A Framework for Real-Time Utilities for Ada 2005[*]

A.J. Wellings and A. Burns
Department of Computer Science
University of York, UK
{andy,burns}@cs.york.ac.uk

Abstract

Modernlarge real-time systems are becoming more complex. Whilst Ada 2005 provides a comprehensive set of programming mechanisms that allow these systems to be implemented, the abstractions are low level. This paper argues that there is a need for a standardised library of real-time utilities that address common real-time problems .

1 Introduction

Ada has comprehensive support for priority-based real-time systems. The approach has been to provide a set of low-level mechanisms that enable the programmer to construct systems solving common real-time problems. Whilst eminently flexible, this approach requires the programmer to re-implement common paradigms in each new system. In the past, these structures have been quite straightforward, perhaps just involving simply periodic or sporadic tasks communicating via protected data. However, modern large real-time systems are much more complex and include hard, soft and non real-time components. The resulting paradigms are similarly more involved, and require activities like deadline miss detection, CPU budget overrun detection, the sharing of CPU budgets between aperiodic threads etc. Ada 2005 has responded admirably, expanding its set of low-level mechanisms. However, the common problems are now much more complex, and it is no longer appropriate to require the programmer to reconstruct the algorithms in each new system. Instead, what is required is a library of reusable real-time utilities; indeed, ideally such a library should become a de facto secondary standard – perhaps in the same way that Java has developed a set of concurrency utilities over the years that have now been incorporated into the Java 1.5 distribution.

[*]An extended version of this paper also appeared in the proceedings of the AdaEurope 2007 conference. This work has been undertaken within the context of the EU ARTIST2 project.

The goal of this paper is to initiate a discussion in the Ada community to both confirm the need for a library of common real-time utilities and to propose (as a starting point) a framework for their construction. In section 2, an overview of the framework is given that also serves to indicate the scope of the utilities presented in this paper. Section 3 then presents the detailed design of the framework and shows how it can be used. Conclusions are drawn in section 4.

2 Real-Time Utilities – Framework Overview

In the field of real-time programming, real-time tasks are often classified as being periodic, sporadic or aperiodic. Simple real-time periodic tasks are easy to program but once more complicated ones are needed (such as those that detect deadline misses, execution time (budget) overruns, minimum inter-arrival violations etc), the paradigms become more complex. Hence, there is a need to package up some of these and provide them as real-time tasking utilities.

A programmer wanting to use a real-time tasking abstraction will want to indicate (for example):

- whether the abstraction is periodic, sporadic or aperiodic (each task is "released" in response to a release event, which is usually time triggered for periodic tasks and event triggered for sporadic and aperiodic tasks);
- in the event of a deadline miss, either to terminate the current release of the task or to simply inform the program that this event has occurred (in which case, the programmer can choose to react or ignore the event);
- in the event of an execution time overrun, either to terminate the current release of the task or to simply inform the program that this event has occurred (in which case, the program can choose to react or ignore the event);

- whether a task is associated with an execution-time server that can limit the amount of CPU-time it receives.

- whether a task can operate in one or more modes, and if so, the nature of the mode change.

This section illustrates how real-time task abstractions, supporting some of these variations, can be developed in Ada 2005.

The approach that has been taken is to divide the support for the tasks into four components.

1. The functionality of the task – this is encapsulated by the `Real_Time_Task_State` package. Its goal is to define a structure for the application code of the tasks. It is here that code is provided: to execute on each release of the task, to execute when deadline misses occur and when execution time overruns occurs. In this paper, it is assumed that the task only wishes to operate in a single mode.

2. The mechanisms need to control the release of the real-time tasks and to detect the deadline misses and execution time overruns – this is encapsulated in the `Release_Mechanisms` package. Each mechanism is implemented using a combinations of protected objects and the new Ada 2005 timer and execution time control features.

3. The various idioms of how the task should respond to deadline misses and execution time overruns – this is encapsulated in the `Real_Time_Tasks` package. It is here that the actual Ada tasks are provided.

4. The mechanisms needed to encapsulate subsystems and ensure that they are only given a fixed amount of the CPU resource (often called temporal firewalling) – this is the responsibility of the `Execution_Servers` package. This paper focuses on using these mechanisms to support aperiodic task execution.

Figure 1 illustrates the top level packages that make up the abstractions. The details are discussed in the following section.

3 Framework Design

This section describes the details of the design of the framework introduced in Section 2. It assumes fixed priority scheduling and that the deadlines of all periodic tasks are less than or equal to their associated periods.

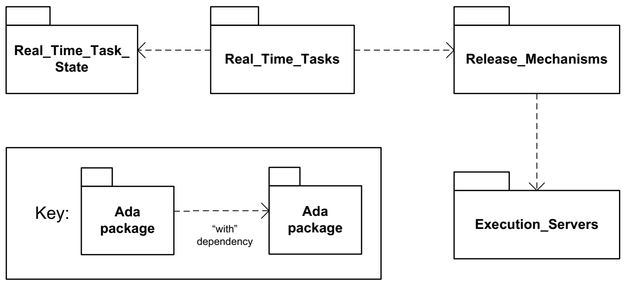

Figure 1. Top-Level Packages

3.1 Real-Time Task State

First, it is necessary to provide a structure within which the programmer can express the code that the real-time task wishes to execute, along with its associated state variables. This is achieved, in the usual Ada object-oriented fashion, by defining the state within a tagged type, and providing operations to execute on the state. The following package shows the state and operations that all real-time tasks need.

```
-- with and use clauses omitted
package Real_Time_Task_State is
  type Task_State is abstract tagged record
    Relative_Deadline : Time_Span;
    Execution_Time : Time_Span := Time_Span_Last;
    Pri : Priority := Default_Priority;
  end record;
  procedure Initialize(S: in out Task_State)
            is abstract;
  procedure Code(S: in out Task_State) is abstract;
  procedure Deadline_Miss(S: in out Task_State)
            is null;
  procedure Overrun(S: in out Task_State) is null;
  type Any_Task_State is access all Task_State'Class;
end Real_Time_Task_State;
```

Every real-time task has a deadline, an execution time and a priority. Here, these fields are made public, but they could have just as well been made private and procedures to 'get' and 'set' them provided. No additional assumptions have been made about the values of these attributes. For example, the execution time could be worst-case or average.

The operations to be performed on a task's state are represented by the four procedures:

- `Initialize` – this code is used to initialize the real-time task's state when the task is created;

- `Code` – this is the code that is executed on each release of the task;

- `Deadline_Miss` – this is the code that is executed if a deadline is missed.

- `Overrun` – this is the code that is executed if an execution time overrun occurs.

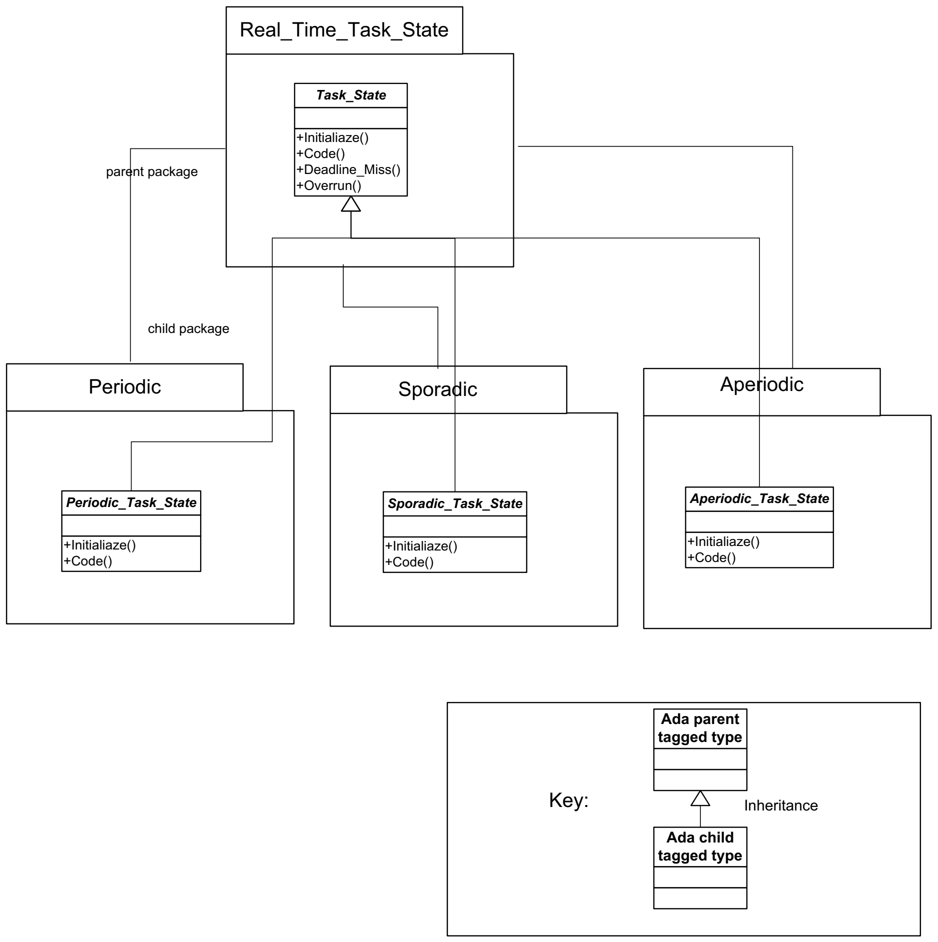

Figure 2. Task States

Note, all real-time code must provide the `Initialize` and the `Code` procedures. There are default null actions on a missed deadline and on an execution time overrun.

Child packages of `Real_Time_Task_State` provide support for periodic, aperiodic and sporadic task execution (as illustrated in Figure 2).

A periodic task's state includes that of a real-time task with the addition of its period of execution. In other words, it has regular time-triggered releases.

```
package Real_Time_Task_State.Periodic is
  type Periodic_Task_State is abstract new Task_State
   with record Period : Time_Span; end record;
  procedure Initialize(S: in out Periodic_Task_State)
            is abstract;
  procedure Code(S: in out Periodic_Task_State)
            is abstract;
  type Any_Periodic_Task_State is access all
       Periodic_Task_State'Class;
end Real_Time_Task_State.Periodic;
```

There is more than one model of a sporadic task; here, it is assumed that the task must have an enforced minimum inter-arrival time between releases (another approach would be to enforce a maximum arrival frequency). Hence, the state includes this value.

```
package Real_Time_Task_State.Sporadic is
  type Sporadic_Task_State is abstract new Task_State
  with record MIT : Time_Span; end record;
  procedure Initialize(S: in out Sporadic_Task_State)
            is abstract;
  procedure Code(S: in out Sporadic_Task_State)
            is abstract;
  type Any_Sporadic_Task_State is access all
       Sporadic_Task_State'Class;
end Real_Time_Task_State.Sporadic;
```

The state for aperiodic tasks has no new fields over the normal `Task_State`, but for uniformity, a new type can be created.

Application real-time tasks choose the appropriate real-time state to extend, and add their own state variables. For example, the following shows the application code to be used with the declaration of a periodic real-time task that is not interested in any missed deadlines or execution-time overruns.

```
  type My_State is new Periodic_Task_State with
  record
    -- state variables
  end record;
  procedure Initialize(S: in out My_State);
  procedure Code(S: in out My_State);

  Example_State: aliased My_State := (
          Pri=> System.Default_Priority + 1);
```

3.2 Real-Time Task Release Mechanisms

Real-time tasks can be released by the passage of time or via a software/hardware event. The following package (`Release_Mechanisms`) provides the common interfaces for all mechanisms (illustrated in Figure 3).

The root of the interface hierarchy (`Release_Mechanism`) simply supports the facility for a real-time task to wait for notification of its next release to occur (be it a time or an event triggered release). `Release_Mechanism_With_Deadline_Miss` is provided for the case where the real-time task wishes to be informed when it has missed a deadline. Similarly, `Release_Mechanism_With_Overrun` is provided for the case where the real-time task wishes to be informed when it has overrun its execution time. Finally, `Release_Mechanism_With_Deadline_Miss_And_Overrun` allows both detection of deadline misses and execution time overruns. The Ada code is shown below for some of the above.

```
package Release_Mechanisms is
  type Release_Mechanism is synchronized interface;
  procedure Wait_For_Next_Release(R: in out
       Release_Mechanism) is abstract;
  type Any_Release_Mechanism is access all
       Release_Mechanism'Class;

  type Release_Mechanism_With_Deadline_Miss is
       synchronized interface and Release_Mechanism;
  procedure Wait_For_Next_Release(R : in out
       Release_Mechanism_With_Deadline_Miss)
       is abstract;
  procedure Inform_Of_A_Deadline_Miss(R : in out
       Release_Mechanism_With_Deadline_Miss)
       is abstract;
  type Any_Release_Mechanism_With_Deadline_Miss
       is access all
       Release_Mechanism_With_Deadline_Miss'Class;
  ...
end Release_Mechanisms;
```

Child packages provide the actual release mechanisms. For example, Figure 4 shows the mechanisms for periodic real-time tasks.

For example, the following shows the structure of a protected type that implements a periodic release mechanisms. The period of the task is obtained from the task's state, which is passed in as an access discriminant. The application code simply calls the `Wait_For_Next_Release` entry.

```
-- with and use clauses omitted
package Release_Mechanisms.Periodic is
  protected type Periodic_Release(
          S: Any_Periodic_Task_State) is
          new Release_Mechanism with
    entry Wait_For_Next_Release;
    pragma Priority(System.Interrupt_Priority'Last);
  private
    ...
  end Periodic_Release;
end Release_Mechanisms.Periodic;
```

Another protected type can support deadline miss detection

```
  protected type Periodic_Release_With_DM(
       S: Any_Periodic_Task_State;
       Termination : Boolean) is
       new Release_Mechanism_With_Deadline_Miss with
    entry Wait_For_Next_Release;
    entry Inform_Of_A_Deadline_Miss;
    pragma Priority(System.Interrupt_Priority'Last);
  private
    ...
  end Periodic_Release_With_DM;
```

Here, a boolean indicates whether the application requires notification or termination of a deadline miss. If termination is required, the `Inform_Of_A_Deadline_Miss` entry can be used by the framework in a select-then-abort statement.

3.3 Aperiodic release mechanisms and execution servers

The final type of release mechanism is that for handling aperiodic releases. Typically, the CPU time allocated to aperiodic tasks must be constrained as they potentially can have unbounded resource requirements. The Ada 2005 group budget facility can be used to implement the various approaches.

Before the release mechanism can be programmed, it is necessary to consider how it interacts with the execution severs. This paper will only consider periodic execution servers. The following package specification defines the common interface.

```
package Execution_Servers is
  type Server_Parameters is tagged record
    Period : Time_Span; Budget : Time_Span;
  end record;

  type Execution_Server is synchronized interface;
  procedure Register(ES: in out Execution_Server;
           T : Task_Id) is abstract;
  ...
  type Any_Execution_Server is access
       all Execution_Server'Class;
end Execution_Servers;
```

All servers have parameters that determine the servers characteristics. They include:

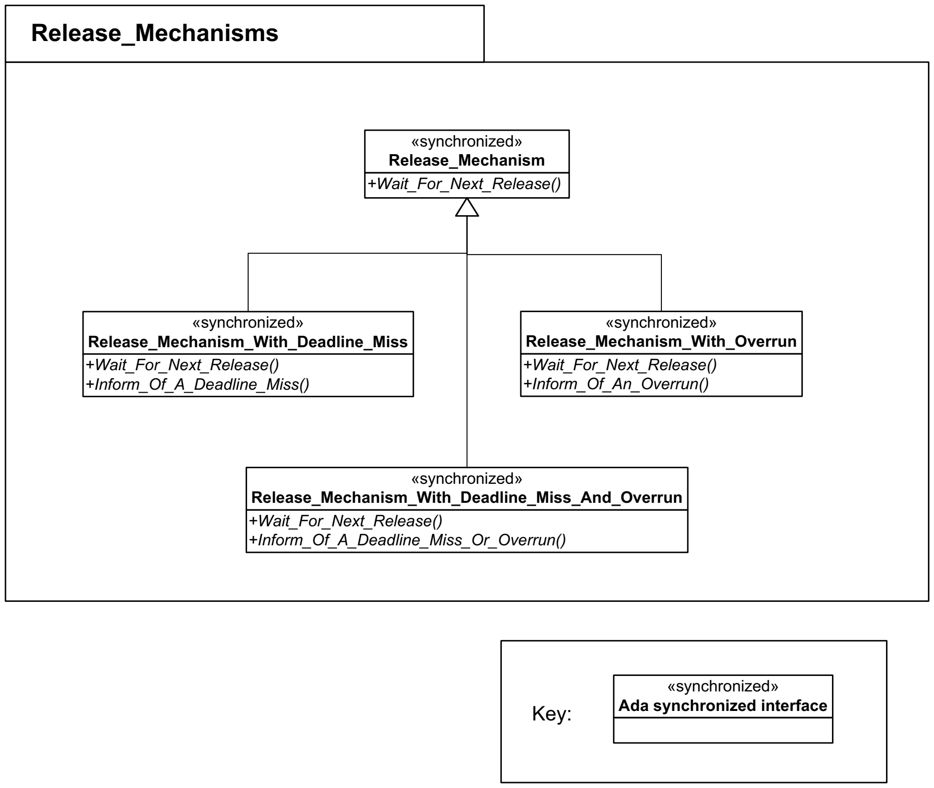

Figure 3. Release Mechanism Interfaces

- the budget – how much CPU time has been allocated to the server;

- the period – this relates to how often the server's budget is replenished.

Two of the main servers found in the literature (the deferrable [1] and sporadic servers[2]) also require their clients to have foreground and background priorities, but in the general case this may not be the situation. Some servers suspend their clients when their execution time expires, and other servers allow their clients to have different priorities.

All execution servers require their clients to register. Here, any task can register any other task. Some types of execution servers will also want to know when the client tasks are executable. These periods of execution are called *sessions*. The associated procedures have default null values.

To facilitate the use of execution servers, it is necessary to modify the release mechanism to allow the actual server to be passed as a discriminant. The approach is illustrated by considering aperiodic releases (although, the same approach could be applied to the periodic or sporadic release mechanisms).

```
package Release_Mechanisms.Aperiodic is
  protected type Aperiodic_Release(
      S: Any_Aperiodic_Task_State;
      ES: Any_Execution_Server)
        is new Release_Mechanism with
    entry Wait_For_Next_Release;
    procedure Release;
    pragma Priority(Interrupt_Priority'Last);
  private
    ...
  end Aperiodic_Release;
end Release_Mechanisms.Aperiodic;
```

3.4 Real-Time Tasks

The `Real_Time_Tasks` package provides the code that integrates the task states with the release mechanisms to provide the required application abstraction. Several task types are shown in the following package specification.

```
-- with and use clauses omitted
package Real_Time_Tasks is
  task type Simple_RT_Task(
       S : Any_Task_State;
       R : Any_Release_Mechanism;
       Init_Prio : Priority) is
    pragma Priority(Init_Prio);
  end Simple_RT_Task;
```

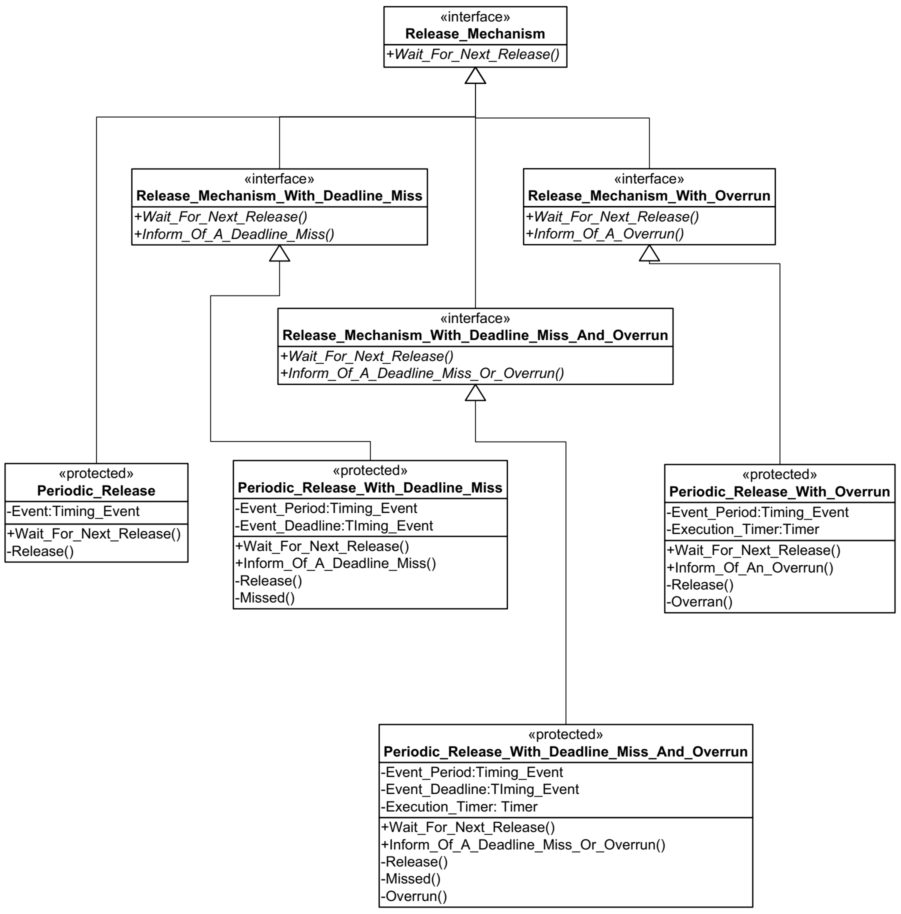

Figure 4. Release Mechanism Classes

```
  task type RT_Task_With_Deadline_Termination(
       S : Any_Task_State;
       R : Any_Release_Mechanism_With_DM;
       Init_Prio : Priority) is
    pragma Priority(Init_Prio);
  end RT_Task_With_Deadline_Termination;
  ...;
end Real_Time_Tasks;
```

Others can be designed; for example, for the cases where the current release of the task must be immediately terminated if the execution time is exceeded.

Note that the priority at which the task activates can be given as a discriminant (this can be changed in the application initialization code for the execution phase of the task).

Note also that if the `Simple_RT_Task` is used with one of the release mechanisms that support deadline miss or execution-time overrun detection, it should set the `Termination` discriminant to false. This will allow the task to be notified using a direct call to the `Deadline_Miss` and `Overrun` operations via the `Task_State`.

The body of this package shows the various structures. Note the use of the ‘select-then-abort’ statement to achieve the termination semantics.

```
package body Real_Time_Tasks is
  task body Simple_RT_Task is
  begin
    S.Initialize;
    loop
      R.Wait_For_Next_Release;
      S.code;
    end loop;
  end Simple_RT_Task;

  task body RT_Task_With_Deadline_
            Termination is
  begin
    S.Initialize;
    loop
      R.Wait_For_Next_Release;
      select
        R.Inform_Of_A_Deadline_Miss;
        S.Deadline_Miss;
      then abort
        S.code;
      end select;
    end loop;
  end RT_Task_With_Deadline_
      Termination;
  -- similar structures for the other task types
end Real_Time_Tasks;
```

3.5 A simple example

Consider a simple example of two identical periodic tasks that wish to detect deadline misses: one requires termination semantics, the other just wishes to be notified. First, the application code is defined

```
 type My_State is new Periodic_Task_State with record
    I : Integer;
  end record;

  procedure Initialize(S: in out My_State);
  procedure Code(S: in out My_State);
  procedure Deadline_Miss(S: in out My_State);

  Example_State1: aliased My_State;
  Example_State2: aliased My_State;

  Releaser1 : aliased  Periodic_Release_With_DM(
      Example_State1'Access, Termination => True);
  Releaser2 : aliased Periodic_Release_With_DM(
      Example_State2'Access,
      Termination => False);
```

In the above, two instances of the state are created, one for each real-time task. There are two protected objects for the release mechanisms : `Release1` supports the termination model, and `Release2` supports the notification model

Now, the real-time tasks can be declared:

```
  T1 : RT_Task_With_Deadline_Termination(
            Example_State1'Access, Releaser1'Access,
            Default_Priority);
  T2 : Simple_RT_Task (
            Example_State2'Access, Releaser2'Access,
            Default_Priority);
```

Here, `T1` uses the real-time task type that supports the termination semantics, and `T2` uses the simple real-time task type.

For completeness, the code of the tasks are given (they are identical, in this example and simply manipulate an integer state variable).

```
  procedure Initialize(S: in out My_State) is
  begin
    S.I := 2; S.Pri := Default_Priority + 2;
    S.Relative_Deadline := To_Time_Span(0.1);
  end Initialize;

  procedure Code(S: in out My_State) is
  begin  S.I := S.I * 2; end Code;

  procedure Deadline_Miss(S: in out My_State) is
  begin S.I := 2; end Deadline_Miss;
```

The body initializes the state, sets the priority and the deadline; it then squares the state value on each periodic release. On a deadline miss, the value is reset to 2.

4 Conclusions

The Ada designers have learned well the lesson of trying to support too higher level abstractions for real-time programming, that was evident in Ada 83. That version was heavily criticised. Ada 95 responded to those criticisms and is an efficient language for high-reliable long-lived real-time applications. Ada 2005 has continued the recovery, and the language now provides a comprehensive set of mechanisms that can support the development of modern large real-time systems.

However, the complexity of modern real-time systems means that there is now the need to provide high-level real-time programming abstractions in the form of standardised library utilities. The goal of this paper has been to start the debate on how the Ada community should respond to this challenge. The initial focus has been on the construction of a framework that allows the provision of real-time tasking utilities. The framework has been implemented using the evolving Ada 2005 compiler from AdaCore along with a local simulation facilities for the new Ada 2005 real-time mechanisms.

References

[1] J.P. Lehoczky, L. Sha, and J.K. Strosnider. Enhanced aperiodic responsiveness in a hard real-time environment. In *8th IEEE RTSS*, pages 261–270, 1987.

[2] B. Sprunt, J. Lehoczky, and L. Sha. Exploiting unused periodic time for aperiodic service using the extended priority exchange algorithm. In *9th IEEE RTSS*, pages 251–258, 1988.