A Real-Time Framework for Multiprocessor Platforms Using Ada 2012[⋆]

Sergio Sáez, Silvia Terrasa, and Alfons Crespo

Instituto de Automática e Informática Industrial,
Universidad Politécnica de Valencia,
Camino de vera, s/n, 46022 Valencia, Spain
{ssaez,sterrasa,alfons}@disca.upv.es

Abstract. The next release of the Ada language, Ada 2012, will probably incorporate explicit support for multiprocessor execution platforms. However, the implementation of multiprocessor scheduling approaches over the low-level abstractions offered by Ada forces the programmer to reconstruct complex task templates and algorithms in each new system. This work proposes to extend the previous Real-Time Utilities by Wellings and Burns to support multiprocessor platforms and to complete the framework with a code generation tool that translates the scheduling analysis reports into the real-time applications code.

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

1 Introduction

Real-Time and embedded systems are becoming more complex, and multiprocessor/multicore systems are becoming a common execution platform in these areas. Although schedulability analysis techniques for real-time applications executing over multiprocessor platforms are still not mature, some feasible scheduling approaches are emerging. However, to apply these techniques a flexible support at kernel and user-space level is needed. The forthcoming release of Ada 2012 is expected to offer explicit support for multiprocessor platforms through a comprehensive set of programming mechanisms [1].

However, the complexity of current real-time systems not only requires powerful execution platforms, but also support for different levels of criticality. This situation gives rise to heterogeneous system workloads that mix hard, soft and non real-time tasks. These tasks need to manage different kind of situations as deadline misses, termination of optional parts, control of CPU budgets, etc. Although Ada 2012 will provide powerful mechanisms to implement different multiprocessor scheduling approaches at application level [2,3], the offered abstractions to cope with these new requirements are still low level ones. Under this

[⋆] This work was partially supported by the Vicerectorado de Investigación of the Univ. Politécnica de Valencia under grant PAID-06-10-2397.

situation, Wellings and Burns argued in their work [4] that *there is a need for a standardized library of real-time utilities that address common real-time problems*. With the introduction of multiprocessor support and the related scheduling mechanisms, the need for a real-time standardized library avoiding the programmer to reconstruct the same algorithms and task templates on each system is exacerbated.

The main goal of this work is to make a step forward in this direction and to extend the Real-Time Utilities proposed in [4] to support the new requirements that arise in multiprocessor platforms. However, complex multiprocessor systems normally require complex analysis techniques. The results of this analysis, i.e. the *real-time analysis report*, will contain the scheduling attributes for each task in the system. These scheduling attributes could be composed by multiple task priorities, relative deadlines, release offsets and task processor migrations at specified times. Additionally, the programmer may want to handle some special events, e.g. deadline misses and execution time overruns. To translate this set of attributes into the real-time application code is an error-prone process. 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 Utilities, leaving the functional task behavior to the system programmer.

The rest of this paper is organized as follows. In section 2, the system load the new framework will support is presented. Section 3 briefly describes some multiprocessor scheduling approaches and their implementation feasibility at application-level. Section 4 outlines the previous proposal of a real-time support library and the new requirements imposed by the multiprocessor scheduling approaches. Then, section 5 presents the new components of the multiprocessor real-time framework. Finally, the code generation framework and the work conclusions are drawn in sections 6 and 7.

2 System Load Model

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. Depending on the activation mechanism used to release each job the tasks are classified into aperiodic and sporadic tasks, if their jobs are released by asynchronous events such as an external interrupt from a physical device, or periodic tasks, if their jobs are released by equally spaced internal clock events.

Typically, a job performs its work in a single step without suspending itself during the execution, and therefore, a task is suspended only at the end of a job execution to wait for the next activation event. However, some kind of jobs organize the code as a sequence of well-differentiated *steps* that can be temporally spaced to achieve a given system goal. An example of such tasks is a real-time control task following the IMF model [5], that are 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 to achieve input/output jitter reduction during sampling and actuation steps.

These job steps constitute the *code units* where the programmer of the real-time system will implement the behavior of each task. However, as pointed out in [4], complex real-time system 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 could be a real-time control task with *optional* parts. These optional steps or subtasks could help to improve control performance, but they have to be cancelled if a certain deadline is missed in order to send the control action in time.

When and where a given code unit is executed is determined by the scheduler of the underlying operating system. This scheduler will use a set of *scheduling attributes* to determine which job is executed and, specially in multiprocessor platforms, which CPU will use to execute it. Some of the scheduling attributes a real-time scheduler could use are: a *release offset* after the job activation, the job *priority*, its relative *deadline*, the *CPU affinity* information, different *execution times* 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 scheduling approach used to ensure the correct execution of the whole system. The next section presents some of these scheduling approaches that can be used to schedule real-time tasks in multiprocessor platforms.

3 Multiprocessor Scheduling Approaches

In order to achieve a predictable schedule of a set of real-time tasks in a multiprocessor platform several approaches can be applied. Based on the capability 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 fulfil the real-time constraints of the system tasks, different off-line schedulability tests can be performed [6,7]. If the scheduling decisions are computed off-line, releases times, preemption instants and processors where tasks have to be executed are 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 have to be 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.

In addition to these basic approaches, new techniques that mix task partitioning with task that migrate from one processor to another at specified times are already available in the literature. In this approach, known as *task splitting*, some works suggest to perform the processor migration of the split task at a given time after each job release [8] or when the job has performed a certain amount of execution [9]. It is worth noting that this approach normally requires the information about the processor migration instant to be somehow coded into the task behavior.

In the case that a task is composed by multiple job steps, specific scheduling analysis tools usually decompose these steps as different real-time subtasks [5,10]. These subtasks share the same release mechanism, typically a periodic one, and separate each job execution using a given release offset. Figure 1 depicts this decomposition of a control task. The rest of the scheduling attributes of these new tasks are established according to a given goal, e.g., to improve the overall control performance by means of input and output jitter reduction. Once the task steps are separated into different tasks, the multiprocessor scheduling approaches shown above can be combined with the control specific ones.

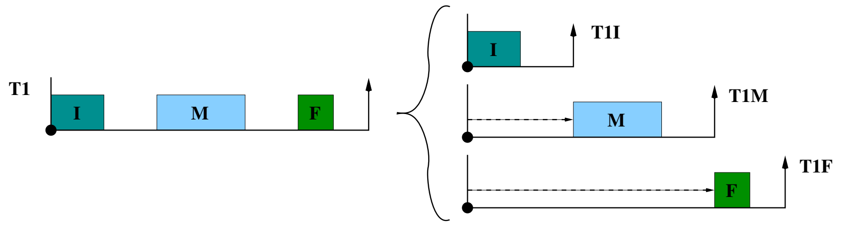

Fig. 1. Decomposition of a task with multiple job steps

Although schedulability analysis techniques that ensure the timeliness execution of the task set are not mature enough for all the scheduling approaches presented above, to offer a flexible support to easily apply the most established ones is part of the goals of this work. The next section analyzes the required functionalities a library of Real-Time Utilities has to provide to help in the applicability of these approaches and proposes a redesigned Real-Time Framework that helps to implement complex real-time systems over multiprocessor platforms.

4 New Design of Real-Time Utilities

This section analyses the current proposal of a library of Real-Time Utilities and revise the needed requirements to support the existing multiprocessor scheduling approaches presented above. This work proposes to redesign the framework presented by Wellings and Burns [4] to cope with these new requirements.

4.1 Previous Proposal of the Real-Time Utilities

The Real-Time Utilities proposed by Wellings and Burns offer a set of high-level abstractions that allow the Ada 2005 programmer to build real-time task with different release mechanisms (periodic, aperiodic and sporadic) and with different ways to manage deadline miss and execution time overrun events.

This framework organizes the support for building complex real-time systems around four kind of components:

Task State. This component encapsulates the main *code units* of a real-time task: initialization, the code to be executed on each release of a task job and specific handlers for deadline miss and execution time overrun events. Extending this component the programmer can easily build the tasks that compound the system. The task state also maintains the scheduling attributes for that task.

Release Mechanisms. This set of components provides different mechanisms to control the activation of each system task: periodic, sporadic and aperiodic tasks. However, release mechanism components also implement the support for deadline miss and execution time overrun detection and notification. As the detection and notification of these events are optional and orthogonal also with respect to the kind of release mechanism used, this gives rise to four different classes per release mechanism ($2^{\text{number of events}} \times$ number of release mechanisms). As the implementation of multiprocessor scheduling approaches explained in section 3 could require an increased number of events to be managed by the task code, this solution seems to be inadequate.

Real-Time Task. This kind of components implements the main structure of a real-time task, integrating the task state and its corresponding release mechanism. Different task templates provide support to immediately terminate a task on the occurrence of a given event. Once again, as the number of task templates depends on the number of termination events ($2^{\text{number of events}}$), this approach does not scale properly in relation to the number of such events[1].

Execution Servers. This component allows to manage group of tasks ensuring that a certain proportion of CPU time is not exceeded. As new Ada 2012 will tie group budgets to a single processor [11], this work is not going to consider this kind of components.

[1] Although not present in the original framework, a notifycation mechanism introduced in a later version allows to mitigate this drawback.

Although these components provide a useful set of high-level abstractions to implement real-time systems, they can be inadequate to implement multiprocessor scheduling techniques. As introduced above, scalability problems will arise if the number of events that could be managed by a given task is increased, as the number of types required to implement each release mechanism has an exponential relation with the number of events. Currently only deadline miss and execution time overrun events are managed, but the introduction of a new event will require to double the number of supporting types.

It is also important to remark that the current class hierarchy of the Real-Time Utilities is not compatible with the code generator framework proposed in this work. 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 only can set up 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 type to implement the task behavior its initialization code will collide with the one provided by the code generation tool. Although a possible solution to this problem could be to provide an additional Setup procedure to allocate the initialization generated code, this work proposes bellow a complete separation of scheduling and behavior task code.

Listing 1. A simple example of a periodic task using the Real-Time Utilities

```
-- 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;
```

4.2 Real-Time Multiprocessor Requirements

Real-Time multiprocessor scheduling techniques presented in section 3, such global scheduling, have some implementation requirements that have to be provided by the underlying Real-Time Operating System or be implemented at user level be means of some kind of Application Defined Scheduler [12], that also requires some support from the RTOS. However, the rest of the scheduling approaches, i.e. task partitioning, job partitioning and task splitting can be implemented using the low-level constructions that will be probably available in Ada 2012. This new support at language level has already been used in several works to show these multiprocessor scheduling techniques can be feasibly implemented [3,2,13]. This section presents the requirements the new framework has to fulfill to allow the implementation of the techniques with a new set of high-level abstractions.

Previous sections have presented a task model and a set of scheduling scenarios where a task could require:

– To establish its scheduling attributes, including the CPU where each job will be executed. These scheduling attributes can be set at the task initialization to support task partitioning, they can be dynamically changed at the beginning of each job to provide job partitioning support or after a given amount of system or CPU time has elapsed to provide support for some of the task splitting techniques.
– To program and to be notified about the occurrence of a wide set of runtime events. These events could include: deadline misses, execution time overruns, mode changes, timed events using system or CPU clocks to inform about a programed task migration, etc. Some of these events could also terminate the current task job.
– To specify task release delays or offsets in order to support the decomposition of tasks with multiple steps into several subtasks.

To support these requirements a new set of components are proposed in the following section.

5 New Framework Components

Taking into account the behavioral and scalability requirements that will be desirable in a library of Real-Time Utilities to support multiprocessor platforms and automatic code generation, the original components presented in section 4.1 have been split in: *Real-Time Task State*, *Real-Time Task Sched*, *Control Mechanisms*, *Release Mechanisms* and *Real-Time Tasks*. Figure 2 shows dependencies among new packages. The details are discussed in the next sections.

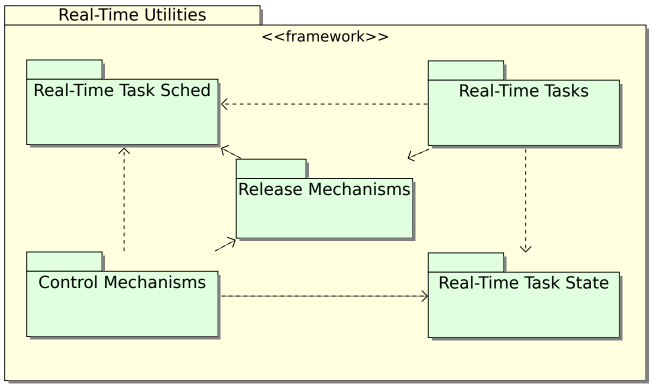

Fig. 2. Structure of new Real-Time Utilities packages

5.1 Real-Time Task Scheduling and Task State

The original Task_State tagged type provides the structure that allows the programmer to implement the code the real-time task will execute. Along with this code, the scheduling attributes of the task and the state variables associated with the real-time task are also allocated in this type or in the extended type the programmer defines to implement the task behavior.

As the scheduling attributes of a real-time task have been shown as something that different scheduling approaches can change dynamically during the task lifespan, it could be interesting to present this concept as an independent type. This will allow one to fully define the task attributes before the final task state is defined, to associate scheduling attributes to a task event, or to define arrays of attributes to implement static scheduling plans or job partitioning schemes.

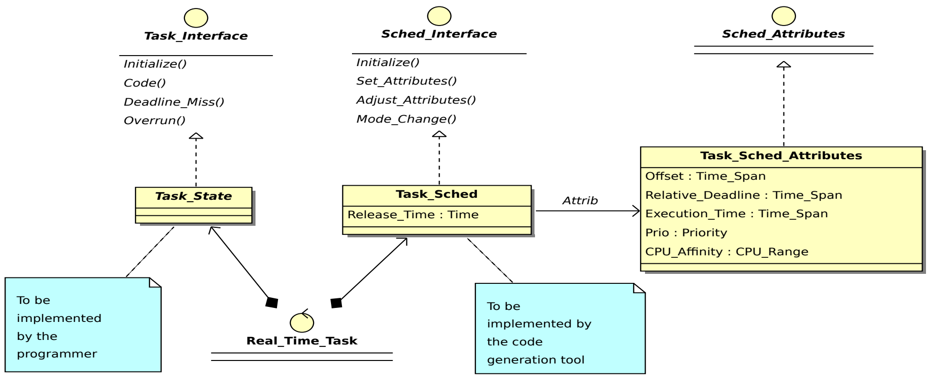

Fig. 3. Task related types

On the other hand, some multiprocessor scheduling approaches will require to implement part of the scheduler behavior in the task code, giving rise to task templates that depend on the results of the real-time analysis tool. Examples of these task templates in Ada for task splitting techniques can be found in [3] and a brief outline of a task using job partitioning in [13]. Since this work proposes to automatically generate this *scheduling task behavior* using a properly formatted real-time analysis report, it is suggested to encapsulate this code in a new Task_Sched tagged type. Although this conceptual separation of the scheduling behavior is not strictly necessary, it will prevent the programmer from accidentally overriding the scheduling code. Figure 3 shows the relations among these new types and Listing 2 sketches their basic operations.

The new tagged type Task_Sched will implement the operations to manage common scheduling situations. Example procedures shown in Listing 2 are intended to support different scheduling behaviors:

Listing 2. Task tagged types

```
package Real_Time_Task_Sched is
   ...
   type Task_Sched_Attributes is new Sched_Attributes with private;
   -- Get/Set operations omitted
   ...
   type Task_Sched is abstract new Sched_Interface with private;

   procedure Initialize (S : in out Task_Sched) is abstract;
   procedure Set_Attributes (S : in out Task_Sched) is null;
   procedure Adjust_Attributes (S : in out Task_Sched) is null;
   procedure Mode_Change(S: in out Task_Sched) is null;
   -- Get/Set operations omitted
private
   type Task_Sched_Attributes is new Sched_Attributes with
      record
         Offset            : Time_Span := Time_Span_Zero;
         Relative_Deadline : Time_Span := Time_Span_Last;
         Execution_Time    : Time_Span := Time_Span_Last;
         Prio              : Priority :=  Default_Priority;
         CPU_Affinity      : CPU_Range := Not_A_Specific_CPU;
      end record;
   ...
end Real_Time_Task_Sched;
```

– `Initialize` – this code will be used by the code generation tool to initialize the task attributes and the task scheduling mechanisms.
– `Set_Attributes` – this code will be used to establish the task attributes to its original values on each job release after a task split or to establish new values in the case of job partitioning scheme.
– `Adjust_Attributes` – this code will adjust task attributes during its execution, e.g., to perform a task split or a dual priority scheme [14].
– `Mode_Change` – this is the code that specifies how the task scheduling attributes are adapted to the new execution mode.

5.2 Real-Time Task Control Mechanisms

Multiprocessor scheduling techniques will require a higher number of events to be managed by the task code. This work proposes to detach the event management from the previous version of the release mechanisms, moving this support to the new package Control_Mechanisms. This new package contains two main component, Control_Objects and Control_Events, that will collaborate to implement the *Command* design pattern [15]. Control Objects will perform the *Invoker* role, that will ask to execute the *Command* implemented by the Control Event upon 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. Specialized versions of Control Objects will enable to trigger scheduling events under different situations:

Listing 3. Execution Time Overrun Control Event

```
...
   -- Cost Overrun Event
   type Cost_Overrun_Event (State: Any_Task_State;
                            Attrib:  Any_Task_Sched_Attributes;
                            Termination: Boolean) is new
      Control_Event_Using_CPU_Clock(Termination) with null record;
   procedure Dispatch(E: in out Cost_Overrun_Event);
   function Get_Event_Time(E: in Cost_Overrun_Event) return CPU_Time;
...
   -- Cost Overrun Event
   procedure Dispatch(E: in out Cost_Overrun_Event) is
   begin
      E.State.Overrun;
   end Dispatch;

   function Get_Event_Time(E: in Cost_Overrun_Event) return CPU_Time is
   begin
      return Clock + E.Attrib.Get_Execution_Time;
   end Get_Event_Time;
```

– *On job release*: it will allow the job partitioning scheme to be implemented by creating a new control event that will execute Set_Attributes procedure of the Task_Sched object before a new job is released.
– *After a given amount of system time*: The use of an Ada Timing_Event will allow a task splitting based on system time to be implemented. Command

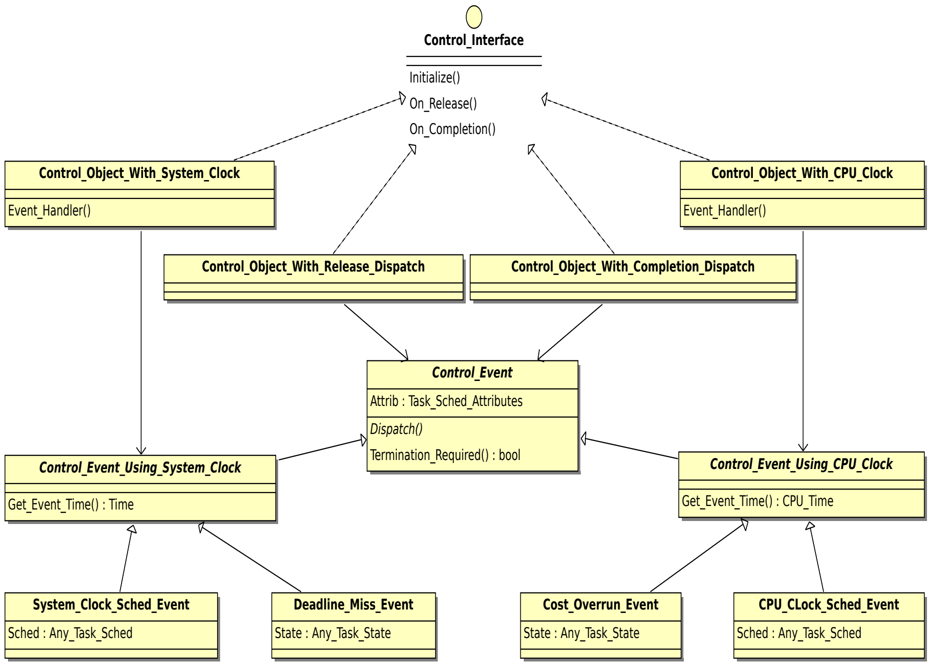

Fig. 4. Control Mechanisms

executed by the control event will invoke the Adjust_Attributes procedure of the Task_Sched object.

- *After a given amount of CPU time*: task splitting based on CPU time will use Ada Execution_Time.Timers to trigger the appropriated event. In this case, the command executed by the control event will also invokes the Adjust_Attributes procedure after the specified CPU time.
- *On job completion*: It will allow a task to execute a given procedure to respond to scheduling events that have deferred their actions until the current job completes its execution. This task procedure could change the task attributes before the next job activation occurs, e.g., the Mode_Change procedure could be used to change the priority and period of a task before reprogramming its next release event.

Listing 4. Periodic Release Mechanism

```
...
   protected type Periodic_Release (S: Any_Periodic_Task_Sched) is
         new Release_Mechanism with
      entry Wait_For_Next_Release;
      pragma Priority(System.Interrupt_Priority'Last);
   private
      procedure Release(TE : in out Timing_Event);
      TE : Timing_Event;
       ...
   end Periodic_Release;
...
   protected body Periodic_Release is
      entry Wait_For_Next_Release when New_Release or not Completed is
      begin
         if First then          -- Release mechanism initialization
            First := False;
            Epoch_Support.Epoch.Get_Start_Time (Next);
            Next := Next + S.Get_Period;
            S.Set_Release_Time (Next + S.Get_Offset);
            TE.Set_CPU (S.Attrib.Get_CPU);
            TE.Set_Handler (S.Get_Release_Time, Release'Access);
            New_Release := False;
            requeue Periodic_Release.Wait_For_Next_Release;
         elsif New_Release then    -- Job begin
            New_Release := False;
            Completed  := False;
            -- Invocation of On_Release procedures of Control Objects
         else                   -- Job end
            Completed := True;
            -- Invocation of On_Complete procedures of Control Objects
            -- TE.Set_CPU (S.Attrib.Get_CPU); --> Version with Control Objects
            Next := Next + S.Get_Period;
            S.Set_Release_Time (Next + S.Get_Offset);
            TE.Set_Handler (S.Get_Release_Time, Release'Access);
            requeue Periodic_Release.Wait_For_Next_Release;
         end if;
      end Wait_For_Next_Release;

      procedure Release (TE : in out Timing_Event) is
      begin
         New_Release := True;
         -- Set_CPU(S.Attrib.Get_CPU, T_Id); --> Version with Control Objects
      end Release;
   end Periodic_Release;
```

On the other hand, Control Events only have to implement the Dispatch operation that will execute the corresponding task behavior. Most of the control events are triggered by a time event and, in this case, they also have to provide the Get_Event_Time function to program that time event adequately. Listing 3 shows the implementation of *Execution Time Overrun* event.

Figure 4 depicts Control Objects and Control Events hierarchy, showing how *Deadline Miss* and *Execution Time Overrun* events along with events to support task splitting can be mapped into this new mechanism.

Control Object procedures On_Release and On_Completion, that will be invoked by the corresponding release mechanism, will allow the task to program the *Timing Event* or the *Execution Time Timer* on job release and to cancel them on job completion.

5.3 Real-Time Task Release Mechanisms

Once the control mechanisms have been introduced, the release mechanisms become simpler. Only two kind of protected types are needed per release mechanism. The first one, Release_Mechanism, remains similar to the previous version of the Real-Time Utilities with some minor changes to support CPU affinities and release offsets. Listing 4 shows the definition of Periodic_Release.

The second release mechanism, Release_Mechanism_With_Control_Objects, is almost identical to the former one but invoking On_Release and On_Completion procedures of all registered control objects each time a job is released or completed (marked as commentaries in Listing 4). It also offers the notification operations Notify_Event and Trigger_Event to add task termination support. As suggested in [13], the new Set_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 31 and 42).

Listing 5. Real-Time Task with Event Termination

```
   task type Real_Time_Task_With_Event_Termination (
         S : Any_Task_State; C : Any_Task_Sched;
         R : Any_Release_Mechanism_With_Control_Objects) is
   end Real_Time_Task_With_Event_Termination;
...
   task body Real_Time_Task_With_Event_Termination is
      E : Any_Control_Event;
   begin
      C. Initialize ; S. Initialize ;
      loop
         select
            R.Notify_Event(E);
            E.Dispatch;
         then abort
            R.Wait_For_Next_Release;
            S.Code;
         end select;
      end loop;
   end Real_Time_Task_With_Event_Termination;
```

5.4 Real-Time Tasks

Finally, although the *Simple Real-Time Task* remains almost identical, the template of a *Real-Time Task With Event Termination* becomes simpler due to the new event dispatching mechanism. As it can be observed in Listing 5, the task initializes the Task Sched object, containing the automatically generated initialization code, and Task State object, containing the programmer initialization code, before starting the main loop. It is also worth noting that the Wait_For_Next_Release procedure can also be aborted to support event notification while a task job is inactive.

6 Code Generator Tool

The proposed real-time multiprocessor framework have been redesigned to better adapt its components to multiprocessor platform requirements. However, the number of small components required to implement a complex task has increased and also the relations among them. This fact gives rise to a more elaborated initialization code to set up a task scheduling infrastructure. As translating scheduling attributes to application code and interconnect the resulting components can be an error-prone process, this work suggest the use of a very simple development that generates the *scheduling task behavior and initialization* from the real-time analysis report.

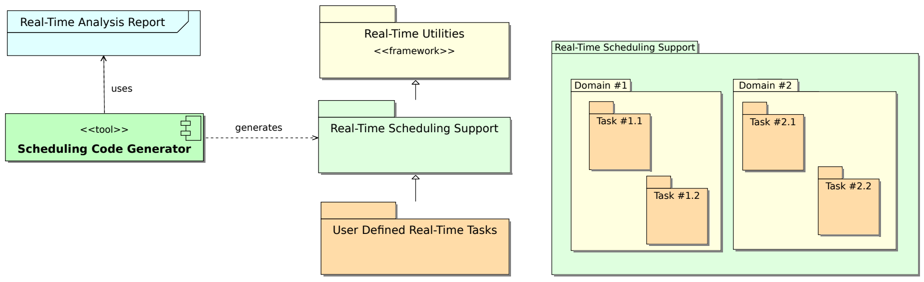
Real-Time Analysis Report
uses
<<tool>>
Scheduling Code Generator
generates
Real-Time Utilities
<<framework>>
Real-Time Scheduling Support
User Defined Real-Time Tasks
Real-Time Scheduling Support
Domain #1
Task #1.1
Task #1.2
Domain #2
Task #2.1
Task #2.2

Fig. 5. Real-Time Multiprocessor Framework

The current prototype of the code generation tool has been implemented using the PHP script language[2], and the YAML language [16] to format the file with the real-time analysis report. PHP languages is used due to its agile programming style and code generaton facilites that eases the prototyping process, while YAML language selection is based on its human readability. Specific XML2YAML conversion tools can be provided to support systems specified in standard XML formats.

[2] Using the PHP Command-Line Interface.

A scheme of the proposed real-time multiprocessor framework is shown on the left side of the Figure 5. On the right side, it is depicted the structure of the generated code: one package per Scheduling Domain and each Task of a given domain represented by a child package. The system programmer only has to extend the specialized Task_State type provided for each task in the system and implement the task behavior in the corresponding procedures.

7 Conclusions

The complexity of modern real-time system will require the use of multiprocessor platforms. Future Ada 2012 will provide basic mechanism to support part of the scheduling approaches proposed for that platforms. However, the Ada language support provides low-level mechanisms and the programmer have to reconstruct complex task templates and algorithms on each system.

This work has extended the previous Real-Time Utilities presented in [4] to support multiprocessor platforms and to better adapt its structure to an automatic code generation framework. The multiprocessor scheduling approaches have been analyzed, and the new requirements have been taken into account in the new framework components. The resulting framework allows the programmer to center the implementation efforts only in the task behavior, letting the scheduling mechanisms to be automatically generated from the real-time analysis report. Currently, the support for multi-stepped jobs is being finished and a web site to share the real-time framework code and the development tools is being set up.

References

1. Burns, A., Wellings, A.J.: Multiprocessor systems session summary. In: 14th International Real-Time Ada Workshop (IRTAW-14) (2009)
2. Burns, A., Wellings, A.J.: Dispatching domains for multiprocessor platforms and their representation in ada. In: Real, J., Vardanega, T. (eds.) Ada-Europe 2010. LNCS, vol. 6106, pp. 41–53. Springer, Heidelberg (2010)
3. Andersson, B., Pinho, L.M.: Implementing multicore real-time scheduling algorithms based on task splitting using ada 2012. In: Real, J., Vardanega, T. (eds.) Ada-Europe 2010. LNCS, vol. 6106, pp. 54–67. Springer, Heidelberg (2010)
4. Wellings, A.J., Burns, A.: Real-Time Utilities for Ada 2005. In: Abdennahder, N., Kordon, F. (eds.) Ada-Europe 2007. LNCS, vol. 4498, pp. 1–14. Springer, Heidelberg (2007)
5. Balbastre, P., Ripoll, I., Vidal, J., Crespo, A.: A task model to reduce control delays. Journal of Real-Time Systems 27(3), 215–236 (2004)
6. Baruah, S., Fisher, N.: Global fixed-priority scheduling of arbitrary-deadline sporadic task systems. In: Rao, S., Chatterjee, M., Jayanti, P., Murthy, C.S.R., Saha, S.K. (eds.) ICDCN 2008. LNCS, vol. 4904, pp. 215–226. Springer, Heidelberg (2008)
7. Baruah, S.K., Baker, T.P.: Schedulability analysis of global EDF. Real-Time Systems 38(3), 223–235 (2008)
8. Lakshmanan, K., Rajkumar, R., Lehoczky, J.P.: Partitioned fixed-priority preemptive scheduling for multi-core processors. In: 21st Euromicro Conference on Real-Time Systems, ECRTS 2009, pp. 239–248. IEEE Computer Society, Los Alamitos (2009)

9. Kato, S., Yamasaki, N., Ishikawa, Y.: Semi-partitioned scheduling of sporadic task systems on multiprocessors. In: 21st Euromicro Conference on Real-Time Systems, ECRTS 2009, pp. 249–258. IEEE Computer Society, Los Alamitos (2009)
10. Hong, S., Hu, X.S., Lemmon, M.: Reducing delay jitter of real-time control tasks through adaptive deadline adjustments. In: Euromicro Conference on Real-Time Systems, ECRTS 2010, pp. 229–238. IEEE Computer Society, Los Alamitos (2010)
11. Ada 2005 Issues. AI05-0169-1/06: Defining group budgets for multiprocessor platforms. (2010) Version: 1.7. Status: Amendment 2012
12. Aldea, M., Miranda, J., González Harbour, M.: Implementing an Application-Defined Scheduling framework for Ada tasking. In: Llamosí, A., Strohmeier, A. (eds.) Ada-Europe 2004. LNCS, vol. 3063, pp. 283–296. Springer, Heidelberg (2004)
13. Sáez, S., Crespo, A.: Preliminary multiprocessor support of Ada 2012 in GNU/Linux systems. In: Real, J., Vardanega, T. (eds.) Ada-Europe 2010. LNCS, vol. 6106, pp. 68–82. Springer, Heidelberg (2010)
14. Burns, A., Wellings, A.J.: Dual priority assignment: A practical method for increasing processor utilisation. In: 5th Euromicro Workshop on Real-Time Systems, pp. 48–55. IEEE Computer Society, Los Alamitos (1993)
15. Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
16. Ben-Kiki, O., Evans, C., Ingerson, B.: YAML ain't markup language (YAML) (tm) version 1.2. Technical report, YAML.org (September 2009)