Adding Multiprocessor and Mode Change Support to the Ada Real-Time Framework

Sergio Sáez, Jorge Real, and Alfons Crespo
Universitat Politécnica de Valéncia, Spain
{ssaez,jorge,alfons}@disca.upv.es

Abstract

Based on a previous proposal of an Ada 2005 framework of real-time utilities, this paper deals with the extension of that framework to include support for multiprocessor platforms and multiple operating modes and mode changes. The design of the proposed framework is also intended to be amenable to automatic code generation.

1. Introduction

One of the topics discussed at the 13th International Real-Time Ada Workshop was a proposal from Wellings and Burns [17, 18] to develop a framework of real-time utilities. The aim of that proposal was to provide a set of high-level abstractions to ease the development of real-time systems, taking advantage of facilities included in Ada 2005 such as timers and CPU timing mechanisms, and the integration of object oriented and concurrent programming. These new facilities of Ada 2005 are low-level abstractions like setting a timer, synchronizing concurrent tasks, or passing data among them. They are indeed 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 typically 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 increasingly being executed on multiprocessors,
- may execute in several modes of operation, characterized by performing different sets of activities under different timing requirements.

The proposal from Wellings and Burns (hereafter *the original framework*) was a first step in the direction of implementing such a library of real-time utilities. It addressed some of the features above, namely: the flexibility 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 [9].

This paper proposes extensions to the original framework in two aspects: (i) adapting the framework to multiprocessor platforms and (ii) defining modes of operation and providing the mechanisms to enable mode changes.

The paper is organized as follows: section 2 defines the context by summarizing the main features of the original framework. Section 3, and its subsections, enumerate the requirements we consider necessary for multiprocessor and multimode applications, and our idea about the design process to develop an application based on the framework. Section 4 describes the framework proposal. The flow of events and handlers in two scenarios (deadline miss and mode change) are described in Section 5. Finally, Section 6 gives our conclusions and points out pending issues.

2. Original Framework

The framework proposed in [17, 18] is explained in those publications and, to a larger extent, in chapter 16 of [6]. Therefore, only a brief overview of the original framework is given here. The main goal of the original framework is to provide a reusable set of 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,
- mechanisms to manage deadline misses and execution-time overruns at run time, and
- the possibility to limit the amount of CPU time devoted to tasks by means of execution-time servers.

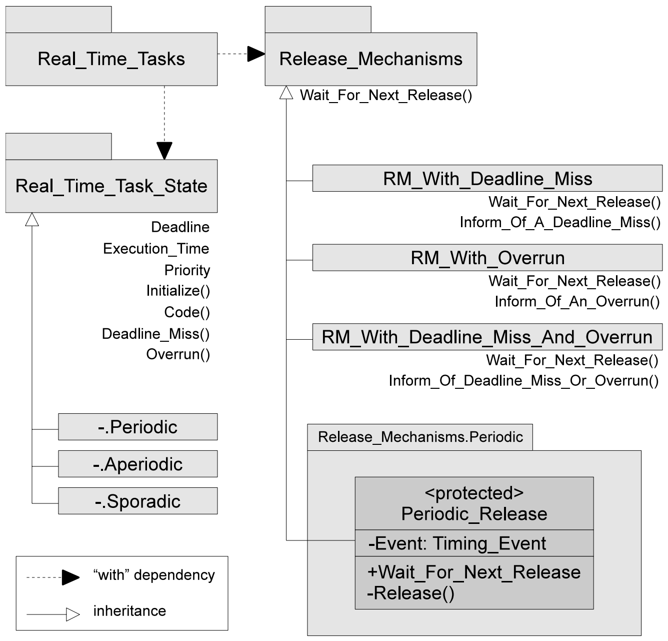

Figure 1. Top-level packages of the original framework

The top-level packages of the original framework are depicted[1] 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

[1]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.

the periodic release mechanism extension, implemented by means of the protected interface `Periodic_Release`. 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.

Since the detection and notification of relevant events is optional and orthogonal with respect to the kind of release mechanism selected for a task, the original framework needs to provide four different classes per release mechanism ($2^{\text{nr of events}} \times$ nr of release mechanisms). Namely, the original framework provides the following release mechanisms for periodic tasks: `Periodic_Release`, `Periodic_Release_With_Deadline_Miss`, `Periodic_Release_With_Overrun`, and `Periodic_Release_With_Deadline_Miss_And_Overrun`. Then the same amount of variants for sporadic tasks. The implementation of multiprocessor scheduling approaches (see section 3.2) requires an exponentially increased number of variants to cope with all the possible events to be handled. We consider that this is an important drawback for extensions to the original framework.

We must point out that the original framework was not aimed at targeting multiprocessor platforms, nor was it prepared to support modes and mode changes. Although we shall enumerate the requirements for these two cases in the next section, we briefly note here that, in the original framework, tasks (and jobs[2]) are not allowed to migrate through CPUs. We also note that, in the original framework, any event such as a deadline miss or execution time overrun will cause the task to be aborted and execute the corresponding handler. While this behavior may be appropriate for these two particular events, it is not flexible enough to accommodate a number of mode change protocols [14, 13].

It is also important to remark that the class hierarchy of the original framework is not compatible with the use of a code generator tool, as proposed here. For example, Listing 1 shows how the periodic release mechanism `M` and the real-time task `T` have to be declared after the programmer defines and declares the final task state `P`. With this code structure, a code generation tool can only set the scheduling attributes of a task in its `Initialize` procedure, and therefore it has to provide a task-specific `Periodic_Task_State`with this procedure already implemented. When the programmer extends this new task state to implement the task behavior, its initialization code would collide with the one that should be provided by the code generator.

Listing 1. A simple example of a periodic task using the original framework

```
-- with and use clauses omitted
package Periodic_Test is
  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);

  P : aliased My_State;
  M : aliased Periodic_Release (P'Access);
  T : Simple_Real_Time_Task(P'Access, M'Access, 3);
end Periodic_Test ;
```

3. Framework requirements and system model

In the following subsections we enumerate the requirements we have considered to extend the original framework to support efficient execution on multiprocessor platforms and to incorporate the concept of operating modes. We first describe the design context we are assuming, and then list the particular requirements for multiprocessor and multimode support we have considered. This set of requirements also serves the purpose of describing the system model we are assuming.

3.1. Design process requirements

The design and implementation of complex multiprocessor real-time systems requires using schedulability analysis techniques to ensure the system will meet its timing requirements at execution time. From the results of this analysis, the

[2]The term *job* refers to a particular activation of a task. Note that this concept is not directly supported by the Ada language.

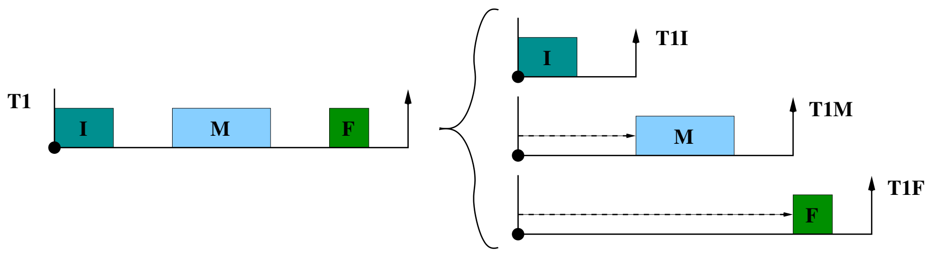

Figure 2. Decomposition of a task with multiple job steps.

scheduling attributes are derived for each task in the system, across the different operating modes. The same applies to ceiling priorities of resources shared by means of protected operations under the ceiling locking protocol. These scheduling attributes may include multiple task priorities, relative deadlines, release offsets and task processor migrations at specified times. Additionally, the programmer may want to handle some special events, like deadline misses, mode changes and execution time overruns. Translating *manually* this set of attributes into the application code is error-prone, since it implies dealing with functional and non-functional properties at the same time in the code space. Hence this work proposes to use a specific development tool that will generate the scheduling task behavior and initialization code on top of the new real-time framework, leaving the purely functional behavior to the system programmer. However, the design and implementation of this tool is still work in progress and cannot be presented until the framework proposal reaches a sufficient degree of consolidation and agreement.

3.2. Multiprocessor requirements

A real-time system is composed of a set of *tasks* that concurrently collaborate to achieve a common goal. Each real-time task can be viewed as an infinite sequence of *job* executions. In many cases, a job performs its work in a single step without suspending itself during the execution. Therefore, a task is suspended only at the end of a job execution to wait for the next activation event. However, some systems organize the code as a sequence of *steps* that can be temporally spaced to achieve a given system goal. An example of such systems is the IMF model [3], oriented to control systems, where each job is divided into three steps or parts: an Initial part for data sampling, a Mandatory part for algorithm computation and a Final part to deliver actuation information. Although these steps usually share the job activation mechanism, different release offsets and priorities can be used for each step in order to reduce the input/output jitter of the sampling and actuation steps.

These job steps constitute the *code units* where the programmer will implement the behavior of each task. However, as pointed out in [18, 17], complex real-time systems could be composed by tasks that need to detect deadline misses, execution time overruns, minimum inter-arrival violations, etc. The system behavior when these situations are detected is task-specific and it has to be implemented in different code units in the form of task control handlers. An example of this task-specific behavior is a real-time control task with optional parts. These optional steps would help to improve control performance in case there is sufficient CPU time available, but they have to be cancelled if a deadline is in danger, in order to send the control action in time.

For the purpose of timing analysis, the steps are usually regarded as subtasks [3, 10]. The subtasks share the same release mechanism, typically periodic, and separate each job execution by a given release offset. Figure 2 shows the decomposition of a control task following the IMF model. The rest of the scheduling attributes of these notional subtasks (e.g. priority, or maximum CPU time allowed) are established according to a given goal, such as improving the overall control performance by reducing input/output jitter.

When and where a given code unit is executed is determined by the scheduler of the underlying operating system. The scheduler will use a set of *scheduling attributes* to determine which job is executed and, in multiprocessor platforms, on which CPU. Examples of scheduling attributes are: release offset of the steps relative to the job activation, job priority, relative deadline, CPU affinity, worst-case execution time of the job, etc.

In a complex multiprocessor system, each job step can have a different set of scheduling attributes that could change during its execution depending on the selected scheduling approach. Based on the ability of a task to migrate from one processor to another, the scheduling approach can be:

Global scheduling: All tasks can be executed on any processor and after a preemption the current job can be resumed in a different processor. If the scheduling decisions are performed online, in a multiprocessor platform with M CPUs, the

M active jobs with the highest priorities are the ones selected for execution. To ensure that online decisions will not jeopardize the real-time constraints of the system, different off-line schedulability tests can be applied [4, 5]. When the scheduling decisions are computed off-line, release times, preemption instants and processor affinities can be stored in a static scheduling plan.

Job partitioning: Each job activation of a given task can be executed on a different processor, but a given job cannot migrate during its execution. The processor where each job is executed can be decided by an online global dispatcher upon the job activation, or it can be determined off-line by a scheduling analysis tool and stored in a processor plan for each task. The job execution order on each processor is determined online by its own scheduler using the scheduling attributes of each job.

Task partitioning: All job activations of a given task are executed in the same processor. No job migration is allowed. The processor where a task is executed is part of the task's scheduling attributes. As in the previous approach, the order in which each job is executed on each processor is determined online by the scheduler of that processor.

Task splitting is a different technique that combines task partitioning with controlled task migration at specified times. Under this approach, some authors suggest to perform the processor migration of the split task at a given time after each job release [12] or when the job has performed a certain amount of execution [11]. It is worth noting that this approach normally requires the information about the processor migration instant to be somehow coded into the task behavior.

We note that, from the different techniques and approaches enumerated so far in this section, global scheduling imposes implementation requirements that need to be either fulfilled by the underlaying real-time operating system, or implemented at user level by means of some kind of application-defined scheduler [1]. But the rest of approaches (task and job partitioning, and task splitting) can be implemented by using features that are being considered for inclusion in Ada 2012. Some examples have recently being proposed in [7, 2, 16]. In particular, tasks will impose the following requirements to the framework for multiprocessor platforms:

- The ability to establish the tasks' scheduling attributes, including the CPU where each job will be executed. These scheduling attributes can be set at the task initialization phase to support task partitioning, but they can also be dynamically changed at the beginning of each job activation to provide support for job partitioning, or after a given amount of system or CPU time has elapsed, to provide support for task splitting techniques.
- The flexibility to program, and to be notified about, the occurrence of a wide set of runtime events. These events include: deadline miss, execution time overrun, mode change, timed events driven by the system clock or CPU clock to manage programed task migrations, etc. Some of these events may additionally require to terminate the current job.
- The possibility to specify time offsets in order to support the decomposition of tasks with multiple steps into several subtasks.

3.3. Mode change requirements

Multimoded real-time systems differ from single-mode systems in that the set of tasks to schedule changes with time. There exists one set of running tasks per mode of operation, each with the proper scheduling parameters (tasks periods, deadlines, priorities, etc.) Operating modes are decided at design time to accommodate the different situations to be faced during the system's mission.

A mode change request is a request for tasks to either:

- change their scheduling attributes (priority, deadline, period, affinity, etc.) to those of the new mode, or
- become active, if they were not active in the old mode, or
- become inactive, if they are not needed in the new mode, or
- preserve their scheduling attributes and activation pattern, if they remain unchanged from the old to the new mode.

When tasks share protected objects, it should also be possible to change the ceilings of those protected objects to accommodate the requirements of the new mode. Since Ada 2005, it is possible to dynamically change ceiling priorities of protected objects. Protocols for choosing the right time to change ceilings are proposed in [15]. Note that ceilings must be changed from a priority that is not above the current ceiling, to avoid violation of the ceiling locking protocol. Hence we cannot delegate the change of ceilings to a task with an arbitrarily high priority.

4. Framework proposal

Considering all the requirements described in Section 3, we propose now a redesign of the original framework. The following subsections describe the components of this new proposal.

4.1. Real-Time Task State, Real-Time Task Scheduling and Real-Time Task Attributes

The state variables of all tasks will be spread in three task state objects, containing different types of information.

Task_State Derives from the interface Task_State_Interface. It contains the task code in the form of four abstract or null procedures:

- Initialize: abstract procedure that must contain the user initialization code for the task;
- Code: abstract procedure that contains the task's functional code, to be executed at each new release of the task;
- Deadline_Miss: a null procedure to be redefined if the task needs to implement a deadline miss handler;
- Overrun: similarly to Deadline_Miss, this is the place to implement the execution-time overrun handler.

Task_Sched This type offers the following operations:

- Initialize. This is the abstract procedure where the user is required to set up the task's scheduling attributes (priority, deadline, period, CPU), define the events to be handled for that task and connect those events with their corresponding event handlers (deadline miss and overrun).
- Set_Task_Attributes. This procedure is defined as null in the Task_Sched type. It may be overriden by a not null procedure in order to reset the task attributes to the task's original values for the current operating mode. It is useful after a task split, or as the last action of a job when implementing job partitioning.
- Adjust_Job_Attributes. This procedure (null by default) is intended to dynamically change the current job attributes, for example for job migration or for supporting a dual priority scheme.
- Mode_Change_Handler. This is the user's handler for mode changes. A mode change may be handled in part by user's code, at the user's old-mode priority. This will, for example, allow the proper task to safely change the ceiling priorities to the new-mode values. This is also the adequate place for the application code to perform device initialization for the new mode, if needed.

Task_Sched_Attributes This type offers the setter and getter subprograms for the individual task scheduling attributes, namely the execution time, relative deadline, priority, and offset. It also offers setters and getters for handling the *active* flag associated to each task. This flag informs the system about whether a task is currently running or waiting for activation.

4.2. Control Mechanisms

Adapting the original framework to support multiprocessor platforms and mode changes requires the framework to handle some *new* events. We propose to detach the event management from the release mechanisms, as proposed in the original framework. Hence the proposal of a new *control mechanism* abstraction.

A control mechanism is formed by a Control_Object and a Control_Event, that collaborate to implement the *Command* design pattern [8]. Control Objects will perform the *Invoker* role, that will ask to execute the *Command* implemented by the Control Event when some scheduling event occurs. The *Receiver* role is played by Task_State or Task_Sched types, while the *Client* role is performed by the initialization code that creates the event command and sets its receiver. Control objects allow triggering events at different times of the task's lifespan:

- *On job release*: This is useful, for example, to reset the deadline of a task scheduled under a deadline-based policy, such as EDF.
- *After a given amount of system time*: The use of Timing_Event allows, for example, to implement task splitting based on system time. The command executed by the control event will invoke the Adjust_Job_Attributes procedure of the Task_Sched object. Another example is to trigger the execution of a deadline miss handler

- *After a given amount of CPU time*: Task splitting based on CPU time will use an execution time timer to trigger the appropriate event. Triggering a cost overrun handler is another example of an event controlled by a CPU timer.
- *On job completion*: Handling an event at the time of completion of a job allows to handle events whose handling must be deferred until the job completes its execution. This type of handler may change the task attributes before the next job activation occurs, e.g., the Mode_Change_Handler procedure could be used to change the priority and period of a task before reprogramming its next release event.

Each control object is complemented by a Control_Event. A control event only needs now to execute the designated event handler, provided either by the Task_State or the Task_Sched object, depending on the particular event to be handled (deadline miss, mode change, etc.). There are three classes of events: *immediate*, *abortive* and *deferred*, depending on when they will be handled. Immediate events are immediately dispatched when they are triggered. This is useful e.g. for implementing task splitting or dual priority. Abortive events may abort the execution of the task's code. Examples of use are deadline miss and cost overrun events. Finally, deferred events are handled only when the task is not running (in other words,when the task is waiting for the next release). This behaviour is adequate for handling mode changes because it lets the last old-mode job of the task to complete before adjusting the task parameters for the new mode.

4.3. Release Mechanisms

Release mechanisms are the abstractions used to enforce task activation at the proper times. We propose to use two kinds of release mechanisms:

- Release_Mechanism, as in the original framework but with some minor changes to support CPU affinities and release offsets. Specializations of this basic mechanism implement the periodic, sporadic and aperiodic release patterns.
- Release_Mechanism_With_Control_Object, almost identical to the former but invoking On_Release and On_Completion procedures of all registered control objects each time a job is released or completed. It also offers the notification operations Notify_Event, Pending_Event and Trigger_Event to add event management support. Listing 2 shows part of the specification and body of Periodic_Release_With_Control_Objects. The code for event handling procedures is shown separately in listing 3. As suggested in [16], the new Set_Next_CPU procedures of Timing_Event and Dispatching_Domains are used to avoid unnecessary context switches when a job finishes its execution in a different CPU than the next job release is going to use, e.g, due to the application of a job partitioning or task splitting scheme (see lines 38 and 62).

4.4. Task Templates

There are three types of tasks defined in the framework. One is the Simple_Real_Time_Task, that must be attached to a *simple* release mechanism (one with no control object associated) and whose body implements just an infinite loop with two sentences: a call to Wait_For_Next_Release and an invocation to the Code procedure defined in the task state. Not being attached to any control object, this type of task is insensitive to events other than its own releasing event.

The second type of task, shown in Listing 4, is the Real_Time_Task_With_Event_Termination. This type of task must be attached to a release mechanism with control object, and its body is more complex. During the time the task is waiting for its next release, it may be notified of a pending event (line 16) and handle it (line 17), going back to wait for the next release. In the absence of an event during the wait for the next release, enters an asynchronous transfer of control (ATC, a *select then abort* statement) where the task's code may be aborted in the event of a deadline miss or execution time overrun.

A third type of task is provided (Real_Time_Task_With_Deferred_Actions) that differs from tasks *with event termination* in that the task's code is never aborted. This is for tasks whose last job in a mode must not be aborted when a mode change request arrives.

Listing 2. Periodic Release Mechanism with Control Objects

```
1  -- spec
2    protected type Periodic_Release_With_Control_Objects
3        (A: Any_Periodic_Task_Attributes ; NoC: Natural) is new Release_Mechanism_With_Control_Object with
4      entry Wait_For_Next_Release;
5      entry Notify_Event(E: out Any_Event);
6      entry Pending_Event(E: out Any_Event);
```

```
      procedure Trigger_Event(E: in Any_Event);
      procedure Set_Control(I: in Natural; C: in Any_Control_Interface );
      procedure Cancel_Control(I: in Natural );
      pragma Priority(System.Priority'Last);
   private
      entry Dispatch_Event(Postponed_Events)(E: out Any_Event);
      procedure Release(TE: in out Timing_Event);
      -- internal variables omitted
   end Periodic_Release_With_Control_Objects ;

-- body

   protected body Periodic_Release_With_Control_Objects is

      entry Wait_For_Next_Release when New_Release or not Completed is
         Cancelled: Boolean;
      begin
         if First then                    -- Release mechanism initialization
            New_Release := False;

            -- Initialize control objects
            for I in 1 .. NoC loop
               if Control_Objects(I) /= null then
                  Control_Objects(I). Initialize (Tid);
               end if;
            end loop;
            if A.Task_Is_Active then
               First := False;
               Epoch_Support.Epoch.Get_Start_Time (Next);
               Tid := Periodic_Release_With_Control_Objects.Wait_For_Next_Release'Caller;
               Next_Release := Next + A.Get_Offset;
               Event.Set_Next_CPU (A.Get_CPU);
               Event.Set_Handler (Next_Release, Release'Access);
            end if;
            requeue Periodic_Release_With_Control_Objects.Wait_For_Next_Release;
         elsif New_Release then           -- Job release
            New_Release := False;
            Completed := False;
            -- On release control objects
            for I in 1 .. NoC loop
               if Control_Objects(I) /= null then
                  Control_Objects(I).On_Release(Next_Release);
               end if;
            end loop;
         else                                      -- Job completed or aborted
            Completed := True;
            -- On completion control objects
            for I in 1 .. NoC loop
               if Control_Objects(I) /= null then
                  Control_Objects(I).On_Completion;
               end if;
            end loop;
            if A.Task_Is_Active then
               Next := Next + A.Get_Period;
               Next_Release := Next + A.Get_Offset;
               Event.Set_Next_CPU (A.Get_CPU);
               Event.Set_Handler (Next_Release, Release'Access);
            else
               First := True;
               Event.Cancel_Handler (Cancelled);
            end if;
            requeue Periodic_Release_With_Control_Objects.Wait_For_Next_Release;
         end if;
      end Wait_For_Next_Release;
       ...
      -- private

      procedure Release (TE : in out Timing_Event) is
      begin
         New_Release := True;
      exception
         when others =>
            Put_Line ("Release␣died");
      end Release;

   end Periodic_Release_With_Control_Objects ;
```

Listing 3. Code of event handling subprograms

```
entry Notify_Event(E: out Any_Event) when Event_Occurred(Abortive) is
begin
   requeue Dispatch_Event(Abortive);
end Notify_Event;

entry Pending_Event(E: out Any_Event) when Event_Occurred(Deferred) or Event_Occurred(Abortive) is
begin
   if Event_Occurred(Abortive) then
      requeue Dispatch_Event(Abortive);
   else
      requeue Dispatch_Event(Deferred);
   end if;
end Pending_Event;

entry Dispatch_Event(for T in Postponed_Events)(E: out Any_Event) when True is
begin
   E := Task_Event(T);
   Event_Occurred(T) := False;
   Completed := False;   -- In case of abortion during WFNR
end Dispatch_Event;

procedure Trigger_Event(E: in Any_Event) is
   ED : Event_Dispatching := E.Get_Event_Dispatching;
begin
   case ED is
      when Immediate =>
         E.Dispatch;
      when Postponed_Events =>
         Task_Event(ED) := E;
         Event_Occurred(ED) := True;
   end case;
end Trigger_Event;
```

Listing 4. Task template for real-time tasks with event termination

```
--spec
  task type Real_Time_Task_With_Event_Termination ( State : Any_Task_State_Interface ;
                                                    Sched: Any_Task_Sched_Interface;
                                                    R: Any_Release_Mechanism_With_Control_Objects) is
  end Real_Time_Task_With_Event_Termination;

--body
  task body Real_Time_Task_With_Event_Termination is
     E : Any_Event;
     Released: Boolean;
  begin
     Sched. Initialize ( State );
     State . Initialize ;
     loop
        select
           R.Pending_Event(E);
           E.Dispatch;
           Released := False;
        then abort
           R.Wait_For_Next_Release;
           Released := True;
        end select;
        if Released then
           select
              R.Notify_Event(E);
              E.Dispatch;
           then abort
              State.Code;
           end select;
        end if;
     end loop;
  end Real_Time_Task_With_Event_Termination;
```

5. Propagation and handling of events

The following subsections explain the flow and handling of events in two scenarios: a deadline miss event, and a mode change event.

5.1 Event handling example 1: Deadline_Miss

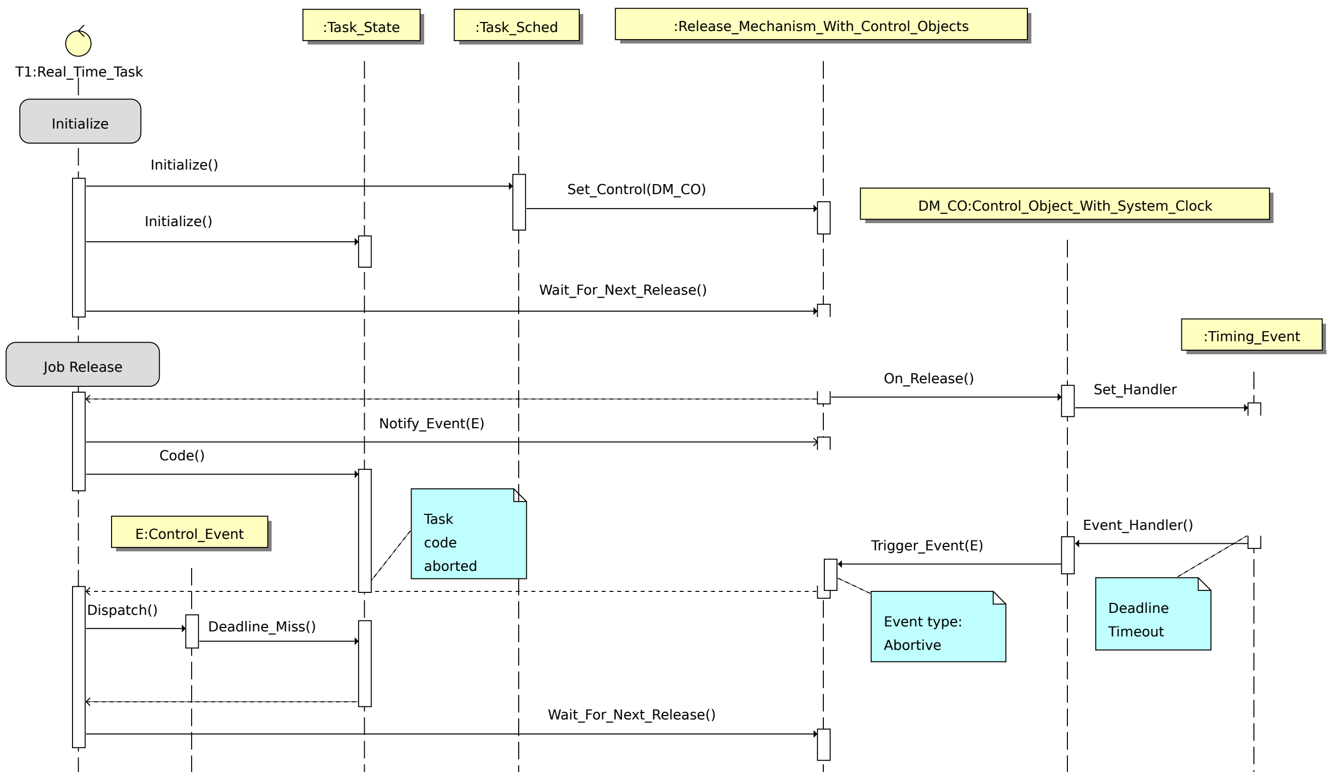

Figure 3. Control flow for a deadline miss scenario

Figure 3 shows the flow of events and actions under a deadline miss scenario. The framework entities involved in the process of handling the event (Task_State, Task_Sched, release mechanism, control object, timing event and control event) are shown in white-background rectangles. Vertical thin rectangles represent code executed in those entities. Everything starts on the top left corner of figure 3, when the Real_Time_Task starts and executes Initialize.

- During the task initialization, Task_Sched registers the control object DM_CO (of type Control_Object_With_System_Clock) with the task's release mechanism RM. The second call to Initialize refers to the initialization required by the task's logic, fully dependent on the application. In other words, this is user's code and not code related to any scheduling parameters of the task.
- Before the end of Wait_For_Next_Release, the On_Release procedure of the task's release mechanism RM is called. Within that procedure, the control object DM_CO programs a Timing_Event with the task's deadline before the task begins its execution.
- The task then enters the ATC with a call to Notify_Event, in the *select* part of the ATC, and an abortable call to State.Code. During the execution of State.Code, a deadline miss occurs (at the point marked as *Deadline timeout*, in figure 3) and the timing event executes the Event_Handler. The Event_Handler then triggers the event (via Trigger_Event) to the release mechanism, informing of the type of event (Deadline_Miss_Event).

- Since this is an *abortive* event, the barrier of Notify_Event becomes open, which aborts the call to State.Code and executes E.Disptach, which in turns ends up invoking the task's selected handler Task_State.Deadline_Miss.

The end of the loop makes the task go back to queue in Wait_For_Next_Release and the cycle starts over again.

5.2 Event handling example 2: Mode_Change

The event handling scenario in case of a mode change is shown in Figure 4. The flow of events and their handling occurs as follows:

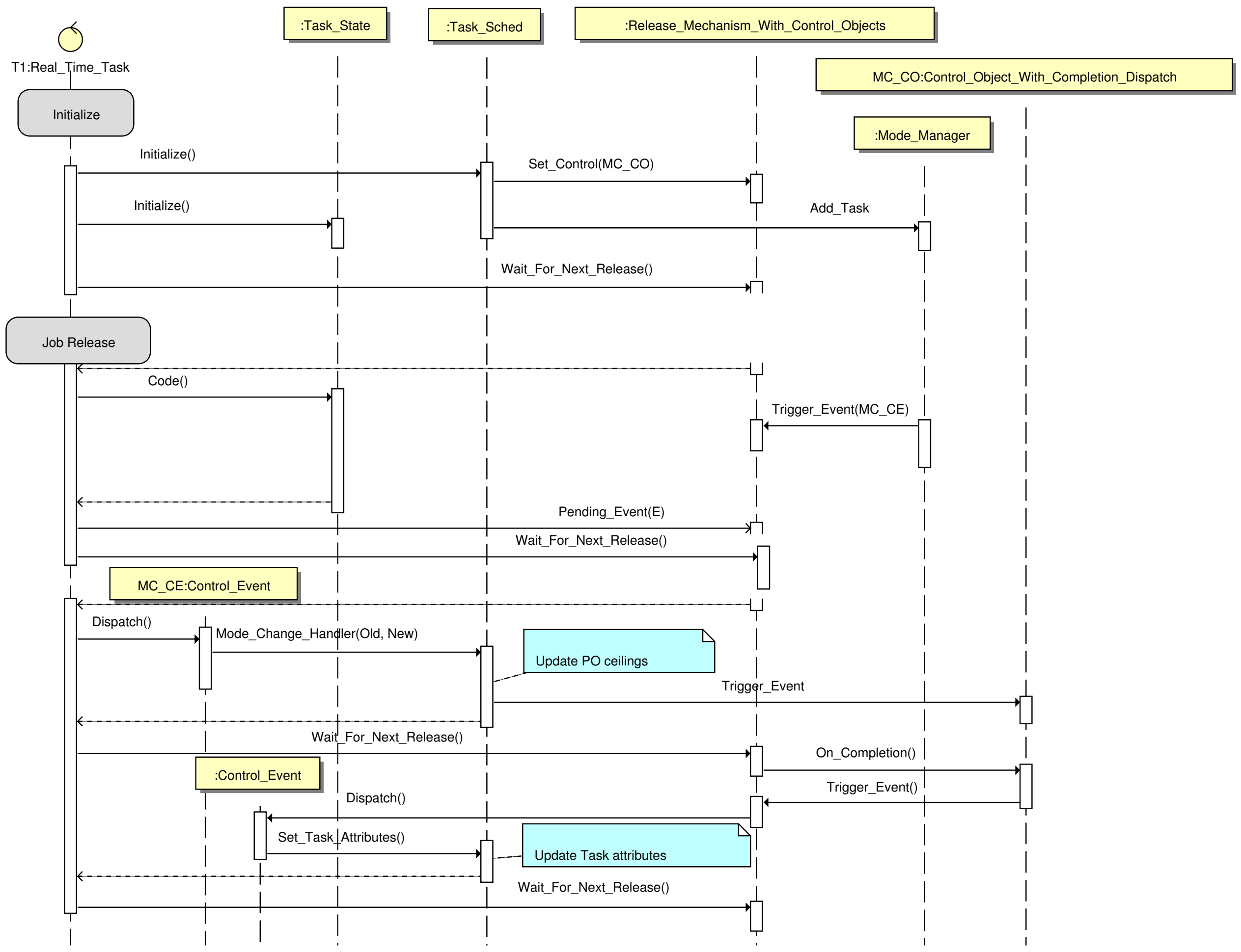

Figure 4. Control flow for a mode change scenario

- During the task initialization, Task_Sched registers the control object MC_CO (of type Control_Object_With_Completion_Dispatch) with the release mechanism RM. The task's release mechanism RM is also registered with the mode manager.
- The job is released and, during the execution of of Task_State.Code, a mode change request occurs. Consequently, the mode manager invokes Trigger_Event of RM, indicating that the event was a Mode_Change_Event.
- Since the mode change event is of the type *deferred*, the barrier of Pending_Event is the one that gets open.

- When State.Code ends, it goes back in the loop and invokes both Pending_Event and Wait_For_Next_Release (lines 9 and 13 in Listing 4, respectively). Since the call to Pending_Event is immediately accepted, Wait_For_Next_Release is aborted and E.Dispatch is executed, which in turn calls the mode change handler of the task (Task_Sched.Mode_Change_-Handler) at the task's current priority.
- The mode change handler configures the task's parameters according to the new mode, including the task's offset for its first activation in the new mode — in case the task is active in that new mode. This handler may also change the ceilings of its protected objects to the values of the new mode, in case it is the adequate task to do it [15].
- Then the handler calls Trigger_Event in MC_CO, so that the next call to Wait_For_Next_Release, the On_Completion procedure will be invoked and trigger the event Update_Task_Attributes. Since this is an event of type immediate, it will then invoke Task_Sched.Set_Task_Attributes, in charge of adapting the task attributes to the new mode, within the context of the protected action.
- In case the task is active in the new mode, the new attributes will be in effect upon completion of the next call of the task to Wait_For_Next_Release.

6. Conclusions

This paper has given account of the main design principles for a new (actually a redesigned) framework of real-time utilities. Our goals were (i) to make the framework amenable to automatic code generation tools; (ii) to adapt the framework to support execution on multiprocessor platforms; and (iii) to incorporate the mechanisms to deal with modes and mode changes.

The use of control objects and the associated control events, and the range of possibilities of event handling (immediate, abortive, deferred), have contributed to the scalability of the framework with respect to the number of types of events to handle. There's no need anymore to implement an exponential number of different task patterns depending on the number of events and release mechanisms. Furthermore, the new design allows to handle events at different times during the lifespan of a job (on release, on completion, or after a certain amount of system or CPU time), which is useful for implementing different multiprocessor techniques (such as task splitting, or other *ad hoc* techniques) and mode changes.

We note however that this is work in progress and the proposal is not fully complete[3]. We need to complete the design with a mode manager (to receive mode change requests and redirect mode change events to all tasks) and we definitely need more testing and discussion around the proposed design.

References

[1] M. Aldea, J. Miranda, and M. González-Harbour. Implementing an Application-Defined Scheduling Framework for Ada Tasking. In A. Llamosí and A. Strohmeier, editors, *9th International Conference on Reliable Software Technologies – Ada-Europe 2004*, volume 3063 of *Lecture Notes in Computer Science*, pages 283–296. Springer, 2004.

[2] B. Andersson and L. Pinho. Implementing Multicore Real-Time Scheduling Algorithms based on Task Splitting using Ada 2012. In J. Real and T. Vardanega, editors, *15th International Conference on Reliable Software Technologies – Ada-Europe 2010*, volume 6106 of *Lecture Notes in Computer Science*, pages 54–67. Springer, 2010.

[3] P. Balbastre, I. Ripoll, J. Vidal, and A. Crespo. A Task Model to Reduce Control Delays. *Real-Time Systems*, 27(3):215–236, September 2004.

[4] S. Baruah and T. P. Baker. Schedulability Analysis of Global EDF. *Real-Time Systems*, 38(3):223–235, April 2008.

[5] S. Baruah and N. Fisher. Global Fixed-Priority Scheduling of Arbitrary-Deadline Sporadic Task Systems. In *9th International Conference on Distributed Computing and Networking – ICDCN*, pages 215–226, 2008.

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

[3]The reader interested in the details of the code in their current state, may contact the authors to obtain the most updated version.

[7] A. Burns and A. Wellings. Dispatching Domains for Multiprocessor Platforms and their Representation in Ada. In J. Real and T. Vardanega, editors, *15th International Conference on Reliable Software Technologies – Ada-Europe 2010*, volume 6106 of *Lecture Notes in Computer Science*, pages 41–53. Springer, 2010.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Welsey, Reading, MA, 1995.

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

[10] S. Hong, X. Hu, and M. Lemmon. Reducing Delay Jitter of Real-Time Control Tasks through Adaptive Deadline Adjustments. In IEEE Computer Society, editor, *22nd Euromicro Conference on Real-Time Systems – ECRTS*, pages 229–238, 2010.

[11] S. Kato, N. Yamasaki, and Y. Ishikawa. Semi-partitioned Scheduling of Sporadic Task Systems on Multiprocessors. In IEEE Computer Society, editor, *21st Euromicro Conference on Real-Time Systems - ECRTS*, pages 249–258, 2009.

[12] K. Lakshmanan, R. Rajkumar, and J. Lehoczky. Partitioned Fixed-Priority Preemptive Scheduling for Multi-core Processors. In IEEE Computer Society, editor, *21st Euromicro Conference on Real-Time Systems - ECRTS*, pages 239–248, 2009.

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

[14] 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.

[15] J. Real, A. Crespo, A. Burns, and A. Wellings. Protected Ceiling Changes. *Ada Letters*, XXII(4):66–71, 2002.

[16] S. Sáez and A. Crespo. Preliminary Multiprocessor Support of Ada 2012 in GNU/Linux Systems. In J. Real and T. Vardanega, editors, *15th International Conference on Reliable Software Technologies – Ada-Europe 2010*, volume 6106 of *Lecture Notes in Computer Science*, pages 68–82. Springer, 2010.

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

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