Advanced Ada Support for Real-Time Programming *

Mario Aldea Rivas
Universidad de Cantabria, 39005 Santander, Spain; email: aldeam@unican.es

Abstract

This paper is an extended summary of the tutorial given at Ada-Europe 2012.

In the 2005 and 2012 revisions of the Ada standard, real-time programming has experienced a large improvement but most of the new services introduced are unknown or underused due to the lack of free software implementations. The tutorial presented an overview of these new services trying to focus on their utility for real-time systems and their typical use patterns.

Keywords: Ada 2005, Ada 2012, real-time systems, programming languages.

1 Introduction

The support of Ada for real-time programming has experienced a large improvement in the last years positioning the language one step ahead of other real-time languages and operating system interfaces. Functionalities such as hierarchical scheduling based on priority ranges, new dispatching policies, execution time clocks and timers, timing events, etc. are in the standard from the 2005 revision [1] [2] but they have not gotten the relevance that they deserve due to the lack of free software implementations.

Most of these relatively new services are starting to be available in some platforms. In particular, in this tutorial we used the MaRTE OS/GNAT [3] platform. This platform supports most of these new services [4] [5]:

- Timing events.
- Execution time clocks and timers.
- Task group execution time budgets.
- Dynamic ceiling priorities for protected objects.
- Additional scheduling policies: Round robin, EDF, Mixed (priority-specific policies).
- Immediate priority changes.
- Execution Time for Interrupt Handlers.

*This work has been funded in part by the Spanish Government and FEDER funds under grant number TIN2011-28567-C03-02 (HI-PARTES).

The first objective of this tutorial was to provide an overview of the real-time "classic" model established in Ada 95, and to show how this classic model has been reinforced with extensions defined in Ada 2005 and Ada 2012 [6] [7].

The second objective was to perform an intensive review of the new real-time services added in Ada 2005 and Ada 2012 trying to describe the utility of each service along with examples and use patterns[1].

1.1 Evolution of the real-time Ada

Table 1 shows the evolution of the real-time services included in the Ada language. The core of the "classic" real-time concurrency model based in preemptive fixed priorities was established in Ada 95, having the tasks and the protected objects as its most relevant elements.

A very important group of real-time facilities was added to the standard in the revision of the year 2005. The most relevant additions were related to time management (timing events, execution time clocks and timers and group budgets), to dispatching (new dispatching policies and priority specific dispatching) and the Ravenscar profile.

The number and the importance of the new real-time services added in Ada 2012 is not as impressive as in Ada 2005 but it is also quite important. Among all the new services added in this revision of the standard it deserves a special mention the support for multiprocessor architectures.

2 Classic Ada real-time model

In the Ada 95 real-time model the *task* is the concurrency unit and the synchronization and mutual exclusion among tasks is accomplished by the use of *protected objects* or *rendezvous*. Analysable scheduling is achieved with the use of the *FIFO_Within_Priorities* dispatching policy and the locking policy *Ceiling_Locking*.

Time management, and in particular the periodic activation of tasks, is performed with the *monotonic clock* provided by package `Ada.Real_Time` and the `delay until` statement. Other services related to real-time in Ada 95 are the *dynamic priorities for tasks* and the *interrupt handling* facilities among others.

The classic model allowed to dynamically change the priority of the tasks, but there was not a similar functionality to

[1] Most of the examples used in the tutorial are not included in this summary due to the lack of space.

Ada 95	Ada 2005 additions	Ada 2012 additions
Tasks	Ravenscar profile	Multiprocessor and Dispatching domains
FIFO_Within_Priorities	Timing events	Group budgets and multiprocessors
Dynamic priorities	Execution time clocks and timers	Barriers
Protected objects	Group budgets	Suspension objects
Ceiling_Locking	Non-preemtive dispatching	"Synchronization" aspect
Monotonic clock	EDF dispatching	Yield_To_Higher
delay until	Round robin dispatching	Execution time of interrupt handlers
	Priority specific dispatching	Suspend_Until_True_And_Set_Deadline
	Dynamic priorities for POs	
	Synchronized interfaces	
	Task termination	
	Partition elaboration policy	

Table 1: Evolution of the real-time Ada

Listing 1: Use of the `Priority` attribute

```
protected body PO is
  procedure Change_Ceiling (Prio: in System.Priority) is
  begin
    ... -- PO'Priority has old value here
    PO'Priority := Prio;
    ... -- PO'Priority has new value here
  end Change_Ceiling; -- ceiling is changed here
  ...
end PO;
```

change the ceiling of the protected objects. This was a serious limitation in many applications, specially in those that require "mode changes". This problem was solved in Ada 2005 with the definition of the attribute `Priority` for the protected objects (see Ada 2012 Reference Manual[2] D.5.2). An example of use of this attribute is shown in Listing 1. Note the unusual syntax: it is the only attribute in the language that is possible to assign a value to.

Other addition to the classic model is the introduction in Ada 2005 of the `Detect_Blocking` pragma (RM H.5). Its use in a partition forces to detect potentially blocking operations within any protected action.

The introduction of the "aspects" in Ada 2012 has had an important impact everywhere in the language. In relation to the real-time concurrency model the most important effect is the change in the assignment of the priority to tasks and protected objects. Now aspects should be used for this purpose instead of the old pragmas (that are declared obsolescent):

```
task Controller with Priority => 12;

protected PO with Priority => 20 is
  ...
end PO;
```

Other very relevant addition in Ada 2005 was the Ravenscar Profile (RM D.13). A profile is a collection of restrictions and other pragmas that describes a subset of the language intended for a particular purpose. The Ravenscar Profile is a subset of the Ada tasking model targeted to critical real-time applications. The main objectives of this profile are:

- Produce a deterministic concurrent execution model that can be analysable.
- Allow an efficient and small implementation of the run-time library.

Most of the restrictions of the profile have the objective of produce static applications where all the tasks and protected objects are defined at library level and tasks never terminate. The profile also avoids constructs that can be very complex or difficult to analyse like the abort, select or requeue statements.

3 Advanced time management

In Ada 95 the real-time management was based on the monotonic clock defined in the `Ada.Real_Time` package and on the `delay` and `delay until` statements. Nowadays Ada provides a much larger diversity of services for time management with the definition of new clocks and timers:

- Execution time clocks for tasks (Ada 2005).
- Execution time clocks for interrupt handlers (Ada 2012).
- Timing events (Ada 2005).
- Execution time timers for tasks (Ada 2005).
- Execution time timers for groups of tasks (Ada 2005).

3.1 Timing events

Package `Ada.Real_Time.Timing_Events` (RM D.15) allows user-defined handlers to be executed at a specific time. The handler is a protected procedure that is executed at interrupt priority directly by the system timer interrupt service routine without the need of using an auxiliary task or a delay statement.

They are intended for applications that require to execute a short action at a very precise time. A typical example would be a control system where the output to the actuators must be updated at a very precise rate. They are also useful to implement scheduling algorithms that requires programming scheduling actions to be done at a future point in time.

As a simple example, Listing 2 shows the body of a protected object used to generate a periodic pulse based on the use of a timing event.

[2]From now on shortened as "RM".

Listing 2: Periodic pulse generator based on a timing event

```
-- Timing event declaration
Pulse : Timing_Event;

...

protected body Pulser is
  procedure Start is
  begin
    Output_High;
    Next_Time := Clock + Pulse_Interval;

    -- Program first timing event expiration
    Set_Handler (Pulse, Next_Time, Handler'Access);
  end Start;

  procedure Stop is
    Cancelled : Boolean;
  begin
    Cancel_Handler(Pulse, Cancelled);
    if not Cancelled then
      raise Handler_Not_Set;
    end if;
  end Stop;

  -- This is the handler of the timing event
  procedure Handler (T : in out Timing_Event) is
  begin
    Output_Swap;
    Next_Time := Next_Time + Pulse_Interval;

    -- Program next timing event expiration
    Set_Handler (Pulse, Next_Time, Handler'Access);
  end Handler;
end Pulser;
```

3.2 Execution time clocks

Each task has an associated execution time clock (package `Ada.Execution_Time`, RM D.14) which measures its execution time, that is, the time spent by the system executing that task. Having such clocks eases the measurement of the worst-case execution times (WCET) of the tasks, a key parameter in the schedulability analysis of the real-time applications.

Besides their usefulness for measuring the WCET, the execution time clocks are also very important in real-time systems because they are the base clocks of the execution time timers, as it will be described in Section 3.3.

3.3 Execution time timers

The execution time timers (package `Ada.Execution_Time.Timers`, RM D.14.1) allow user-defined handlers to be executed when the execution time clock of a task has reached the desired value. From the user's interface point of view, they look very much like the timing events, both have an expiration time and a protected handler procedure.

The main application of these timers is to take corrective actions on WCET overrun situations. This situations are relatively common in modern architectures since pipelines, branch prediction, cache effects, and so on, makes it very complex to measure the actual WCET of a task. Using the execution time timers, the corrective action (lower the task priority, enter in a safe operation mode, etc.) can be done by the timer handler at the very moment the task overruns its WCET.

3.4 Group execution time budgets

The package `Ada.Execution_Time.Group_Budgets` (RM D.14.1) allows to assign execution time budgets to a group of tasks (with the restriction that a task can only belongs to one group). The execution of any task member of the group results in the budget counting down. When the budget becomes exhausted, the user-defined handler (a protected procedure) is executed.

The main application of the group budgets is the implementation of "aperiodic servers" [8] to achieve temporal isolation among different parts of a complex application where each part, maybe an independent application, is made up of a number of tasks.

With the multiprocessor support provided in Ada 2012, the group budget definition has been tuned and now a group budget is attached to a particular processor. Only execution of the tasks in this particular processor reduces the remaining budget of the group.

3.5 Execution time of interrupt handlers

A common assumption is that the effect of the interrupt handlers on the execution time clocks of the tasks is negligible because handlers are usually very short pieces of code. Under that assumption systems charge the time consumed by the interrupt handlers to the task executing when the interrupt is generated.

This assumption may not be realistic in real-time systems that undertake an intensive use of interrupts or uses timing events with relatively long handlers. In these systems, it would be desirable to have a separate account of the interrupt handlers execution time.

The Ada language, in the packages `Ada.Execution_Time` and `Ada.Execution_Time.Interrupts` (RM D.14.3) provides support for the separate accounting of the execution time of interrupt handlers. This time includes the time consumed by the timing event handlers since they are executed directly by the system timer ISR.

The RM allows to the implementations to provide three support levels:

1. No support at all: "it is implementation defined which task, if any, is charged the execution time that is consumed by interrupt handlers" (RM D.13,11/3).
2. Execution time of interrupt handlers not charged to tasks and accounted by one global clock.
3. Execution time of interrupt handlers not charged to tasks and accounted by one different clock for each interrupt.

The support level is defined by the value of the boolean constants `Interrupt_Clocks_Supported` and `Separate_Interrupt_Clocks_Supported` defined in the package `Ada.Execution_Time`.

4 Advanced dispatching

In Ada 95 there was only one predefined dispatching policy called `FIFO_Within_Priorities`. It defines a fixed priority scheduling with FIFO order for tasks with the same priority.

In Ada 2005 three new dispatching policies where defined:

- `Non_Preemptive_FIFO_Within_Priorities` (RM D.2.4): fixed priority without preemption when a higher priority task is runnable.
- `Round_Robin_Within_Priorities` (RM D.2.5): fixed priority with cyclic scheduling between tasks with the same priority.
- `EDF_Across_Priorities` (RM D.2.6): "Earliest Deadline First" scheduling policy.

The policies can be applied to the whole partition using the configuration pragma:

```
pragma Task_Dispatching_Policy (dispatching_policy);
```

They can also be applied to a particular priority range using pragma `Priority_Specific_Dispatching`. The priority specific dispatching will be described in Section 4.4.

4.1 Non-Preemptive dispatching

The policy identifier `Non_Preemptive_FIFO_Within_Priorities` defines a policy identical to `FIFO_Within_Priorities` but without preemption when a higher priority task is runnable. When this policy is in use, a task will run until completion or until it is blocked or executes a delay statement (More technically: the only dispatching points are blocking or termination of a task, a delay, or a call to a "yield" procedure).

This policy is intended to be used in high-integrity applications since it is much more deterministic than preemptive policies and it is a intermediate step between cyclic executives and preemptive multitasking.

Since non-preemption reduces schedulability, it is usual when using a non-preemptive dispatching that tasks volunteer to be preempted at some points of its execution. Traditionally Ada tasks yields the CPU using a `delay 0.0` statement.

In Ada 2012 new yield procedures has been added to packages `Ada.Dispatching` and `Ada.Dispatching.Non_Preemptive` (RM D.2.4). The most interesting of this new yield procedures is `Yield_To_Higher`, this procedure only yields the CPU to tasks with higher priority than the calling task. An interesting point about this procedure is that it can be used from inside a protected action, since if the `Ceiling_Locking` policy is in use the possible preemption cannot put in danger the mutual exclusion achieved by the protected object.

4.2 Round Robin dispatching

The policy `Round_Robin_Within_Priorities` allows a set of tasks with the same priority to make progress at a similar rate:

1. Each task can execute at most during an interval of time called "quantum".
2. When the quantum is exhausted, and the task is not executing a protected operation, it is moved to the tail of its priority queue.
3. The task at the head of the priority queue gets the CPU.

This policy is usually applied to mixed dispatching applications, where the tasks with real-time constrains are dispatched under FIFO or EDF policies at the highest priority levels, and the non real-time tasks are executed at the lowest priority under the Round Robin policy, in order to share the spare time.

The package `Ada.Dispatching.Round_Robin` allows to get and set the quantum assigned to each priority level where the Round robin policy is applied. Note that the RM allows to the implementations to restrict the available quantum values and, in consequence, the quantum assigned by the programmer (using procedure `Set_Quantum`) could be different from the one that is actually been used by the implementation (returned by `Actual_Quantum`).

4.3 Earliest Deadline First dispatching

The EDF policy (`EDF_Across_Priorities`) is the most popular dynamic priority policy. It is based on the concept of "deadline", that is, the time when an activation of a task should have finished its job.

The policy requires a new scheduling attribute to be defined for the tasks: the "relative deadline". For each activation a task has a different "absolute deadline" that is equal to its activation time plus its relative deadline. Tasks at the same priority level are ordered according to their absolute deadlines (the task with the earliest absolute deadline is executed first).

Scheduling theory proves that dynamic priority policies allow a better resource usage in some cases. EDF is the most popular dynamic priority policy for several reasons:

- Its implementation is relatively simple compared to other dynamic priority policies.
- It is optimal in monoprocessors: if a set of tasks is schedulable by any dispatching policy then it will also be schedulable by EDF.
- It can guarantee all the tasks' deadlines at higher processor load (up to 100%) than fixed priorities.

Of course, EDF has disadvantages compared to fixed priority policies:

- It requires a more complex implementation than the fixed priority policies what implies a higher scheduler overhead.
- Under an overload situation the set of tasks that will miss their deadlines is unpredictable.

4.3.1 Managing EDF tasks

The initial relative deadline of a task can be specified with the `Relative_Deadline` aspect (pragma `Relative_Deadline` is declared obsolescent):

```
task EDF_Task with Relative_Deadline => Time;
```

The `Time` value is an expression of type `Real_Time.Time_Span`. The first absolute deadline of the task will be its activation time plus the relative deadline assigned with this aspect. If the aspect is not specified, then the initial absolute deadline of a task is `Ada.Real_Time.Time_Last`.

The package `Ada.Dispatching.EDF` provides operations to set and get the absolute deadline of a task. It also provides the procedure `Delay_Until_And_Set_Deadline` mainly intended to create periodic EDF tasks:

```
-- Periodic EDF task with deadline equal to period
task body Periodic_Task is
   Interval : Time_Span := Milliseconds (10);
   Next : Time;
begin
   Next := Clock; -- Start time
   loop
      -- task's body
      ...
      Next := Next + Interval ;
      Delay_Until_And_Set_Deadline (Next, Interval);
   end loop;
end Periodic_Task;
```

The call to `Delay_Until_And_Set_Deadline` delays the task until the time `Next` and, when the task becomes runnable again, it will have an absolute deadline equal to `Next` plus `Interval`. The use of this procedure avoids unnecessary context switches that can happen if the `Set_Deadline` procedure and the `delay until` statement are used instead.

4.3.2 EDF and the priority ceiling protocol

The definition of the `Ceiling_Locking` policy suffers some changes when applied to EDF tasks. In such situation, the protocol defined by the Ada RM is know in the literature as the "Preemption Level Control Protocol" (PLCP) (also known as "Baker's Protocol" or "SRP Protocol") [9]. This protocol is a generalization of the "Immediate Ceiling Priority Protocol" (ICPP) the Ada interpretation of the `Ceiling_Locking` policy for FIFO tasks.

The PLCP has the same good properties than the ICPP on fixed priorities:

- Minimizes the priority inversion.
- In a uniprocessor, the protocol itself ensures the mutual exclusion (no lock is required).
- A task can only be blocked at the very beginning of its execution.
- A task can only suffer a single block.
- The protocol ensures that deadlocks cannot occur.

The PLCP requires a new parameter for tasks and protected objects: the "preemption level". In the definition of the PLCP, the preemption level is a small integer number that should be assigned to the tasks in deadline monotonic order, the shorter the relative deadline of a task, the higher its preemption level. The preemption level of a protected object is the maximum preemption level of any task that uses it.

In the Ada definition of PLCP the priority of tasks and protected objects is used in the role of the preemption level. So, the declaration of two EDF tasks could be:

```
task EDF_Task_With_Short_Deadline with
     Relative_Deadline => Ada.Real_Time.Milliseconds (10),
      Priority  => 4;

task EDF_Task_With_Long_Deadline with
     Relative_Deadline => Ada.Real_Time.Milliseconds (20),
      Priority  => 3;
```

The rules to integrate the PLCP in the Ada priority based model are quite complex (see RM D.2.6, 23/2-26/3) but they are only relevant for implementers. The programmer only needs to care about setting the preemption level (priority) of tasks and protected objects as explained above.

4.4 Mixed hierarchical dispatching

Ada goes a step further in dispatching flexibility by supporting mixed hierarchical scheduling configurations. A two-levels dispatching model is defined with a base fixed priority policy and several second-level dispatching policies in non-overlapping priority ranges.

The configuration pragma `Priority_Specific_Dispatching` is used for this purpose:

```
pragma Priority_Specific_Dispatching ( policy_identifier ,
     first_priority  ,  last_priority );
```

By using this pragma, tasks with active priority in the range `[first_priority, last_priority]` are scheduled under the policy specified by `policy_identifier` (where `policy_identifier` can be any Ada dispatching policy but the Non-Preemptive which can only be used as the global partition dispatching policy).

When several priority ranges are defined, high priority ranges take precedence over low priority ranges according to the key rule of the base fixed priority policy: the processor is assigned to the first task of the highest occupied priority queue.

A task can "jump" from one priority range to another when its base priority is changed (using package `Dynamic_Priorities`) or while it is inheriting a priority. Special care has to be taken when, due to a base priority change, a task "jumps" to an EDF range. In such situation, and in the case the `Relative_Deadline` aspect was not specified for the task, it will have the longest absolute deadline and, consequently, it will be the less prioritary task in the range.

Protected objects can be used to share data between tasks in different priority ranges. As it could be expected, the

Listing 3: Priority ranges configuration

```
pragma Priority_Specific_Dispatching
   (FIFO_Within_Priorities, 10, 16);
pragma Priority_Specific_Dispatching
   (EDF_Across_Priorities, 2, 9);
pragma Priority_Specific_Dispatching
   (Round_Robin_Within_Priorities, 1, 1);
```

`Ceiling_Locking` rules are obeyed and the promoted task competes with the other tasks in the range according to the priority it has just inherited.

Mixed scheduling allows to combine in the same application the good properties of the different policies. For example, with the configuration described in Listing 3 an application could take advantage of the predictability of the FIFO scheduling for the critic tasks, the better resource usage provided by EDF for the non-critic tasks and the fair distribution of resources provided by the Round robin policy for the non-RT tasks.

5 Multiprocessor support

Multiprocessor architectures are becoming popular in many application areas including the embedded systems. Ada is ready to face this important architectural change thanks to the new services defined in Ada 2012. The core of the new Ada multiprocessor support are the "Dispatching Domains", other services like the "Synchronous Barriers" are also targeted to the multiprocessor architectures.

Ada multiprocessor support is intended for "Symmetric multiprocessing" (SMP). In a SMP architecture two or more *identical* processors are connected to a single shared memory.

Package `System.Multiprocessors` (RM D.16) defines the integer type to identify the processors and also provides a function to know the number of processors in the system.

5.1 Dispatching domains

The package `System.Multiprocessors.Dispatching_Domains` (RM D.16.1) allows to group processors into "Dispatching domains". Each domain is a contiguous range containing one or more processors. Each processor belongs to only one dispatching domain.

At the beginning of the execution all the processors belong to the `System_Dispatching_Domain` and the environment task is allocated to it.

During the *elaboration* of the partition the programmer can create new domains that will remain unchanged during the rest of the execution of the application. As processors are added to the new dispatching domains they are removed from the `System_Dispatching_Domain`. Dispatching domains are created using the `Dispatching_Domains.Create` function:

```
Domain_1 : Dispatching_Domain := Create (15, 17);
```

Every task is allocated to a dispatching domain. Inside its domain, a task can execute in any processor unless it is explicitly assigned to a particular processor. The processor affinity of a task inside its domain can be changed at run-time as many times as desired.

This flexibility allows Ada to support the most popular allocation approaches:

1. Fully Partitioned: each task is allocated to a single processor on which all its jobs must run.
2. Dynamically Partitioned: at run-time the application can change the assignment of a task from one processor to another.
3. Partially Partitioned: tasks are restricted to a subset of the available CPUs, jobs may migrate during execution.
4. Global: all tasks/jobs can run on all processors, jobs may migrate during execution.

By default all the tasks are allocated to the `System_Dispatching_Domain`. A task can be allocated to a user defined dispatching domain using the `Dispatching_Domain` and `CPU` aspects:

```
-- Allocate task to Domain_1 (the task can execute in any
-- processor in the domain)
task T with Dispatching_Domain => Domain_1;

-- Allocate task to Domain_1 and assign it to the processor 16
task T with CPU => 16, Dispatching_Domain => Domain_1;
```

Alternatively, a task allocated to the `System_Dispatching_Domain` can be allocated to a user defined domain with the `Dispatching_Domains.Assign_Task` procedure:

```
-- Allocate task to Domain_1 (the task can execute in any
-- processor in the domain)
Assign_Task (Domain_1, T'Identity);

-- Allocate task to Domain_1 and assign it to the processor 16
Assign_Task (Domain_1, 16, T'Identity);
```

Note that `Assign_Task` can only be used to move tasks from the `System_Dispatching_Domain`. Once a task has been allocated to a user defined domain it will remain in that domain forever.

At any time we can use the the `Set_CPU` procedure to change the affinity of a task inside its domain:

```
-- Assign task to the processor 17
Set_CPU (17, T'Identity);

-- Allow task to execute in any processor in its domain
Set_CPU (Not_A_Specific_CPU, T'Identity);
```

During the revision process it was considered to include in the Ada standard the possibility of specifying dispatching policies on a per-dispatching domain basis.

Although this functionality was finally rejected for been considered too complex, there is a less elegant but efficient approach that can be used. This approach consist on including in a dispatching domain tasks in a specific priority range and use the `Priority_Specific_Dispatching` pragma to apply the desired dispatching policy to that range and consequently to the tasks in the dispatching domain.

5.2 Synchronous Barriers

The synchronous barriers (package `Ada.Synchronous_Barriers`, RM D.10.1) are a synchronization primitive intended for massively parallel machines. To take advantage of the parallelism provided for such architectures, it is usual to use algorithms that can be performed in parallel for a large number of tasks. After this parallel part, it is very common that the algorithm has a final sequential part to recombine the results.

Usually the parallel computations are very short. In that case the use of a complex synchronization primitive would remove any gains obtained from the use of the parallel algorithm. Synchronous barriers have been designed to solve this problem since they can be implemented very efficiently.

Synchronous barriers are used to synchronously release a group of tasks after the number of blocked tasks reaches a specified count value.

A barrier is created specifying is "release threshold" (the number of blocked tasks required for the barrier to be open). When a task reaches the barrier (calls the `Wait_For_Release` procedure) it is blocked in the barrier.

```
procedure Wait_For_Release (
    The_Barrier : in out Synchronous_Barrier;
    Notified  : out Boolean);
```

If the number of blocked tasks reaches the release threshold the barrier is open an all the tasks are released. Only one of the released tasks will be notified with the `Notified` parameter set to `True`. In the case that a final sequential part of the algorithm is required, the programmer can use this notifications to be sure that one, and only one, task will do this final part of the computation.

References

[1] S. T. Taft, R. A. Duff, R. Brukardt, E. Plödereder, and P. Leroy (2006), *Ada 2005 Reference Manual. Language and Standard Libraries - International Standard ISO/IEC 8652/1995 (E) with Technical Corrigendum 1 and Amendment 1*, ser. Lecture Notes in Computer Science, Springer, vol. 4348.

[2] J. Barnes (2008), *Ada 2005 Rationale: The Language, The Standard Libraries*, ser. Lecture Notes in Computer Science, Springer, vol. 5020.

[3] MaRTE OS website. http://marte.unican.es/ Mar. 2013.

[4] M. Aldea Rivas and J. F. Ruiz (2007), *Implementation of New Ada 2005 Real-Time Services in MaRTE OS and GNAT*, Proceedings of the 12th International Conference on Reliable Software Technologies, Springer-Verlag, pp. 29–40.

[5] M. Aldea Rivas, M. González Harbour, and J. F. Ruiz (2009), *Implementation of the Ada 2005 Task Dispatching Model in MaRTE OS and GNAT*, ser. Lecture Notes in Computer Science, F. Kordon and Y. Kermarrec (Eds.), vol. 5570, Springer, pp. 105–118.

[6] *Ada Reference Manual (2013). Language and Standard Libraries - International Standard ISO/IEC 8652/2012 (E) with Technical Corrigendum 1 and Amendment 1*.

[7] *Ada 2012 rationale*. http://www.adacore.com/knowledge/technical-papers/ada-2012-rationale/," Feb. 2013. [Online].

[8] A. Burns and A. Wellings (2005), *Programming Execution-Time Servers in Ada 2005*, Real-Time Systems Symposium, 27th IEEE International, pp. 47 –56.

[9] T. P. Baker (1991), *Stack-based Scheduling of Real-Time Processes*, Real-Time Systems, vol. 3, no. 1, pp. 67–99.