Incorporating Operating Modes to an Ada Real-Time Framework

Jorge Real and Alfons Crespo
Universidad Politécnica de Valencia, Spain
jorge@disca.upv.es*

Abstract

Based on a previous proposal of a framework of real-time utilities in Ada 2005, this paper deals with the extension of that framework to include operating modes and mode changes.

1 Introduction

Ada 2005 has widened the range of real-time facilities in the language: new scheduling policies, new timing and CPU timing mechanisms, integration of object oriented and concurrent programming, etc. The new set of abstractions has significantly increased the versatility and expressive power of the language.

However, these are low level abstractions like setting a timer, synchronizing concurrent tasks, or passing data among them. Abstractions that are adequate for a programming language, but not powerful enough themselves to abstract away much of the complexity of modern, large real-time systems. These systems:

- are formed by components with different levels of criticality that need sharing the available computing resources, e.g. by means of execution-time servers,
- perform activities subject to different release and scheduling mechanisms,
- require the management of timing faults if and when they occur at execution time,
- are composed by several modes of operation, characterized by performing different sets of activities and under different timing requirements.

At the last International Real-Time Ada Workshop, a position paper was submitted that revealed these issues and proposed the implementation of a framework for the construction of a library of common real-time utilities [6, 5]. That paper, which constitutes the starting point of this one, addressed some of the needed features: the possibility to choose the activation pattern of a task; the possibility to implement deadline and overrun handlers; and the implementation of execution-time servers. Other features were not covered but left as future work, as reported in [2].

This paper proposes an approach to extend the original framework by defining modes of operation and providing the mechanisms to enable mode changes.

The paper is organized as follows: Section 2 brings the context by summarizing the originally proposed framework. Section 3 details the proposed approach for the incorporation of operating modes to the framework. Section 4 illustrates the use of the extended framework by means of an example. Finally, Section 5 concludes the paper and identifies topics that need be addressed in the future.

*This work is partly funded by the Spanish *Ministerio de Ciencia e Innovación*, project RTModel, reference TIN 2008-06766-C03-02.

2 Original Framework

The framework proposed in [6, 5] (referred to as *the original framework* in this paper) is explained in those publications and, to a larger extent, in chapter 16 of [1]. Therefore, only a brief, limited overview of the original framework is given in this section for the sake of completeness.

The main goal of the original framework is to provide reusable, high-level abstractions for building real-time systems. These abstractions represent real-time, concurrent activities. In the original proposal, they allow to define:

- the nature of the activation mechanism: periodic, sporadic or aperiodic,
- a mechanism to manage deadline misses at run time,
- a mechanism to manage execution-time overruns,
- the possibility to limit the amount of CPU time devoted to tasks by means of execution-time servers.

The proposal in this paper is only prototypical, a starting point for discussion, and it does not cover the full original framework. Therefore this summary will skip the details of the original framework concerning deadline miss and execution overrun management, as well as the use of execution-time servers. Our purpose is limited to explore the ability of the framework to be extended to support multi-moded systems.

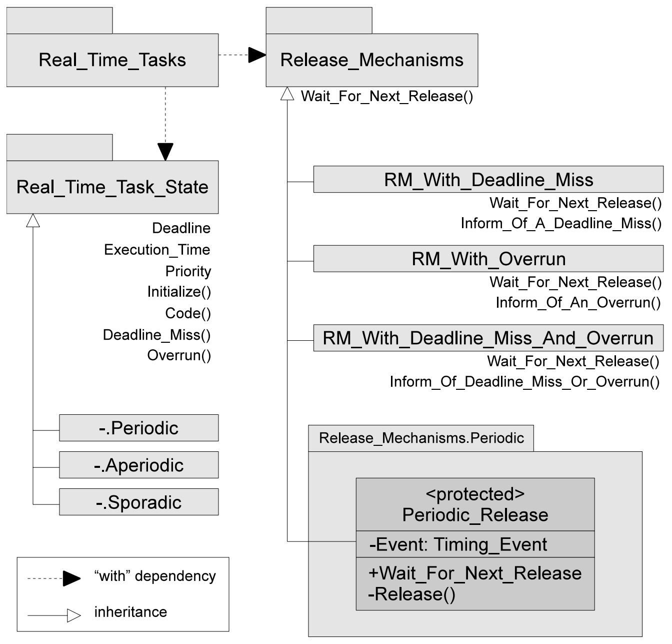

Figure 1. Top-level packages of the original framework

The top-level packages of the original framework are depicted in figure 1. They give the support needed for implementing different kinds of real-time tasks. In particular,

`Real_Time_Tasks` provides idioms to specify how tasks must respond to deadline misses and execution-time overruns. Currently the package offers two types of tasks: a simple real-time task, not affected by these events, and a task type that will execute a proper handler just before being aborted.

`Real_Time_Task_State` encapsulates the functionality of the task. It allows defining its deadline, execution time and priority, and to implement procedures for the initialization, regular code, and handlers for both deadline miss and overrun (the latter two are predefined as null procedures). Three child packages extend and specialize this common interface to provide specific support for periodic, aperiodic and sporadic tasks.

Release_Mechanisms is a synchronized interface from which task activation mechanisms are derived. The original framework allows for activations with or without notification of events to the task (deadline miss and/or overrun). Child packages are provided for periodic, aperiodic and sporadic release mechanisms. Figure 1 shows in more detail the periodic release mechanism extension, implemented by means of a protected interface. At the root of release mechanisms, an abstract procedure Wait_For_Next_Release is provided. This procedure is called by the task once per activation and is in charge of triggering the next release of the task at the proper time.

The original framework gives support to tasks whose actions are aborted (by means of asynchronous transfer of control, ATC) upon a deadline miss or an execution-time overrun. This feature is in turn supported by another package called Notification_Mechanisms (not shown in figure 1). This package is solely used by Release_Mechanisms to give support to the so called Release_Mechanism_With_Event_Notification synchronized interface. It is from this interface that release mechanisms with deadline miss and/or overrun detection are implemented.

The implementation of notification mechanisms in the original *single-moded* framework, suggests that mode change requests can be supported via notifications[1]. However, ATC is not the way to go in general for mode changes: tasks may require to completely execute their actions even in the presence of a mode change transition [4, 3]. Hence section 3 presents a different approach based on the management of the release mechanism, instead of the notification mechanism. Nevertheless, the mode change event might be useful for some types of multi-moded tasks.

Also in the top level of the original framework, a package called Execution_Servers provides temporal firewalling mechanisms. This package is not represented in figure 1 for simplicity, since the use of execution-time servers is not considered in this paper.

3 Adding Operating Modes

The addition of operating modes to the original framework has, a priori, a considerable impact in its overall structure, because the concept of operating modes was not considered in its design. We divide this section in two subsections, the first one deals with the needed data structures to support the notion of operating modes and the second explains how mode change mechanisms can be programmed under the multi-moded framework.

Note that this work focuses only on a part of the original framework. We are excluding execution-time servers from this proposal, as well as all release mechanisms except periodic. Our aim is to tackle those parts of the framework at a later time.

For simplicity, we shall use the names of regular periodic, simple real-time tasks; a further implementation of the multi-moded framework may provide new types of tasks that are multimoded (hence they exhibit a different structure) and leave the original task type names for mode-independent tasks.

3.1 Data Structures

The first thing needed for our purpose is a data type that represents the notion of operating modes. For that purpose, the user must provide the following simple package specification:

```
package List_Of_Modes is
   type Operating_Modes is (Slow,Medium,Fast); --Say
   Initial_Mode : Operating_Modes := Slow;     --Say
end List_Of_Modes;
```

Now we have a system-wide notion of operating modes. Since tasks will participate in different operating modes, their state must be extended to contain mode-dependent information. The package Real_Time_Task_State defines the interface type Task_State that we have modified to include the task attributes in a per mode basis, as well as the needed read and write subprograms to access these attributes. The corresponding specification[2] is shown in figure 3, in appendix A. For example, instead of a single deadline, the task state contains now an array of deadlines, one for each possible operating mode. Other needed data structures are the array of booleans Runs_In, which indicates whether the task is active or not for all modes; and the matrix Transition_Offset with the needed offsets for the first release of a task in a given new mode, when coming from a given previous mode. The initialization code of the task must give appropriate values to the tasks' state variables.

[1]One of the literals used to identify events in the corresponding enumeration type is Mode_Change_Event — although it is obviously not used in the original framework.

[2]The body of this package is trivial, since all its subprograms simply read or write the appropriate state variables.

3.2 Mode Change Mechanisms

According to the object oriented approach followed by the original framework, the implementation of mode change mechanisms follows the same paradigm. The package `Mode_Managers` (see below) defines the synchronized interface `Mode_Manager_Type` from where mode managers can be derived. The abstract operations declared are two procedures (that might be implemented by entries) and three functions. The subprogram `Change_To` allows tasks to request a change of operating mode. `Detect_Mode_Change` allows tasks to ask the mode manager about the occurrence of a mode change. The second parameter of this subprogram, an access to task state, is intended for the task to communicate its mode notion to the mode manager, so that it can be easily decided whether the task has a correct notion of what the current mode is (hence there is no mode change to inform the task about) or not (hence there has been a mode change and the task is not yet aware of it). This second parameter is also useful to update the task's mode notion when there is a mode change. The third parameter of `Detect_Mode_Change` is a release mechanism. This is for the mode manager to be able to cancel further releases of the task upon a mode change (when the task is not active in the new mode), or to reschedule the next release at the adequate time after the mode change request (MCR).

Finally, the functions provided by the `Mode_Manager_Type` interface allow the callers to obtain the current mode, the previous mode, and the absolute time when the last MCR occurred.

```ada
with List_Of_Modes;        use List_Of_Modes;
with Ada.Real_Time;        use Ada.Real_Time;
with Real_Time_Task_State; use Real_Time_Task_State;
with Release_Mechanisms;   use Release_Mechanisms;
package Mode_Managers is
   type Mode_Manager_Type is synchronized interface;
   procedure Change_To(M : in out Mode_Manager_Type;
                       New_Mode : Operating_Modes) is abstract;
   procedure Detect_Mode_Change (M : in out Mode_Manager_Type;
                                 S : Any_Task_State;
                                 R : Any_Release_Mechanism) is abstract;
   function Current_Mode (M : in Mode_Manager_Type) return Operating_Modes is abstract;
   function Previous_Mode (M : in Mode_Manager_Type) return Operating_Modes is abstract;
   function Time_Of_Last_MCR (M : in Mode_Manager_Type) return Time is abstract;
   type Any_Mode_Manager_Type is access all Mode_Manager_Type;
end Mode_Managers;
```

A new mode manager can be defined by extending the `Mode_Manager_Type` interface. It is appropriate to define our own mode manager in a child package of `Mode_Managers`. An example will be given in section 3.3.

We have mentioned the convenience for the mode manager to access the release mechanisms of tasks in order to cancel further releases upon a MCR. Release mechanisms of the original framework are not prepared for canceling an already scheduled release of a periodic task. We have modified the original `Release_Mechanism` synchronized interface to provide new subprograms that allow canceling and also rescheduling the next release to a given time. The latter is particularly useful to schedule the first activation of the task in the new mode. The package `Release_Mechanisms` is now as follows (excluding the `Release_Mechanism_With_Event_Notification` synchronized interface):

```ada
with Ada.Real_Time; use Ada.Real_Time;
package Release_Mechanisms is
   type Release_Mechanism is synchronized interface;
   procedure Wait_For_Next_Release(R : in out Release_Mechanism) is abstract;
   procedure Stop_Releasing (R : in out Release_Mechanism) is abstract;
   procedure Resume_Releasing (R: in out Release_Mechanism;
                               Next_Release_At : Time) is abstract;
   type Any_Release_Mechanism is access all Release_Mechanism'Class;
end Release_Mechanisms;
```

The mode manager must be invoked by periodic tasks, which affects their structure. In the original framework, two task patterns define the available task structures: `Simple_Real_Time_Task` and `Real_Time_Task_With_Event_Termination`. Focusing on the former, we propose the multi-moded, periodic task pattern shown in figure 4. Note that the first `with` clause refers to the package `Modes`, not yet mentioned. This package must define the mode manager in use. It has the following aspect:

```
with Mode_Managers.My_Protocol; use Mode_Managers.My_Protocol;
package Modes is
   Mode_Manager : My_Mode_Manager_Type;
end Modes;
```

We are assuming here that a type `My_Mode_Manager_Type` is implemented in the child package `Mode_Managers.My_Protocol` as an extension to the synchronized interface `Mode_Manager_Type`. The following section gives an example implementation of a mode manager.

Back to the body of periodic multi-moded tasks in figure 4, we see that the task first calls the initialization code and then enters an infinite loop. In the loop, the task uses a select-then-abort statement. The abortable part is the call to `Wait_For_Next_Release`, managed by the periodic release mechanism. The triggering statement is a call to `Detect_Mode_Change`. If a mode change occurs while the task is waiting for the next release, then the waiting is aborted and the mode manager must deal with the mode change with respect to the task. If the task is active in the new mode, then the release mechanism is reprogrammed to offset the task's activation in the new mode. The conditional sentence after the asynchronous select statement allows the task to either execute its code (if there has been no mode change during the waiting) or to set the task's priority for the new mode. If the task is not active in the new mode, then the mode manager must cancel its release mechanism and hold it inactive until a new mode change occurs to an active mode for the task.

3.3 A mode manager example

This section deals with the implementation of a particular mode manager. Figure 5 in appendix A shows the specification of the type `My_Mode_Manager_Type`. This mode manager type is a protected interface that specializes the synchronized interface `Mode_Manager_Type` (see listing in previous section). `My_Mode_Manager_Type` overrides the subprograms provided by its parent type and defines two private entry families and state variables of the mode manager. The body is given in figure 6.

The implementation of functions is straightforward: they simply return the value of the corresponding state variables of the mode manager (current mode, previous mode and time of the last MCR). The protected procedure `Change_To` updates these three values according to the MCR. This in turn causes the re-evaluation of some entry barriers in the protected object.

We have seen in figure 4 how the entry `Detect_Mode_Change` is used by periodic tasks in an asynchronous select statement. The purpose of this entry is to inform the task of a mode change. The task passes its state and release mechanism as parameters to the entry. The entry should evaluate whether the mode notion of the task corresponds or not with the current operating mode. Since the task state is a parameter, this cannot be checked in the entry barrier. Therefore, the entry call is immediately requeued (with abort) to the adequate member of the private entry family `Evaluate_Mode_Change_Condition`. It is now possible to evaluate the mode change in the barrier of the private entry (`when M /= The_Current_Mode`). Note that, since the call was requeued with abort, it is possible to cancel the task's call to `Detect_Mode_Change` when the call to `Wait_For_Next_Release` terminates.

The mode manager considers that there has been a mode change for the calling task when the task's mode notion differs from the actual current mode. This is checked in the barrier of `Evaluate_Mode_Change_Condition`. In the body of this entry, the mode manager decides what to do with the calling task when there is a mode change. The first action is to stop the task's release mechanism, still programmed for the previous mode. Then a decision is made depending on whether the task is active or not in the new mode. If the task is active in the new mode, then its mode notion is changed to the new mode. Otherwise, the task is requeued to the appropriate member of the entry family `Wait_For_Active_Mode`, where it will be queued until a new MCR arrives. The body of `Wait_For_Active_Mode` is very similar to the body of `Evaluate_Mode_Change_Condition`, with the difference that there is no need to stop the release mechanism of the task. If the task is active in the new mode, then its mode notion is set to the current mode. Otherwise, the task is requeued again to `Wait_For_Active_Mode`.

4 A Use Example

A simple multi-moded system has been implemented to check the proposed approach. The system has three operating modes (named `Slow`, `Medium` and `Fast`) and three periodic tasks (`T1`, `T2` and `T3`). The participation of the tasks in the different modes is shown in table 1, along with their respective priorities and periods.

	Slow	Medium	Fast
T1	*P=6; T=2000*	*P=7; T=1000*	—
T2	*P=10; T=1000*	—	—
T3	—	—	*P=15; T=500*

Table 1. Modes and tasks of the example system. P represents priority and T period in milliseconds

Mode `Slow` is defined to be the initial mode of the system. Mode change requests are originated only by tasks `T1` and `T3`. The possible transitions are depicted in figure 2. Note there is no possible transition from `Fast` to `Slow`. Finally, upon a mode change, transition offsets will be applied to tasks as follows:

- `T1` will be released 1500 ms after the MCR when changing from `Slow` to `Medium`.
- `T1` will be released 3000 ms after the MCR when changing from `Medium` to `Slow`.
- `T1` will be released immediately after the MCR when changing from `Fast` to `Medium`.
- `T2` will be released 3000 ms after the MCR when changing from `Medium` to `Slow`.
- `T3` will be released 1000 ms after the MCR when changing from `Medium` to `Fast`.
- `T3` will be released immediately after the MCR when changing from `Slow` to `Fast`.

In order to build this example system we need to:

1. Create the package `List_Of_Modes`. For this example, the package given in section 3.1 is perfectly valid.
2. Instantiate the tasks and associate their release mechanisms. We create a new package for that purpose, named `Periodic_Test_MM` and instantiate the tasks in its specification. Figure 7 shows this specification. Note that, although all the state types look the same, we need to define three different types because the subprograms `Initialize` and `Code` are different for the different types (see package body below).
3. Implement the tasks' actions. These are implemented in the body of `Periodic_Test_MM`. Concrete bodies must be given for the `Initialize` and `Code` subprograms of each task state. Figure 8 lists the test body. For all tasks, the `Initialize` procedure sets the task's state variables appropriately: participation of the task in the different modes, periods, transition offsets, priorities, and the initial priority. Then the internal variable `I` is given an initial value and a message is printed.The `Code` subprograms simply increment `I` by one, print its value, and in the case of tasks `T1` and `T3`, produce a MCR at certain points, depending on that value.
4. Put it all together. Since we have decided to define and implement the system in the package `Periodic_Test_MM`, we need a subprogram to put it all together by simply importing that package. For example:

```
with Periodic_Test_MM; use Periodic_Test_MM;
with Ada.Text_IO;      use Ada.Text_IO;
procedure Main_MM is
begin
   Put_Line ("Main execution");
end Main_MM;
```

We have traced the execution of `Main_MM` to check that all mode changes are processed correctly. Periods, priorities and transition offsets were all correct. We have used the GAP 2008 compiler of AdaCore on a Linux platform combined with the MaRTE run-time system.

The following is a screen sample of an instrumented version of the example system, showing (an excerpt of) the trace of the first 30 seconds of execution. The state variables of tasks T1, T2 and T3 are initialized to 1000, 2000 and 3000 respectively, so that they can be easily followed in this trace.

The first 5 lines show the initialization of the tasks. The first one is task `T3`. According to the listing in figure 4, the task calls `Initialize` and then enters the loop and the ATC by calling `Detect_Mode_Change`. Then according to the

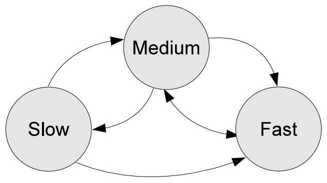

Figure 2. Mode transitions in the example system

body in figure 6, the call is requeued to `Evaluate_Mode_Change_Condition (Fast)` because the initial mode for T3 is precisely `Fast`. Since the task's mode notion differs from the actual current mode, the barrier of this entry is open. Therefore the release mechanism associated to T3 is stopped and, since T3 is not active in mode `Slow` (the initial mode), the call is requeued now to `Wait_For_Active_Mode (Slow)`. Tasks T1 and T2 start executing at times 1.0 and 2.0 seconds respectively[3]. They execute at their assigned periods of 2.0 and 1.0 seconds, respectively. At time 4.0 task T1 produces a MCR to mode `Slow`, which happens to be the current mode. It can be seen that this circumstance does not affect the pattern of activation of any task.

Execution continues in mode `Slow` until time 10.0, when T1 issues a MCR to mode `Medium`. T2's waiting for the next release is aborted by the ATC and its next release is cancelled. Since T2 is not active in mode `Medium`, it is requeued from `Detect_Mode_Change` to `Wait_For_Active_Mode (Medium)` until a new MCR occurs. This was the situation of T3, who was waiting in the entry `Wait_For_Active_Mode (Slow)`. The mode change is evaluated and the task is requeued to `Wait_For_Active_Mode (Medium)` now, because it is not active in the current mode either.

Only task T1 is active in mode `Medium` and its first release in the new mode must be delayed 1.5 seconds, i.e., it must start again at time 11.5. Its priority is changed to 7, its mode notion is changed to `Medium` and its first release effectively occurs at 11.5 seconds. It then starts executing at a period of one second, T1's period for mode `Medium`.

A new mode change to `Slow` occurs at time 13.5. Tasks T1 and T2 are now rescheduled at time 16.5 and execute at the proper periods and priorities.

After some more mode changes, we move now to time 29.0, when T1 forces a mode change from `Slow` to `Fast`. This makes tasks T1 and T2 move to queue `Wait_For_Active_Mode (Fast)` after stopping their further releases. Task T3 is scheduled at *immediately* after the MCR and it starts executing t time 29.0, at the required priority 15 and period 500 ms.

```
Initialize called for task T3
Stopping further releases of task  3
EVALUATE_MC_COND: Requeuing task 3 to queue SLOW
Initialize called for task T2
Initialize called for task T1
T2's Code called at 1.016 I = 2001 * T2's Code finished*
T2's Code called at 2.012 I = 2002 * T2's Code finished*
T1's Code called at 2.012 I = 1001 * T1's Code finished*
T2's Code called at 3.012 I = 2003 * T2's Code finished*
T2's Code called at 4.012 I = 2004 * T2's Code finished*
T1's Code called at 4.012 I = 1002 MODE CHANGE to Slow
 * T1's Code finished*
T2's Code called at 5.012 I = 2005 * T2's Code finished*
T2's Code called at 6.008 I = 2006 * T2's Code finished*
T1's Code called at 6.008 I = 1003 * T1's Code finished*
T2's Code called at 7.011 I = 2007 * T2's Code finished*
T2's Code called at 8.012 I = 2008 * T2's Code finished*
T1's Code called at 8.012 I = 1004 * T1's Code finished*
T2's Code called at 9.008 I = 2009 * T2's Code finished*
T2's Code called at 10.000 I = 2010 * T2's Code finished*
T1's Code called at 10.008 I = 1005 MODE CHANGE to Medium
Stopping further releases of task  2
EVALUATE_MC_COND: Requeuing task 2 to queue MEDIUM
WAIT_FOR_ACTIVE_MODE: Re-requeuing task 3 to MEDIUM
 * T1's Code finished*
Stopping further releases of task  1
EVALUATE_MC_COND: Mode notion set to MEDIUM for task  1
Next activation of task 1 scheduled at 11.508
Changing task 1's prio to 7
Priority changed
T1's Code called at 11.519 I = 1006 * T1's Code finished*
T1's Code called at 12.519 I = 1007 * T1's Code finished*
T1's Code called at 13.519 I = 1008 MODE CHANGE to Slow
WAIT_FOR_ACTIVE_MODE: Re-requeuing task 3 to SLOW
Next activation of task 2 scheduled at 16.520
Changing task 2's prio to 10
Priority changed
 * T1's Code finished*
Stopping further releases of task  1
EVALUATE_MC_COND: Mode notion set to SLOW for task  1
Next activation of task 1 scheduled at 16.520
Next release adjusted for task 1 to time 16.520
Changing task 1's prio to 6
Priority changed
T2's Code called at 16.527 I = 2011 * T2's Code finished*
T1's Code called at 16.528 I = 1009 * T1's Code finished*
T2's Code called at 17.527 I = 2012 * T2's Code finished*
T2's Code called at 18.528 I = 2013 * T2's Code finished*
T1's Code called at 18.528 I = 1010 * T1's Code finished*
T2's Code called at 19.527 I = 2014 * T2's Code finished*
T2's Code called at 20.531 I = 2015 * T2's Code finished*
...
 -- Removed trace until time 29.0
...
T2's Code called at 29.050 I = 2018 * T2's Code finished*
T1's Code called at 29.051 I = 1016 MODE CHANGE to Fast
Stopping further releases of task  2
EVALUATE_MC_COND: Requeuing task 2 to queue FAST
Next activation of task 3 scheduled at 29.051
Changing task 3's prio to 15
Priority changed
T3's Code called at 29.052 I = 3001 *T3's Code finished*
* T1's Code finished*
Stopping further releases of task  1
EVALUATE_MC_COND: Requeuing task 1 to queue FAST
T3's Code called at 29.560 I = 3002 *T3's Code finished*
T3's Code called at 30.056 I = 3003 *T3's Code finished*
T3's Code called at 30.563 I = 3004 *T3's Code finished*
...
```

[3]We will omit all decimals but one in the text.

5 Conclusions and Future Work

The goal of this paper was to check the feasibility of modifying the original framework to include the notion of operating modes and mode changes. The result is, in this sense, satisfactory as shown in section 4.

The user of the framework must provide a very simple package with the list of modes and either derive her own mode manager from the synchronized interface `Mode_Manager_Type`, or use an already implemented mode manager like the one we have shown.

We have worked on a simplified version of the original framework to just produce this proposal, excluding tasks with event notification and execution-time servers. We foresee a number of issues to be tackled as future work:

- Implement other mode managers.
- Consolidate the definition of the `Mode_Manager_Type` interface.
- Add a matrix of allowed transitions and an associated exception for illegal MCRs. This could be placed in the user-defined package `List_Of_Modes`.
- Extend multi-moded tasks to include a mode change handler, in case the task needs to perform initialization when entering a new mode.
- Redefine the hierarchy of real-time abstractions in the framework, so that multi-moded tasks can coexist with mode-independent tasks.
- Incorporate tasks with event notification and tasks with execution-time servers in the hierarchy.

References

[1] A. Burns and A. Wellings. *Concurrent and Real-Time Programming in Ada*. Cambridge University Press, 2007.

[2] M. González-Harbour and J. J. Gutiérrez. Session: Programming patterns and libraries. *Ada User Journal*, 29(1):44–46, March 2008.

[3] P. Pedro. *Schedulability of Mode Changes in Flexible Real-Time Distributed Systems*. Ph.D. thesis, University of York, Department of Computer Science, 1999.

[4] J. Real and A. Crespo. Mode Change Protocols for Real-Time Systems: A Survey and a new Proposal. *Real-Time Systems*, 26(2):161–197, March 2004.

[5] A. J. Wellings and A. Burns. A Framework for Real-Time Utilities for Ada 2005. *Ada Letters*, XXVII(2), August 2007.

[6] A. J. Wellings and A. Burns. A Framework for Real-Time Utilities for Ada 2005. *Ada User Journal*, 28(1):47–53, March 2008.

A Listings

```
with List_Of_Modes; use List_Of_Modes;
with System;        use System;
with Ada.Real_Time; use Ada.Real_Time;
with Ada.Task_Identification;

package Real_Time_Task_State is

   type State is interface;
   procedure Initialize(S: in out State) is abstract;
   procedure Code(S: in out State) is abstract;
   procedure Deadline_Miss(S: in out State) is abstract;
   procedure Overrun(S: in out State) is abstract;
   type Any_State is access all State'Class;

   type Task_State is abstract new State with private;
   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;

   function Get_Relative_Deadline (S: in Task_State; M: in Operating_Modes := Initial_Mode) return Time_Span;
   procedure Set_Relative_Deadline(S: in out Task_State; Deadline : in Time_Span; M: in Operating_Modes := Initial_Mode);
   function Get_Execution_Time(S: in Task_State; M: in Operating_Modes := Initial_Mode) return Time_Span;
   procedure Set_Execution_Time(S: in out Task_State; Execution_Time : in Time_Span; M: in Operating_Modes := Initial_Mode);
   function Get_Priority(S: in Task_State;  M: in Operating_Modes := Initial_Mode) return Priority;
   procedure Set_Priority(S: in out Task_State; Pri : in Priority; M: in Operating_Modes := Initial_Mode);
   function Is_Active(S: in Task_State; M: in Operating_Modes := Initial_Mode) return Boolean;
   procedure Set_Active(S: in out Task_State; M: in Operating_Modes := Initial_Mode);
   procedure Set_Inactive(S: in out Task_State;  M: in Operating_Modes := Initial_Mode);
   function Get_Transition_Offset(S: in Task_State; From: in Operating_Modes := Initial_Mode;
                               To: in Operating_Modes := Initial_Mode) return Time_Span;
   procedure Set_Transition_Offset(S: in out Task_State; From: in Operating_Modes := Initial_Mode;
                                To: in Operating_Modes := Initial_Mode; Offset: in Time_Span);

   function Get_Mode_Notion (S: in Task_State) return Operating_Modes;
   procedure Set_Mode_Notion (S: in out Task_State; M : in Operating_Modes);
   function Get_Task_Id(S: in Task_State) return Ada.Task_Identification.Task_Id;

   type Any_Task_State is access all Task_State'Class;

private
   type Relative_Times is array (Operating_Modes) of Time_Span;
   type Priorities is array (Operating_Modes) of Priority;
   type Transition_Offsets is array (Operating_Modes,Operating_Modes) of Time_Span;
   type Running_Modes is array (Operating_Modes) of Boolean;

   type Task_State is abstract new State with record
      Relative_Deadline : Relative_Times := (others => Time_Span_Last);
      Pri               : Priorities := (others => Default_Priority);
      Execution_Time    : Relative_Times := (others => Time_Span_Last);
      Runs_In           : Running_Modes := (others => False);
      Transition_Offset : Transition_Offsets := (others => (others => Milliseconds(0)));
      Current_Mode      : Operating_Modes := Initial_Mode;
      Task_Id           : Ada.Task_Identification.Task_Id :=  Ada.Task_Identification.Null_Task_Id;
   end record;

end Real_Time_Task_State;
```

Figure 3. The new package `Real_Time_Task_State`

```
with Modes;                   use Modes;
with Ada.Dynamic_Priorities;  use Ada.Dynamic_Priorities;
with Ada.Real_Time;           use Ada.Real_Time;

package body Real_Time_Tasks is

   task body Simple_Real_Time_Task is
      Next_Release_Time : Time;
      Mode_Has_Changed : Boolean := False;
   begin
      S.Initialize;
      loop
         select
            Mode_Manager.Detect_Mode_Change (S,R);
            Mode_Has_Changed := True;
            -- This way of calculating the next activation depends on the
            -- mode change protocol.
            Next_Release_Time := Mode_Manager.Time_Of_Last_MCR +
                S.Get_Transition_Offset(Mode_Manager.Previous_Mode,S.Get_Mode_Notion);
            R.Resume_Releasing(Next_Release_Time);
         then abort
            R.Wait_For_Next_Release;
         end select;
         if Mode_Has_Changed then -- To avoid spurious release after mode change
                                  -- and keep the task non-abortable upon mode change
            Set_Priority(S.Get_Priority(S.Get_Mode_Notion));
            Mode_Has_Changed := False;
         else
            S.Code;
         end if;
      end loop;

   end Simple_Real_Time_Task;

end Real_Time_Tasks;
```

Figure 4. Implementation of a multi-moded periodic task

```
with System;
with Ada.Real_Time;        use Ada.Real_Time;
with Epoch_Support;        use Epoch_Support;
with Real_Time_Task_State; use Real_Time_Task_State;

package Mode_Managers.My_Protocol is

   protected type My_Mode_Manager_Type is new Mode_Manager_Type with
         overriding procedure Change_To (New_Mode : in Operating_Modes);
         overriding entry Detect_Mode_Change (S : Any_Task_State; R : Any_Release_Mechanism);
         overriding function Current_Mode return Operating_Modes;
         overriding function Previous_Mode return Operating_Modes;
         overriding function Time_Of_Last_MCR return Time;
         pragma Priority(System.Priority'Last);
   private
      entry Evaluate_Mode_Change_Condition (Operating_Modes) (S : Any_Task_State; R : Any_Release_Mechanism);
      entry Wait_For_Active_Mode (Operating_Modes) (S : Any_Task_State; R : Any_Release_Mechanism);
      The_Current_Mode     : Operating_Modes := Initial_Mode;
      The_Previous_Mode    : Operating_Modes := Initial_Mode;
      The_Time_Of_Last_MCR : Time := Epoch.Get_Start_Time;
   end My_Mode_Manager_Type;

end Mode_Managers.My_Protocol;
```

Figure 5. Definition of the mode manager `My_Mode_Manager_Type` **in** `Mode_Managers.My_Protocol`

```
package body Mode_Managers.My_Protocol is

   protected body My_Mode_Manager_Type is

      procedure Change_To (New_Mode : Operating_Modes) is
      begin
         The_Time_Of_Last_MCR := Clock;
         The_Previous_Mode := The_Current_Mode;
         The_Current_Mode := New_Mode;
      end Change_To;

      function Current_Mode return Operating_Modes is
      begin
         return The_Current_Mode;
      end Current_Mode;

      function Previous_Mode return Operating_Modes is
      begin
         return The_Previous_Mode;
      end Previous_Mode;

      function Time_Of_Last_MCR return Time is
      begin
         return The_Time_Of_Last_MCR;
      end Time_Of_Last_MCR;

      entry Detect_Mode_Change (S : Any_Task_State; R : Any_Release_Mechanism)
      when True is
      begin
         requeue Evaluate_Mode_Change_Condition (S.Get_Mode_Notion) with abort;
      end Detect_Mode_Change;

      entry Evaluate_Mode_Change_Condition (for M in Operating_Modes) (S : Any_Task_State; R : Any_Release_Mechanism)
      when M /= The_Current_Mode is
      begin
         R.Stop_Releasing;
         if not S.Is_Active(The_Current_Mode) then
            requeue Wait_For_Active_Mode (The_Current_Mode);
         else
            S.Set_Mode_Notion(The_Current_Mode);
         end if;
      end Evaluate_Mode_Change_Condition;

      entry Wait_For_Active_Mode (for M in Operating_Modes) (S : Any_Task_State; R: Any_Release_Mechanism)
      when M /= The_Current_Mode is
      begin
         if not S.Is_Active(The_Current_Mode) then
            requeue Wait_For_Active_Mode (The_Current_Mode);
         else
            S.Set_Mode_Notion(The_Current_Mode);
         end if;
      end Wait_For_Active_Mode;

   end My_Mode_Manager_Type;

end Mode_Managers.My_Protocol;
```

Figure 6. Body of the mode manager

```
with Real_Time_Task_State.Periodic; use Real_Time_Task_State.Periodic;
with Release_Mechanisms.Periodic;   use Release_Mechanisms.Periodic;
with Real_Time_Tasks;               use Real_Time_Tasks;

package Periodic_Test_MM  is

   type My_State_1 is new Periodic_Task_State with
      record
         I : Integer;
      end record;
  procedure Initialize(S: in out My_State_1);
  procedure Code (S: in out My_State_1);

   type My_State_2 is new Periodic_Task_State with
      record
         I : Integer;
      end record;
  procedure Initialize(S: in out My_State_2);
  procedure Code (S: in out My_State_2);

   type My_State_3 is new Periodic_Task_State with
      record
         I : Integer;
      end record;
  procedure Initialize(S: in out My_State_3);
  procedure Code (S: in out My_State_3);

   S1 : aliased My_State_1;
   S2 : aliased My_State_2;
   S3 : aliased My_State_3;

   M1 : aliased Periodic_Release(S1'Access);
   M2 : aliased Periodic_Release(S2'Access);
   M3 : aliased Periodic_Release(S3'Access);

   T1 : Simple_Real_Time_Task(S1'Access,M1'Access,6);
   T2 : Simple_Real_Time_Task(S2'Access,M2'Access,10);
   T3 : Simple_Real_Time_Task(S3'Access,M3'Access,15);

end Periodic_Test_MM;
```

Figure 7. Specification of the example system

```
with Ada.Dynamic_Priorities, Modes, List_Of_Modes, Mode_Managers.My_Protocol, Ada.Real_Time, Ada.Text_IO;
use  Ada.Dynamic_Priorities, Modes, List_Of_Modes, Mode_Managers.My_Protocol, Ada.Real_Time, Ada.Text_IO;

package body Periodic_Test_MM is

  procedure Initialize(S: in out My_State_1) is
   begin
      S.Set_Active(Slow);                    S.Set_Active(Medium);
      S.Set_Period(Milliseconds(2000),Slow); S.Set_Period(Milliseconds(1000),Medium);
      S.Set_Transition_Offset(Slow,Medium,Milliseconds(1500));
      S.Set_Transition_Offset(Medium,Slow,Milliseconds(3000));
      S.Set_Priority(6,Slow);                S.Set_Priority(7,Medium);
      Set_Priority(S.Get_Priority(S.Get_Mode_Notion));

      S.I := 0;
      Put_Line("Initialize called for task T1");
  end Initialize;

  procedure Code (S: in out My_State_1) is
  begin
    S.I := S.I + 1;
    Put("T1 Code called I ="); Put_Line(Integer'Image(S.I));
      if S.I mod 3 = 0 then
         if S.I mod 6 = 0 then
            Mode_Manager.Change_To(Slow);
         else
            Mode_Manager.Change_To(Medium);
         end if;
      else
         if S.I mod 8 = 0 then
            Mode_Manager.Change_To(Fast);
         end if;
      end if;
  end Code;

   procedure Initialize(S: in out My_State_2) is
   begin
      S.Set_Active(Slow);
      S.Set_Period(Milliseconds(1000),Slow);
      S.Set_Transition_Offset(Medium,Slow,Milliseconds(3000));
      S.Set_Priority(10,Slow);
      Set_Priority(10);

      S.I := 1_000;
      Put_Line("Initialize called for task T2 ");
  end Initialize;

  procedure Code (S: in out My_State_2) is
  begin
    S.I := S.I + 1;
    Put("T2 Code called I =");
    Put_Line(Integer'Image(S.I));
  end Code;

  procedure Initialize(S: in out My_State_3) is
   begin
      S.Set_Active(Fast);
      S.Set_Period(Milliseconds(500),Fast);
      S.Set_Transition_Offset(Medium,Fast,Milliseconds(1000));
      S.Set_Priority(15,Fast);
      S.Set_Mode_Notion(Fast); -- Any mode in which the task participates
                               -- Necessary for tasks inactive in initial mode
      Set_Priority(15);

      S.I := 10_000;
      Put_Line("Initialize called for task T3");
  end Initialize;

  procedure Code (S: in out My_State_3) is
  begin
    S.I := S.I + 1;
    Put("Code called I ="); Put(Integer'Image(S.I));
      if S.I mod 5 = 0 then
         Mode_Manager.Change_To(Medium);
       end if;
  end Code;

end Periodic_Test_MM;
```

Figure 8. Body of the example system