented Programming

- **Type extension and inheritance**
  - Powerful
    - Cover most object-oriented design methods
      - Code reuse, programming by extension, etc.
  - Fine for safety-critical systems
- **Dynamic dispatching**
  - Actual flow of control not known statically
  - Worrisome for safety-critical system
- **Controlling dynamic dispatching**
  - Avoid class-wide types
    - In Ada, methods are statically bound by default
  - Enforced by a language-defined restriction (*No_Dispatch*)
  - Each operation declare explicitly whether it is intended to inherit

AdaCore
The GNAT Pro Company

Abstract interfaces

- **Limited form of multiple inheritance**
  - Java-like

Multiple inheritance of specifications, and single inheritance of implementation

- **Extends the Java model**
  - Protected, task, and synchronized interfaces
    - abstraction that can be implemented either with an active task or with a passive monitor
    - Seamless integration between OO and multi-tasking features
- **Much of the power of multiple inheritance**
  - Without most of the implementation and semantic difficulties

Example: Interface

```
type Person is interface;
function Name (This : Person) return Name_Type is abstract;
function Gender (This : Person) return Gender_Type is abstract;

type Worker is interface;
function Name (This : Worker) return Name_Type is abstract;
function Salary (This : Worker) return Natural is abstract;

type Employee is new Person and Worker with
  record
    Name : Name_Type;
    Sex   : Gender_Type;
    Wage : Natural;
  end record;

function Name (This : Employee) return Name_Type;
function Gender (This : Employee) return Gender_Type;
function Salary (This : Employee) return Natural;
```

Example: Synchronized interface

```
type Processing_Entity is task interface;
procedure Replicate (This : Processing_Entity) is abstract;

type Buffer is synchronized interface;
procedure Put (This : in out Buffer; Item : Element) is abstract;
procedure Get (This : in out Buffer; Item : out Element) is abstract;

task type Server_Buffer is new Processing_Entity and Buffer with
   entry Replicate;
   entry Put (Item : Element);
   entry Get (Item : out Element);
end Server_Buffer;
```

Conclusions

- **Increasing need for safe programming**
  - Ada has an impressive track record in avionics, train control, other safety-critical domains
  - Ada is being considered in new domains

- **Ada 2005 addresses the needs of the real-time and high-integrity communities**
  - Expressive, even in safety-critical subsets
    - Safe tasking
    - Safe OOP
  - Flexible
    - New scheduling policies, new capabilities and features
  - High-level abstractions, but ...
    - Deterministic
    - Time analyzable