Towards an Integrated Framework for Multiprocessor, Multimoded Real-Time Applications *

Sergio Sáez, Jorge Real & Alfons Crespo
Instituto de Automática e Informática Industrial,
Universitat Politècnica de València,
Camino de vera, s/n, 46022 Valencia, Spain,

E-mail: {ssaez,jorge,alfons}@disca.upv.es

Abstract

In this paper we propose an approach for building real-time systems under a combination of requirements: specification and handling of operating modes and mode changes; implementation on top of a multiprocessor platform; integration of both aspects within a common framework; and connection with schedulability analysis procedures.

The proposed approach uses finite state machines to describe operating modes and transitions, and a framework of real-time utilities that implements the required behaviour in Ada 2012. Automatic code generation plays an important role: the system is derived from the functional and timing specification, and implemented according to the abstractions provided by the framework. Response time analysis enables assessing the schedulability of the different operating modes and the transitions between modes.

Keywords: Real-Time Framework, Mode Changes, Multiprocessor Scheduling, Ada 2012.

1 Introduction

This paper continues a series of developments around a framework for real-time utilities in Ada. The aim of such effort is to provide a set of high-level abstractions to ease the development of real-time systems, by using the very convenient but low-level facilities provided by Ada. The first version of that framework was introduced in [1]; a second version was proposed for extending the original framework to multimoded systems [2], and then the framework was substantially redesigned to accommodate execution on multiprocessor platforms [3], with an incipient support for modes and mode changes. As a continuation of that work, the framework version used in this paper integrates and extends the support for both multiprocessor and multimoded real-time systems, making use of Ada 2012.

We consider task-partitioned scheduling across processors, where tasks are statically pre-allocated to processors. We will however enrich this model by considering also job-partitioning for selected tasks and task migration between modes.

Another contribution of this paper is the integration with a tool for specifying the behavioral aspects of the system, in terms of operating modes and transitions among them. From this description, an additional code generation tool produces code for handling the conditions that lead to mode changes at run time, and implement the adequate handlers at system level (global mode-change handler) and at the task level (local handlers). At the global level, tasks’ attributes such as period, deadline and priorities are adequately changed when there is a mode change. At the local level, task-specific actions can be handled, such as readjusting ceiling priorities of shared protected objects.

*This work was partially supported by the Vicerrectorado de Investigación of the Universitat Politècnica de València under grant PAID-06-10-2397
The paper illustrates the design process and implementation details by using an example system from the very beginning. Section 2 presents the example system and underlying hardware platform. Section 3 briefly discusses the results of the schedulability analysis of the system, both in the steady state (each of the modes) and during mode transitions. Section 4 shows the specification process and support. Section 5 discusses the implementation within the framework. Finally, Section 6 concludes the paper.

2 Example system

This Section describes an example system to illustrate the concepts introduced in the rest of the paper. We have chosen to use a real example system, for we believe it serves better the purpose of explaining our design choices. However, we have omitted a number of details that would only make the example harder to follow. Although all tasks in the example are periodic, the concepts herein are also applicable to sporadic tasks with a bounded pattern for the arrival of the activation event, i.e., tasks with a bounded minimum inter-arrival time.

2.1 Functional description

The example system is in charge of classifying different mechanical pieces into two categories: cylinders and cubes. Figure 1 shows the example plant. Pieces are supplied to the system by means of a conveyor belt. A video camera is located on top of the conveyor belt and takes images of the first section of the belt. These images therefore reflect the input load to the system. According to that input, up to two manipulator robots will pick the pieces from the belt and place them separately depending on their type (cylinder or cube). In figure 1, this is represented by other two conveyor belts that we will not consider as a part of the system. There is also a console screen that shows information about the process.

Figure 1. View of the example plant

Figure 2. Software elements of the example system
After considering the physical elements of the system, figure 2 shows the software elements and their interconnections. A two-stage process (*segmentation* and *recognition*) analyzes the images captured by the camera. The segmentation part simply detects the number of pieces and their position in each image frame. The output from the segmentation stage is inserted in the *Image Buffer*. After segmentation, the recognition stage completes the analysis of the images by determining the type of each piece and their exact orientation, so that the robots can properly catch them from the belt. The output from the recognition stage is placed in the *ToDo Buffer*, from where they are then collected by one or two robots, controlled by tasks *Robot_0* and *Robot_1*. Each time a robot removes one piece from the belt, its corresponding robot task adds the related information to the *Done Buffer*.

The *Graph* task extracts elements from the *Done Buffer* and displays status information on the console screen (e.g., number and type of pieces processed). Finally, the *Belt* task controls the belt speed. This task is independent from the rest of tasks, in the sense that it does not need to exchange information with them.

### 2.2 Operating modes

The flow of pieces is variable, and so is the number of pieces that remain to be removed from the belt. In order to adapt to this input variability, and to save energy and resources, three operating modes are defined for the system. The current operating mode depends on the amount of pieces that need to be removed from the belt at a particular time interval:

**Normal Mode** During this mode, the number of pieces on the belt is within the range \(1..\text{Threshold}\). In this situation, the system is able to process all the pieces on the belt by using one single robot. The second robot is kept in a standby state in order to save energy. The belt advances at normal speed.

**Overload mode** When the number of pieces in a frame is greater than the threshold, the system operates in overload mode. In this mode, both robots collaborate to remove pieces from the belt. When the amount of pieces in scope is again within the threshold, the system will switch back to normal mode. By incorporating the second robot in the overload mode, we can keep the belt running at normal speed.

**Fetch mode** When there are no pieces to be processed on the belt (the input flow has temporarily ceased), both robots standby and the belt moves at fast speed in order to fetch pieces at the beginning of the belt as fast as possible. There is no need for the recognition process to run in this mode, since there are no pieces to recognize. But we still need to run the segmentation process in order to detect the arrival of new pieces. The fetch mode is abandoned when a non-empty image frame is detected in the segmentation stage. The details about how this mode change is processed are given in Section 4.

### 2.3 Hardware platform and software workload model

We shall assume that the hardware platform is a two-core processor, with the two cores identified as CPU0 and CPU1. A low number of cores keeps the example simple, while it allows us to demonstrate the ability of the proposed framework to take advantage of multi-core processors.

The distribution of software tasks among processors depends on the operating mode. Table 1 shows the use of the two cores in the three defined operating modes. Note that CPU1 is in standby in modes normal and fetch, where only CPU0 is active. The tasks’ names in table 1 identify the activities described in Section 2.1.

In overload mode, when the number of pieces to process is large, both CPUs are used for segmentation and recognition, and each CPU controls one robot. Hence we double the capacity for the system to cope with the input overload.

Figure 3 gives the details of the workload in overload mode, which deserves further explanation. Note that both CPUs perform the same sequence of processing steps, with the difference that CPU0 executes the graph task while CPU1 controls the belt. Both CPUs execute the segmentation step at the same rate. By using an appropriate offset (as shown later in this Section), we will alternate the execution of the pair segmentation-recognition in both cores, so that each CPU processes a different image frame. The segmentation task is unique, but it is scheduled to execute consecutive instances (jobs) in alternate CPUs: it is *job-partitioned*.

The image buffer is split in two local shared objects, *Image Buffer_0* and *Image Buffer_1*. Each instance of the segmentation task uses the buffer corresponding to its current CPU.
Table 1. Distribution of tasks across CPUs and modes

<table>
<thead>
<tr>
<th>Normal mode</th>
<th>Overload mode</th>
<th>Fetch mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU0</td>
<td>CPU1</td>
<td>CPU0</td>
</tr>
<tr>
<td>Segmentation Recognition Robot_0 Belt Graph</td>
<td>Segmentation Recognition Robot_0 Graph</td>
<td>Segmentation Recognition Robot_1 Belt Graph</td>
</tr>
</tbody>
</table>

Figure 3. Workload details in overload mode

Figure 4 compares the execution of the segmentation-recognition process in modes normal and overload. In normal mode, there is time enough for segmentation and recognition to complete within their period, before a new image frame is taken. But when the system is in overload mode, the recognition process takes longer to execute, because there are more pieces to identify. In particular, it can take longer than the frame capture period, and hence one recognition stage could overlap with the next one. Therefore we cannot just job-partition the recognition task. On the contrary, we will use two different tasks, identified in figure 3 as Recognition_0 and Recognition_1. The overlapping of the end of recognition with the processing of the next image frame is therefore solved by using the two cores in parallel.

By using an initial offset for the pair segmentation-recognition when we enter overload in CPU1, we are shifting this pair of tasks with respect to their analogous tasks on CPU0. By doing so, each CPU is processing a different image frame and we keep the system capacity to cope with the overload without having to slow the belt.

Recognition tasks suffer from input jitter: they perform the recognition stage on the latest image taken by

Figure 4. Segmentation and recognition in modes normal and overload. In overload mode, segmentation (S) is job-partitioned and recognition (R) has a larger WCET.
their corresponding segmentation task. We will ensure this relationship by setting a deadline for segmentation equivalent to the maximum input jitter for recognition, and we will use the results of the schedulability analysis to verify that recognition always uses fresh data.

Both recognition tasks share the *ToDo_Buffer*, which is a global resource since it is used from tasks running on both processors.

*Robot_0* and *Robot_1* execute in CPU0 and CPU1, respectively. They control the corresponding robots. Both tasks share the common, global resource *Done_Buffer*, where they insert information about pieces already processed. They also share the *ToDo_Buffer* among them and with the recognition tasks.

The task *Graph* collects information from the *Done_Buffer* to display statistics on the screen. The task *Belt*, in charge of keeping the belt speed adequate to the current mode, runs on CPU1 in overload. In modes normal and fetch, *Belt* runs in CPU0. This task serves us to illustrate task migration between modes.

### 3 Schedulability Analysis

#### 3.1 Steady-state analysis

Table 2 shows the tasks’ timing parameters and the worst-case response time analysis of the different operating modes, considered in isolation. We refer to this as the *steady-state* analysis, because it does not consider transitions between modes. All tasks are periodic and the time units used here are abstract. For each mode and CPU, the table shows the tasks’ worst-case execution time (*C*), period (*P*), deadline (*D*), and input jitter (*J*), as well as the calculated worst-case response time (*R*). This response time has been obtained using the classical response time analysis equations [4, 5], with higher priorities assigned to shorter deadlines, and assuming blocking times of 2 time units for all shared resources. Note that all tasks are schedulable in the steady state, that is, in the absence of mode changes: all worst-case response times are below their respective deadlines. Table 2 also shows the utilization ratios for each CPU and mode. Note that the overload mode would not be schedulable in a single CPU, since the total utilization is above 100%.

Segmentation runs at different periods. A short period of 25 units in fetch mode, since the belt moves at fast speed during fetch. A medium period of 50 units in normal mode, that accommodates the normal speed of the belt in this mode. And a larger period of 100 units in overload. In overload mode however, the segmentation task runs in both processors. Hence the effective rate of segmentation is the same in modes normal and overload. In mode overload, an initial offset of 50 time units for segmentation in CPU1 will ensure that segmentation runs once
every 50 time units in one or the other CPU – see the mode-change analyses in Section 3.2.

Recognition requires more processing time in overload than in normal mode (40 vs. 20 units), since there are more pieces to process during an overload. The recognition task is modeled with two different task descriptions: we use Recognition to describe the task in mode normal, with C = 20, and Recognition_0 and Recognition_1 for the overload mode, with C = 40. Note that this is just an analysis artifact: our model for mode-change analysis allows changes in all tasks’ parameters, except in C. There are several reasons that justify this approach [6].

The input jitter of 10 units for recognition tasks models the required behaviour that recognition always uses the freshest image pre-processed by segmentation. This input jitter is set equal to the deadline for segmentation tasks. In practical terms, the ImageBuffers behave like a one-item stack, written with push and read with a blocking pop. So recognition is blocked until there is a new item in the ImageBuffer.

3.2 Transition analyses

There are 5 possible transitions in the example system. They comprise changes in both directions between normal and fetch (four mode changes), and one more from fetch to overload. The only transition excluded is overload to fetch. This will be explained in more detail in Section 4. The mode-change analysis must be applied to both CPUs, hence there is a total of 5 transitions × 2 CPUs = 10 analyses to consider. Some of them are however trivial. For example, all mode switches in CPU1 are schedulable because there is only one active mode in that CPU: there are no old-mode tasks when switching from normal or fetch to overload; there are no new-mode tasks in a switch from overload to normal; and there are no tasks at all involved in switches between normal and fetch in CPU1. Hence all transitions in CPU1 are guaranteed by the steady-state analysis shown in Table 2.

We analyze the schedulability of transitions using the mode-change response time analysis proposed in [6]. Our tool analyzes the transition and, if it is not schedulable, it then finds appropriate offsets for new-mode tasks so that no deadlines are missed in the mode switch. For tasks with a changing period, this offset is relative to the time when the mode change request occurs. For tasks that keep their activation pace across modes, the calculated offset is relative to the first activation of the task in the new mode. We have slightly adapted the tool to enable setting an initial offset in the new mode, since we needed that for our segmentation and recognition tasks in CPU1 in mode overload.

There is no space available here for showing the results of all the mode-change analyses in detail. We will just note that all transitions proved schedulable after applying appropriate offsets when needed.

4 Operating modes specification

Section 2.2 has described the possible operating modes of the system and the conditions to move from one mode to another. Two main components will be involved in these mode changes: the mode manager, that detects the mode-change conditions and triggers the mode-change process; and the mode changer, that performs the mode-change process updating the attributes of the tasks involved in a given change. There are two different approaches to design these components:

Distributed The logic to detect the mode-change conditions or to update the task attributes is distributed across the system tasks, i.e., the mode-change conditions are detected in the tasks’ code, from where the mode-change request is triggered. Each task is also responsible of changing its own attributes for the new mode under the request of a mode changer.

Centralized The logic to detect the mode-change conditions is centralized in a mode manager component that receives the relevant system events and decides when to trigger the mode-change process. A centralized mode changer processes the mode-change request by updating the attributes of the involved tasks directly.

The design proposed in this work is based on a centralized mode manager that detects all mode-change conditions and sends a mode-change request to a minimalist mode changer when appropriated. The mode changer informs the involved tasks about the mode change request to perform the task attributes updates in a distributed manner. This section deals with the specification and design of the mode manager, while the mode-change process and the mode changer details are explained in section 5.
4.1 Mode Manager specification

A centralized mode manager has to maintain a global state of system variables that are related with the current operating mode, and to trigger a mode-change request when certain conditions hold. The updating of this state is performed when certain system events occurs, e.g., a piece has been removed from the belt. This behavioral pattern adequately matches the formalism defined by Finite State Machines (FSM) [7, 8]. In the proposed framework, the mode manager is specified by means of a detailed behavioral FSM, and the implementation code is derived from this model using a modified version of the framework presented in [9].

4.2 UML Finite State Machine elements

UML Finite State Machines offer a broad set of elements to model the behavior of any system component. However, when this formalism is used to specify the behavior of a mode manager, only a small subset of these elements are really useful and some of them can have a slightly different semantics. This section describes some of the main FSM facilities that can be used to model complex mode managers.

First of all, although it seems obvious that system operating modes will correspond to states of the mode manager, not every state of the mode manager will represent a different operating mode. Therefore, states of the mode manager that do represent system operating modes have to be annotated somehow. The non-mode states can be used to facilitate the modeling of mode-change conditions. An example of this situation can be found in Figure 5. It shows the FSM that describes the operating modes of the example system and the transitions between them. Fetch, Normal and Overload states represent the system operating modes, while the rest of the states are used to model the internal behavior of the Overload state or to group common event responses as can be seen in the OR-state Working.

Since the current state of the FSM is always a simple state\(^1\), when OR-states are used to group event management, the only requirement is that each simple state was either an operating mode or it had an operating mode among its ancestors or superstates. This ensures that the mode manager is always in a valid operating mode while being in a stable state. The FSM that describes the example mode manager clearly fulfills this condition.

\(^1\)A state without any nested state
Although the rest of FSM elements can be used to specify the mode manager behavior, such as *timed* events, *entry* and *exit* actions, other constructs, such as *deferred* events or *do* activities, have a limited usefulness. In the case of deferred events, the fact that a given system event management will be postponed until the mode manager is in a different mode does not seem clearly useful. In the same way, *do* activities require that the mode manager is continuously executing a set of actions while staying in a given state. This behavior does not match the role of a system mode manager.

Finally, AND-states allow the FSM to be in more than one state simultaneously. The next subsection explains how this can be applied to the definition of a mode manager.

### 4.3 Specification of partial mode changes

AND-states can be used to simplify the number of states in a FSM, since they allow different aspects of the component behavior to evolve independently. However, if concurrent states correspond to different system operating modes, the semantics of *being in multiple operating modes* needs to be clarified.

If an AND-State is used to partition the task set among the concurrent regions, this allows the system designer to specify different operating modes for different subsets of tasks. In such a way, a mode change in a concurrent region will not alter the attributes of the task associated with the other concurrent regions. In the example shown in Figure 6, a mode change from mode *C* to *E* will not change the attributes of the tasks associated with modes *A* or *B*.

However, from the analysis point of view, if the task partitions share software or hardware resources, e.g. they are executed in the same set of CPUs, the mode change analysis will require to consider the system as a whole. Although the task attributes of the other partitions do not change, the response times can be affected due to the lack of execution isolation. On the other hand, if the tasks associated with each concurrent region are executed in a different set of CPUs or *Dispatching Domain* [10] and with no resources shared among the partitions, isolation ensures that mode change analysis can be performed independently for each concurrent region, thus reducing the number of transitions to be analyzed.

### 5 Implementation within the Real-Time Framework

This section deals with the implementation of the example application on top of the Real-Time Framework presented in [7], that has been extended in this work to support multimoded real-time applications. Next sections detail the design and the implementation of the main components.

#### 5.1 Mode manager

As mentioned above, the mode manager has been specified by means of a behavioral FSM. Although the resulting mode manager component could be an Active Object and the corresponding task be incorporated in the system analysis, a Passive Object is preferred to implement the proposed approach. Accordingly, an Ada protected object implements the centralized mode manager. The Ada specification of the mode manager is shown in Listing 1.
This code is automatically generated from the FSM specification in SCXML using an extended version of the tools presented in [9].

**Listing 1. Mode manager protected type**

```plaintext
protected type Example_System_Mode_Manager(Num_Tasks: Positive) is
    new Example_System_Mode_Manager_Interface with
    -- Registers a task in the mode manager
    procedure Add_Task (NM: Any_Notification_Mechanism;
                          TS: Any_Task_Sched_Interface);
    -- Init event to initialize the FSM
    procedure Init_Event;
    -- Send event: PiecesDetected
    procedure Pieces_Detected_Event (Num_Pieces: Natural);
    -- Send event: PieceRemoved
    procedure Piece_Removed_Event;

private
    -- Timeout event
    procedure Process_Timeout_Event (TE: in out Timing_Event);
    -- Completion event
    procedure Process_Completion_Event;
    -- FSM context
    Overload_Threshold : Natural := 8;
    Time_To_Process : TimeSpan := Seconds(5);
    Pending_Pieces : Natural := 0;
    Last_Capture : Natural := 0;
    -- FSM attributes
    The_Mode_Changer : Mode_Changer(Num_Tasks);

end Example_System_Mode_Manager;
```

With respect to the FSM context, that defines the scope of the variables and methods used in the FSM conditions and actions, there are two possible scenarios. If the FSM context is defined only by the state variables, as in the example system shown in Figure 5, these variables can be automatically placed in the mode manager protected type by the code generator and no extra context objects are required. However, if the FSM context also requires some procedures or functions, whose implementation has to be provided by the system designer, then a FSM context object has to be passed to the mode manager to complete the implementation of its behavior.

FSM events are directly mapped into protected procedures. Listing 2 shows how the PiecesDetected event is processed, how the decision pseudo-state is implemented and different mode change requests from Fetch to Normal or Overload, depending on the event information. The Process_Completion_Event procedure also shows how the activation and cancellation of a timed event is implemented in the WorkDecreased state by means of an Ada timing event.

**Listing 2. Mode manager PiecesDetected and Completion events handling**

```plaintext
procedure Pieces_Detected_Event (Num_Pieces: Natural) is
begin
    case Current_State is
    when Fetch_State =>
        -- Event action
        Last_Capture := Num_Pieces; Pending_Pieces := Num_Pieces;
        -- Decision pseudo-state
        if Last_Capture <= Overload_Threshold then
            Current_State := Normal_State; -- Target state
            Current_Mode := Normal_Mode; -- Target mode
            The_Mode_Changer.Change_To(Mode_Name'Pos(Normal_Mode));
        else
            Current_State := Still_Overloaded_State; -- Target state
```
Current_Mode := Overload_Mode; -- Target mode
The_Mode_Changer.Change_To(Mode_Name'Pos(Overload_Mode));
end if;
when Normal_State | Still_Overloaded_State | Work_Decreased_State =>
  -- Event action
  Last_Capture := Num_Pieces; Pending_Pieces := Pending_Pieces + Num_Pieces;
when others =>
  null; -- No transition
end case;
-- Check completion conditions
Process_Completion_Event;
end Pieces_Detected_Event_Event;

procedure Process_Completion_Event is
begin
  case Current_State is
    when Normal_State =>
      if Last_Capture > Overload_Threshold then
        Current_State := Still_Overloaded_State; -- Target state
        Current_Mode := Overload_Mode; -- Target mode
        The_Mode_Changer.Change_To(Mode_Name'Pos(Overload_Mode));
      elsif Pending_Pieces = 0 then
        Current_State := Fetch_State; -- Target state
        Current_Mode := Fetch_Mode; -- Target mode
        The_Mode_Changer.Change_To(Mode_Name'Pos(Fetch_Mode));
      end if;
    when Still_Overloaded_State =>
      if Last_Capture <= Overload_Threshold then
        -- Program timed event
        Timeout.Set_Handler (Time_To_Process, Process_Timeout_Event'Access);
        Current_State := Work_Decreased_State; -- Target state
      end if;
    when Work_Decreased_State =>
      if Last_Capture > Overload_Threshold then
        -- Cancel timed event
        Timeout.Cancel_Handler (Timeout_Cancelled);
        Current_State := Still_Overloaded_State; -- Target state
      end if;
    when others =>
      null; -- No transition
  end case;
end Process_Completion_Event;

5.2 Mode changer

Although the code of the mode changer is also generated from the system specification and the analysis results, the implementation is completely different. The mode changer is based on a minimalist Mode_Changer component that has been added to the Real-Time Framework and a set of automatically generated Mode_Change_Handlers distributed across the system tasks. When a mode change request is triggered, each task in the system receives a Mode_Change_Event, and it is the handler of this event who performs the mode change for that task in two phases. A first step, executed at the task priority, establishes the new task attributes, activates the scheduling mechanisms required for the new mode and updates the ceilings of shared resources. The second step effectively changes the task attributes within a task-specific protected procedure.

Listing 3 shows a fragment of the Mode_Change_Handler that implements the change from Normal to Overload for the segmentation task. The first part activates the Job-Partitioning mechanism in the Overload mode, and the second part establishes the specific transition offsets and updates ceilings. Finally, the activation of the second
step to be executed within the Release_Mechanism is shown.

### Listing 3. Mode-change handler for the Segmentation task

```pascal
procedure Mode_Change_Handler (Sched: in out Segmentation_Sched_Type;
    MCR: in Time;
    Old_Mode, New_Mode: in Natural) is

    Current_Mode: Mode_Name := Mode_Name'Val(Old_Mode);
    Target_Mode: Mode_Name := Mode_Name'Val(New_Mode);
    Must_Change: Boolean;

begin
    case Target_Mode is
        ... when Overload =>
            Sched.Job_Partition_CE.Set_Num_Sched_Sets(2);
            Sched.Job_Partition_CE.Set_Sched_Set(1,
                Segmentation_Sched_Attrib_Overload_Mode(1));
            Sched.Job_Partition_CE.Set_Sched_Set(2,
                Segmentation_Sched_Attrib_Overload_Mode(2));
            Sched.RM.Set_Control(Job_Partition_Id, Sched.Job_Partition_CO);
            Sched.SA_Base.all := Segmentation_Sched_Attrib_Overload_Mode(1);
            -- Transition dependent code
            case Current_Mode is
                when Fetch | Normal =>
                    Sched.SA_Base.Set_Offset(Milliseconds(4));
                    -- Ceiling updates for mode Overload
                    Example_System_Resources.Image_Buffer_0.Update_Ceiling(
                        Image_Buffer_0_Ceil_On_Overload_Mode);
                when others =>
                    null;
            end case;
        when others =>
            Sched.SA_Base.Set_Activity_Flag(False);
    end case;
    -- Activate second step
    Sched.Mode_Change_CO.Activate_Trigger;
end Mode_Change_Handler;
```

### 5.3 Shared objects

As shown in Listing 3, in order to support automatic updates of the ceilings during the mode-change process, the system shared resources have to implement the new Shared_Resource_Interface interface provided by the Real-Time Framework, to ensure that the Update_Ceiling procedure will be available. Also the exact package and object name have to be provided by the system designer to allow the code generation tool to produce the necessary mode-change code.

In the case of global shared resources, the mode change analysis tool cannot easily determine which task is the proper one to update the shared resource ceiling, as the execution order in multiple CPUs cannot be reliably established. In this case, the Shared_Resource_Interface is not enough and a simple helper object is provided for each global resource to determine the last task to process the mode-change event at run-time. Listing 4 shows how this helper object can be used to update global resource ceilings during a mode change.

### Listing 4. Global shared resources support

```pascal
-- Ceil updates for mode Normal
To_Do_Buffer_Helper.Must_Change_Ceiling(To_Do_Buffer_Num_Tasks_On_Overload_Mode,
    To_Do_Buffer_Ceil_On_Normal_Mode,
    Must_Change);
```
if Must_Change then
Example_System.Resources.ToDo_Buffer_Update_Ceiling(
    ToDo_Buffer_Ceil_On_Normal_Mode);
end if;

6 Conclusions

This paper has discussed the current state of a framework of real-time utilities in Ada 2012 for multiprocessor platforms, and its integration with a specification model and a code generation tool for multimoded real-time systems. The main contributions are the update of the framework to support modes on multiprocessors and the specification and code generation tools, connected with the results of the schedulability analysis.

The process here described favors separation of concerns between the implementation of tasks and the logic of handling mode changes.

References