Implementing Mixed Criticality Systems in Ada

Sanjoy Baruah and Alan Burns*

Department of Computer Science, The University of North Carolina, USA
*Department of Computer Science, University of York, UK

Abstract. Many safety-critical embedded systems are subject to certification requirements. However, only a subset of the functionality of the system may be safety-critical and hence subject to certification; the rest of the functionality is non safety-critical and does not need to be certified, or is certified to a lower level. The resulting mixed criticality system offers challenges both for static schedulability analysis and run-time monitoring. This paper considers both of these issues and indicates how mixed criticality applications can be implemented in Ada. In particular, code is produced to illustrate how the necessary run-time mode changes can be supported. This support makes use of a number of the new features introduced into Ada 2005.

1 Introduction

One of the ways that standard real-time systems has been extended in recent years is the removal of the assumption that all tasks in the system have the same level of criticality. Models have been produced that allow mixed criticality levels to co-exist on the same execution platform. For systems that contain components that have been given different criticality designations there are two, mainly distinct, issues: run-time robustness [7] and static verification [12, 3].

Run-time robustness is a form of fault tolerance that allows graceful degradation to occur in a manner that is mindful of criticality levels: informally speaking, in the event that all components cannot be serviced satisfactorily the goal is to ensure that lower-criticality components are denied their requested levels of service before higher-criticality components are.

Static verification of mixed-criticality systems is closely related to the problem of *certification* of safety-critical systems. The current trend towards integrating multiple functionalities on a common platform (for example in Integrated Modula Avionics, IMA, systems) means that even in highly safety-critical systems, typically only a relatively small fraction of the overall system is actually of critical functionality and needs to be certified. In order to certify a system as being correct, the certification authority (CA) must make certain assumptions about the worst-case behavior of the system during run-time. CA's tend to be very conservative, and hence it is often the case that the assumptions required by the CA are far more pessimistic than those the system designer would typically use during the system design process if certification was not required. However, while the CA is only concerned with the correctness of the safety-critical part of the system the system designer wishes to ensure that the entire system is correct, including the non-critical parts.

In this paper, we consider some of the scheduling issues involved in static verification. But we consider in more detail the run-time robustness requirements and show how they can be supported in Ada.

2 System Model

A system is defined as a finite set of components $\mathcal{K}$. Each component has a defined level of criticality, L. Each component contains a finite set of tasks. Each task, τ_i, is defined by period, deadline, computation time and criticality level: (T_i, D_i, C_i, L_i). These parameters are however not independent, in particular the following relations are assumed to hold (for the same task) between L and the other parameters in any valid mixed criticality system:

- The worst-case computation time, C_i, will be derived by a process dictated by the criticality level. The higher the criticality level, the more conservative the verification process and hence the greater will be the value of C_i.
- The deadline of the task may also be a function of the criticality level. The higher the criticality level, the greater the need for the task to complete well before any safety-critical timing constraint and hence the smaller the value of D_i.
- Finally, though less likely, the period of a task could depend on criticality. The higher the criticality level, the tighter the level of control that may be needed and hence the smaller the value of T_i.

These relations are formalised with the following axioms: if a component is reclassified so that task, τ_i is moved to criticality level L_i^1 from criticality level L_i^2 then

$$L_i^1 > L_i^2 \Rightarrow C_i^1 \geq C_i^2$$
$$L_i^1 > L_i^2 \Rightarrow D_i^1 \leq D_i^2$$
$$L_i^1 > L_i^2 \Rightarrow T_i^1 \leq T_i^2$$

At run-time a task will have fixed values of T, D and L. Its actual computation time is however unknown; it is *not* directly a function of L. The code of the task will execute on the available hardware, and apart from catching and/or dealing with overruns the task's actual criticality level will not influence the behaviour of the hardware. Rather the probability of failure (executing beyond deadline) will reduce for higher levels of L (due to C monotonically increasing with L).

In a mixed criticality system further information is needed in order to undertake schedulability analysis. Tasks can depend on other tasks with higher or lower levels of criticality. In general a task is now defined by: (T, D, $\boldsymbol{C}$, L), where $\boldsymbol{C}$ is a vector of values – one per criticality level, with the constraint:

$$L1 > L2 \Rightarrow C^{L1} \geq C^{L2}$$

for any two criticality levels $L1$ and $L2$.

The general task τ_i with criticality level L_i will have one value from its $\boldsymbol{C_i}$ vector that defines its *representative* computation time. This is the value corresponding to L_i, ie. $C_i^{L_i}$. This will be given the normal symbol C_i.

Definition 1 (Behaviors) During different runs, any given task system will, in general, exhibit different *behaviors*: different jobs may be released at different instants, and may have different actual execution times. Let us define the *criticality level of a behavior* to be the smallest criticality level such that no job executed for more than its C value at this criticality level.

As previously stated, two distinct issues have been addressed concerning the scheduling of mixed-criticality systems: *static verification*, and *run-time robustness*.

Static verification. From the perspective of static verification, the correctness criterion expected of an algorithm for scheduling mixed-criticality task systems is as follows: for each criticality level ℓ, *all jobs of all tasks with criticality* $\geq \ell$ *will complete by their deadlines in any criticality-*ℓ *behavior*.

Run-time robustness. Static verification is concerned with *certification* – it requires that all deadlines of all tasks τ_i with $L_i \geq \ell$ are guaranteed to be met, provided that *no* job executes for more than its level-ℓ worst-case execution time (WCET). Run-time robustness, in addition, seeks to deal satisfactorily with *transient overloads* due either to errors in the control system or to the environment behaving outside of the assumptions used in the analysis of the system. Even in behaviors that have a high criticality level by Definition 1 above, it may be the case that all jobs executing beyond their WCET's at some lower criticality level did so only for a short duration of time (i.e., a transient overload can be thought to have occurred from the perspective of the lower criticality level). A robust scheduling algorithm would, informally speaking, be able to "recover" from the overload once it was over, and go back to meeting the deadlines of the lower-criticality jobs as well. This is illustrated in the latter half of the paper.

3 Scheduling Analysis for Fixed Priority Scheduling

The distinctive feature of *mixed criticality* as opposed to *partitioned criticality* is that schedulability is obtained from optimising the temporal characteristics of the tasks rather than their important parameters.

Consider the common fixed priority deadline-monotonic scheduling scheme. Here the key operational parameter priority (P) is derived solely from the deadlines of the tasks. For any two tasks τ_i and τ_j: $D_i < D_j \Rightarrow P_i > P_j$. As noted earlier there will be a natural tendency for high criticality tasks to have shorter deadlines, but this is not a strong enough rule to result in a partitioning of deadlines (and hence priorities) via criticality levels.

In general therefore we must consider mixed criticality systems in which a task may suffer interference from another task with a higher priority but a lower criticality level. A phenomenon that could be referred to as *criticality inversion*.

To test for schedulability, the standard Response Time Analysis (RTA) [9, 1] approach first computes the worst-case completion time for each task (its response time, R) and then compares this value with the task's deadline D (ie. tests for $R_i \leq D_i$ for all tasks τ_i). The response time value is obtained from the following (where $\mathbf{hp}(i)$ denotes the set of tasks with priority higher that than of task τ_i):

$$R_i \;=\; C_i \;+\; \sum_{\tau_j \in \mathbf{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{1}$$

This is solved using a recurrence relation.

Three cases need to be considered depending on whether the arbitrary higher priority task τ_j has an equal, higher or lower criticality. For each case the correct value of C_j must be ascertained:

1. If $L_i = L_j$ then the tasks are at the same level of criticality and the normal representative value C_j is used.
2. If $L_i < L_j$ then it is not necessary to use the large value of computation time represented by C_j, rather the smaller amount corresponding to the criticality level of C_i should be used (as this is the level of assurance needed for this task). Hence eqn (1) should use $C_j^{L_i}$.
3. If $L_i > L_j$ then we have criticality inversion. One approach here would be to again use $C_j^{L_i}$, but this is allowing τ_j to execute for far longer than the task is assumed to do at its own criticality level. Moreover, it would require all low criticality tasks to be verified to the highest levels of importance, which would be prohibitively expensive (and in many ways undermine one of the reasons for having different criticality levels in the first place). Rather we should employ C_j, **but the run-time system must ensure that τ_j does indeed not execute for more than this value.**

The latter point is crucially important. Obviously all the shared run-time software must be verified to the highest critically level of the application tasks. One aspect of this is the functionality that polices the execution time of tasks and makes sure they do not ask for more resource that was catered for during the analysis phase of the system's verification. We note that Ada 2005 provides this functionality as illustrated in Section 5.

The response time equation (eqn 1) can be rewritten as:

$$R_i \;=\; C_i \;+\; \sum_{\tau_j \in \mathbf{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j^{min(L_i, L_j)} \tag{2}$$

Note that this use of minimum implies that values of C are only required for the task's criticality level and all lower criticality levels (ie. not for higher).

3.1 Shared Objects

With a more realistic system model, tasks will not be independent but will exchange data via shared objects protected by some mutual exclusion primitive or access control protocol (as in Ada's protected objects).

If an object is only used by tasks from the same component then it itself can be assigned the criticality level of the component. More generally if a shared object is used to exchange data between tasks from different components with different criticality level then the object must be produced and analysed at the ceiling criticality level of the components that use it.

As a consequence of the use of shared objects, a blocking term must be introduced into the scheduling equation:

$$R_i \;=\; C_i \;+\; B_i \;+\; \sum_{\tau_j \in \mathbf{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j^{min(L_i,L_j)} \tag{3}$$

where B_i is the blocking term; its value is the maximum execution time of any operation on a shared object that has a ceiling priority equal or higher than the priority of task τ_i, and which is also used by a task of lower priority.

3.2 Optimal Priority Ordering for Fixed Priority Scheduling

Using the analysis outlined above it is possible to allocate priorities to tasks in a way that optimises schedulability whilst being aware of criticality levels. Vestal [12] was the first to address this issue by assigning priorities to the tasks using Audsley algorithm [2]. That is, it first identifies some task which may be assigned the lowest priority; having done so, this task is removed from the task system and a priority assignment is recursively obtained for the remaining tasks. If the tasks are 'tested' in lowest criticality first order then, if the system is indeed schedulable, a priority ordering is found that delivers a schedulable task set and is as close as possible to being in criticality order (ie. lower criticality tasks are given lower priorities as often as possible). Vestal's claim of optimality for the algorithm was recently proved by Dorin et al [8].

Once a priority ordering is found then an implementation in Ada is straightforward. Tasks are assigned static priorities and a standard, even Ravenscar compliant, program can be developed. However, the run-time behaviour of a mixed-criticality system is not as straightforward. This issue is considered in the rest of this paper.

4 Managing Overruns and Increasing Robustness

As indicated in the introduction one of the important uses of criticality levels is to increase the robustness of the run-time behaviour of a system. This dynamic use of criticality levels can control overloads, either those that are derived from component failure or from excessive work load. The latter could involve a 'best effort' system that must deal with some level of work, but the environment cannot be constrained to never exceed this level. Whatever the level, the system should try and cope. Its main weapon for doing so is to drop less critical work and assign this capacity to the more critical tasks.

The golden rule here is that an overrun by a task at criticality M should not impact on tasks of higher criticality, but can steal resource from tasks with lower criticality.

The problem with this rule of course is that for schedulability the priority of a high criticality task may be below that of the M crit task.

To make run-time trade-offs between tasks of different criticality, it is important to define more precisely the measurement scale of the 'criticality' metric [10]. In the literature on mixed criticality systems there is no evidence to use anything other than an ordinal scale. Hence one task of criticality H is worth more than any number of tasks at criticality M (with $H > M$). It follows that a task, τ_i can execute for no longer than C_i (which is the value corresponding to its criticality level) unless it exploits slack in the system or steals execution time from strictly lower criticality tasks.

There could be a number of implementation schemes that could ensure this 'sharing' of computation time - ie. allow a task to use capacity statically allocated to lower criticality tasks. For example the use of execution time servers [5, 6, 4]. However all of these schemes have considerable run-time complexity and are not currently supportable on high integrity systems. Therefore here we exploit a simple scheme that only involves changes to task priorities. As a change to a task priority will impact on all aspects of the system's schedulability we identify this behaviour as a *mode change*.

The mode change to a more criticality aware priority ordering is triggered by the identification of an overrun of a task's execution time. For a long running system it is likely that the overload situation will eventually pass, and hence it is necessary to return to the original priority ordering. A mode change back to the original ordering must, however, be taken with care as it could cause a high criticality task to miss its deadline if undertaken prematurely. The simplest, and safest, protocol to undertake for this priority change is to wait until there is an idle tick and only switch priority at this time [11]. At this point all effects of the old mode are concluded and the new mode can proceed. The 'idle' tick can actually be of zero duration, but it must be a genuine point at which there is no existing load.

5 Implementation of the Run-time Protocol in Ada

To illustrate the run-time behaviour described above, and to demonstrate that the proposal is implementable, a simple example is programmed in Ada. The example uses just four tasks and two criticality levels (H and L). The details of the task set are given in Table 1.

Task	L_i	T_i	D_i	C_i	$C_i(overload)$
τ_1	H	100	25	12	
τ_2	L	100	50	10	50
τ_3	L	100	70	15	
τ_4	H	100	100	25	

Table 1. Example Task Set

The C values for each task relate to their criticality levels. On the assumption that the two lower critical tasks do not execute for more than their C values, the system is

schedulable with the static priorities in the order (highest first) τ_1, τ_2, τ_3 and then τ_4. In this example task τ_2 is allowed to enter an overloaded state in which its execution time is not 10 but rises to 50. This breaks the schedulability constraint, and hence a mode change is required in which the priority ordering changes to one that is more criticality based, ie τ_1, τ_4, τ_3 and then τ_2.

5.1 Implementation details

This section describes a prototype implementation of the priority changing protocol. The compiled code executed on a bare machine implementation. All priorities are changed from within a protected object (`Pri_Changer`). This object is called by a handler for a `Timer` object, and by the background task. It makes use of the dynamic priorities package. It therefore needs to record the task IDs of the system's tasks, and the specific priorities to be used in the two modes. These values are embedded in the code but in a real implementation would by taken from an external source. The priority changer is specified in the following package:

```
with System;
with Ada.Task_Identification; use Ada.Task_Identification;
with Ada.Execution_Time; use Ada.Execution_Time;
with Ada.Execution_Time.Timers; use Ada.Execution_Time.Timers;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
with Ada.Real_Time; use Ada.Real_Time;
with Ada.Dynamic_Priorities; use Ada.Dynamic_Priorities;
package Overload_Control is
   Max_Tasks : constant natural := 4;
   type Mode is (Normal, Overload);
   type Task_Ids is array(1..Max_Tasks) of Task_ID;

   protected Pri_Changer is
      pragma Priority(Min_Handler_Ceiling);
      procedure Change(M : Mode);
        -- called by background task
      procedure Changer(TM : in out Timer);
        -- called by a Timer
      procedure Register(N : Natural);
        -- called once by each of the four system tasks
   private
      Current_Mode : Mode := Normal;
      IDs : Task_Ids;
   end Pri_Changer;
end Overload_Control;
```

The body of this package is as follows:

```
package body Overload_Control is
   Task_Pris : array(1..Max_Tasks, Mode) of positive;
   protected body Pri_Changer is
      procedure Change(M : Mode) is
      begin
         if not (M = Current_Mode) then
            put("Mode␣Change␣to␣"); put("Normal"); new_line;
```

```
            Current_Mode := M;
            for I in 1..Max_Tasks loop
              Set_Priority(Task_Pris(I,Normal),IDs(I));
            end loop;
         end if;
      end Change;
      procedure Changer(TM : in out Timer) is
      begin
         if Current_Mode = Normal then
           put("Mode␣Change␣to␣"); put("Overload"); new_line;
           Current_Mode := Overload;
           for I in 1..Max_Tasks loop
              Set_Priority(Task_Pris(I,Overload),IDs(I));
           end loop;
         end if;
      end Changer;
      procedure Register(N : Natural) is
      begin
        IDs(N) := Current_Task;
      end Register;
   end Pri_Changer;

begin
  -- the following static values represent the required
  --  priorities in the two modes
  Task_Pris(1, Normal) := 10;  Task_Pris(1, Overload) := 10;
  Task_Pris(2, Normal) := 9;   Task_Pris(2, Overload) := 4;
  Task_Pris(3, Normal) := 8;   Task_Pris(3, Overload) := 8;
  Task_Pris(4, Normal) := 7;   Task_Pris(4, Overload) := 9;
end Overload_Control;
```

The main part of the program contains a task type for the four application tasks and a single background task. The task type uses discriminates to set its temporal characteristics:

```
task type Periodic(Id, Pri, Period, Deadline, WCET, LifeSpan : Natural);
```

`LifeSpan` is used to give a finite duration to the execution of the program.

The 'work' of each task is simulated by a busy loop that executes for the required time, using an execution time clock:

```
procedure Work(Initial : CPU_Time; C : natural) is
   X : integer := 0;
begin
   loop
     X := X + 1;
     exit when Clock - Initial > Milliseconds(C);
   end loop;
end Work;
```

The start of each of the tasks is coordinated by a 'starter' protected object:

```
protected Starter is
    pragma Priority(12);
    procedure Get_Start_Time(T : out Time);
private
```

```
    First : boolean := true;
    St : Time;
end Starter;

protected body Starter is
    procedure Get_Start_Time(T : out Time) is
    begin
      if First then
        St := Clock;
        First := false;
      end if;
      T := St;
    end Get_Start_Time;
end Starter;
```

The background task has a low priority and whenever its get to run it calls `Change` in protected object `Pri_Changer` and attempts to return the system to the normal mode. For most executions of the task this will be a 'null op' as the system will be in this mode. However execution time is not wasted as this task only runs when the system is otherwise idle. Note an entry cannot be used for this interaction; if the task called an entry (and was blocked) then as soon as the mode is changed to `Overload` and the barrier made `True` then the blocked task would execute and set the mode back to `Normal`!

If a more efficient version of the protocol is needed, so that other non-real-time tasks can execute at the lowest priority level in the normal mode, then this background task would need to be prevented from executing when the mode is `Normal`. A further action to take when the overload occurs would be the release (from suspension) of the 'background' task.

```
task Background is
   pragma Priority(1);
end Background;

task body Background is
    Epoc : constant Duration := 4.5;
    End_Time : Time := Clock + Seconds(30);
begin
    delay Epoc;
    -- needs to delay until tasks have registered
    loop
      Pri_Changer.Change(Normal);
      exit when Clock > End_Time;
    end loop;
end Background;
```

The code for each periodic task has the usual form. It gets a (common) start time, waits 5 seconds to allow all initialisation to occur. It then sets up its temporal parameters. For all but the second task (which is involved with the overload) the repeating code is essentially:

```
loop
   Work(CPU,WCET);
   if Clock > Next_Release + Relative_Deadline then
     put("Task␣"); put(Id); put("␣missed␣a␣deadline"); new_line;
   else
     put("Task␣"); put(Id); put("␣met␣a␣deadline"); new_line;
   end if;
   Next_Release := Next_Release + Release_Interval;
   exit when Next_Release > End_Time;
   CPU := Clock;
   delay until Next_Release;
end loop;
```

For the second task a `Timer` is set (and canceled if it does not fire). Also the duration of the work is extended for 3 iterations of the loop:

```
 loop
   Count := Count + 1;
   Set_Handler(Overrun, Milliseconds(WCET), Pri_Changer.Changer'Access);
   if Id=2 and (Count > 4 and Count < 8) then
     Put("Overload␣in␣Task␣"); put(Id); new_line;
     Work(CPU,WCET*5);
   else
     Work(CPU,WCET);
   end if;
   Cancel_Handler(Overrun, TempB);
   ... -- as other tasks
 end loop;
```

Note for completeness, all tasks have Timers set, but they are not called on to execute in the example executions (as execution time is less than WCET by construction).

The full code for the main program is continued in the Appendix.

5.2 Example execution

The task set runs for a number of iterations and meets all its deadlines. The overload then occurs in task τ_2 for three consecutive jobs. After that the task returns to its 'normal' behaviour. If no changes are made to the system then deadlines are missed in tasks τ_2, τ_3 and τ_4; τ_1, still has the highest priority and therefore is not impacted by the overload. A sample run of the code, on a bare board single processor platform, for this situation is as follows:

```
Main Started
Task           1 met a deadline
Task           2 met a deadline
Task           3 met a deadline
Task           4 met a deadline
Task           1 met a deadline
Task           2 met a deadline
Task           3 met a deadline
Task           4 met a deadline
Task           1 met a deadline
Task           2 met a deadline
Task           3 met a deadline
Task           4 met a deadline
Task           1 met a deadline
```

```
Task           2 met a deadline
Task           3 met a deadline
Task           4 met a deadline
Task           1 met a deadline
Overload in Task           2
Task           2 missed a deadline
Task           3 missed a deadline
Task           1 met a deadline
Overload in Task           2
Task           2 missed a deadline
Task           3 missed a deadline
Task           4 missed a deadline
Task           1 met a deadline
Overload in Task           2
Task           2 missed a deadline
Task           3 missed a deadline
Task           4 missed a deadline
Task           1 met a deadline
Task           2 met a deadline
Task           3 met a deadline
Task           4 missed a deadline
Task           4 met a deadline
Task           1 met a deadline
Task           2 met a deadline
Task           3 met a deadline
Task           4 met a deadline
Task           1 met a deadline
Task           2 met a deadline
Task           3 met a deadline
Task           4 met a deadline
Task           1 met a deadline
...
```

After the overload and nine missed deadlines the system does eventually return to a stable situation in which all deadlines are met.

To implement the mode change protocol, a `Timer` is defined for task τ_2 that is executed only when the task executes for more than its WCET (10). The handler calls a protected object that changes the priorities of all tasks to the 'overload' setting. A background, low priority, task is used to change the system back to the 'normal' priority settings. This task will only execute if there is slack in the system, in which case it is appropriate to reinstate the original priorities. An example run of the system when the protocol is engaged is as follows:

```
Main Started
Task           1 met a deadline
Task           2 met a deadline
Task           3 met a deadline
Task           4 met a deadline
Task           1 met a deadline
Task           2 met a deadline
Task           3 met a deadline
Task           4 met a deadline
Task           1 met a deadline
Task           2 met a deadline
Task           3 met a deadline
Task           4 met a deadline
Task           1 met a deadline
Task           2 met a deadline
Task           3 met a deadline
Task           4 met a deadline
Task           1 met a deadline
Overload in Task           2
Mode Change to Overload
```

```
Task           4 met a deadline
Task           3 met a deadline
Task           1 met a deadline
Task           4 met a deadline
Task           3 met a deadline
Task           2 missed a deadline
Overload in Task           2
Task           1 met a deadline
Task           4 met a deadline
Task           3 met a deadline
Task           2 missed a deadline
Overload in Task           2
Task           1 met a deadline
Task           4 met a deadline
Task           3 met a deadline
Task           2 missed a deadline
Task           2 missed a deadline
Mode Change to Normal
Task           1 met a deadline
Task           2 met a deadline
Task           3 met a deadline
Task           4 met a deadline
Task           1 met a deadline
Task           2 met a deadline
Task           3 met a deadline
Task           4 met a deadline
Task           1 met a deadline
```

As evident in this print out, all tasks apart from τ_2 meet their deadlines even when there is an overload. Of course τ_2 misses a series of four deadlines but once it has 'caught up' and the priorities changed by the background task then all tasks behave correctly.

6 Conclusion

In this paper we have considered some of the issues involved in supporting the production of mixed criticality systems. A type of system that is increasingly being considered in a wide range of applications. To verify such systems an extended form of scheduling analysis is needed and a somewhat more complex method of assigning priorities is required. Both of which are now available; and neither of which presents any significant problems for implementation languages such as Ada.

The required run-time characteristics are however beyond what is available in, say, a Ravenscar compliant real-time kernel. As tasks with low criticality may be executing with priorities higher than tasks with higher criticality it is imperative that tasks are not allowed to execute for more than their allotted execution time. Fortunately Ada 2005 allows task execution times to be monitored and code executed when bounds are violated. Such code can manipulate priorities so that critical tasks are protected. In this paper we have illustrated how such a protocol can be supported in Ada. An illustrative prototype multi-tasking program has been produced and executed on a bare-board single processor platform. Example executions of this code illustrate the required isolation between the 'failing' low criticality task and all higher criticality tasks. The code patterns provided by this prototype are such that their use in real industrial high-integrity applications should be seriously evaluated.

References

1. N. C. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.
2. N.C. Audsley. On Priority Assignment in Fixed Priority Scheduling. *Information Processing Letters*, 79(1):39–44, 2001.
3. S.K. Baruah and S. Vestal. Schedulability analysis of sporadic tasks with multiple criticality specifications. In *ECRTS*, pages 147–155, 2008.
4. G. Bernat, I. Broster, and A. Burns. Rewriting history to exploit gain time. In *Proceedings Real-time Systems Symposium*, pages 328–335, Lisbon, Portugal, 2004. Computer Society, IEEE.
5. G. Bernat and A. Burns. Multiple servers and capacity sharing for implementing flexible scheduling. *Real-Time Systems Journal*, 22:49–75, 2002.
6. M. Caccamo, G. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *Proceedings 21th IEEE Real-Time Systems Symposium*, 2000.
7. D. de Niz, K. Lakshmanan, and R. Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 291–300, 2009.
8. F. Dorin, P. Richard, M. Richard, , and J. Goossens. Schedulability and sensitivity analysis of multiple criticality tasks with fixed-priorities. *Real-Time Journal*, 2010.
9. M. Joseph and P. Pandya. Finding response times in a real-time system. *BCS Computer Journal*, 29(5):390–395, 1986.
10. D. Prasad, A. Burns, and M. Atkin. The measurement and usage of utility in adaptive real-time systems. *Journal of Real-Time Systems*, 25(2/3):277–296, 2003.
11. K. Tindell and A Alonso. A very simple protocol for mode changes in priority preemptive systems. Technical report, Universidad Politecnica de Madrid, 1996.
12. S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 239–243, 2007.

Appendix -Example Code

The full code of the example used in this paper is as follows (note the package `Overload_Control` is as given earlier).

```
pragma Task_Dispatching_Policy(FIFO_Within_Priorities);
pragma Locking_Policy(Ceiling_Locking);
with System;
with Ada.Task_Identification; use Ada.Task_Identification;
with Ada.Execution_Time; use Ada.Execution_Time;
with Ada.Execution_Time.Timers; use Ada.Execution_Time.Timers;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
with Ada.Real_Time; use Ada.Real_Time;
with Ada.Dynamic_Priorities; use Ada.Dynamic_Priorities;
with Overload_Control; use Overload_Control;
procedure Mixedcrit is

   task type Periodic(Id, Pri, Period, Deadline, WCET, LifeSpan : Natural) is
      pragma Priority(Pri);
   end Periodic;

   task Background is
      pragma Priority(1);
   end Background;

  protected Starter is
    pragma Priority(12);
    procedure Get_Start_Time(T : out Time);
  private
    First : boolean := true;
    St : Time;
   end Starter;

  procedure Work(Initial : CPU_Time; C : natural) is
    X : integer := 0;
  begin
   loop
     X := X + 1;
     exit when Clock - Initial > Milliseconds(C);
    end loop;
  end Work;

  task body Periodic is
    Epoc : constant Time_Span := Seconds(5);
    Next_Release, Start_Time, End_Time : Time;
    Release_Interval : constant Time_Span := Milliseconds(Period);
    Relative_Deadline : constant Time_Span := Milliseconds(Deadline);
    CPU, CPU_New : CPU_Time;
    T_ID : aliased Task_ID := Current_Task;
    Overrun : Timer(T_ID'Access);
    TempB : Boolean;
    Count : integer := 0;
  begin
    Starter.Get_Start_Time(Start_Time);
```

```
    Pri_Changer.Register(Id);
    Next_Release := Start_Time + Epoc;
    End_Time := Start_Time + Epoc + Seconds(LifeSpan);
    delay until Next_Release;
    CPU := Clock;
    loop
      Count := Count + 1;
      Set_Handler(Overrun, Milliseconds(WCET), Pri_Changer.Changer'Access);
      if Id=2 and (Count > 4 and Count < 8) then
        Put("Overload␣in␣Task␣"); put(Id); new_line;
        Work(CPU,WCET*5);
      else
        Work(CPU,WCET);
      end if;
      Cancel_Handler(Overrun, TempB);
      if Clock > Next_Release + Relative_Deadline then
         put("Task␣"); put(Id); put("␣missed␣a␣deadline"); new_line;
      else
         put("Task␣"); put(Id); put("␣met␣a␣deadline"); new_line;
      end if;
      Next_Release := Next_Release + Release_Interval;
      exit when Next_Release > End_Time;
      CPU := Clock;
      delay until Next_Release;
    end loop;
  end Periodic;

  task body Background is
    Epoc : constant Duration := 4.5;
    End_Time : Time := Clock + Seconds(30);
  begin
    delay Epoc;
    loop
      Pri_Changer.Change(Normal);
      exit when Clock > End_Time;
    end loop;
  end Background;

  protected body Starter is
    procedure Get_Start_Time(T : out Time) is
    begin
      if First then
        St := Clock;
        First := false;
      end if;
      T := St;
    end Get_Start_Time;
   end Starter;

  A : Periodic(1,10,100,25,12,1);  B : Periodic(2,9,100,50,10,1);
  C : Periodic(3,8,100,70,15,1);   D : Periodic(4,7,100,100,25,1);
begin
  put_line("Main␣Started");
end Mixedcrit;
```