Temporal Isolation with the Ravenscar Profile and Ada 2005

Enrico Mezzetti and Marco Panunzio and Tullio Vardanega
Department of Pure and Applied Mathematics, University of Padua
via Trieste 63, 35121 Padova, Italy
{emezzett, panunzio, tullio.vardanega}@math.unipd.it

Abstract

Modern methodologies for the development of high-integrity real-time systems build on abstract representations or models instead of code artifacts. Since analysis techniques are applied to models, it is important that system properties asserted during the analysis and the assumptions made for the analysis to hold are preserved across implementation and execution. In this paper we contend that the extent of properties preservation we require cannot be warranted using exclusively the language constructs allowed by the Ravenscar Profile. Hence, in the light of the new Ada 2005 features, we propose the formalization of a new augmented profile, fit for the purpose and yet still adhering to the pristine Ravenscar rationale.

1 Introduction

State-of-the art methodologies for the development of high-integrity real-time systems provide support for system and software design and analysis based on an architectural description of the system. Various forms of analysis are used at design time and on a representation of a system rather than on the target system itself. Such methodologies build on the assessment of a set of system attributes that are presumed to determine the system behaviour with respect to computation, time, space and communication. Whenever an analysis approach assumes a value (estimate) for a system attribute, that value shall be considered as a constraint and thus as a system property. In that light, it becomes paramount to cater for property preservation from system models down to implementation and run time.

Property preservation through the whole development process may get quite challenging when it comes to non-functional properties that pertain to dimensions such as concurrency, timing, space, etc., as they rely on the variable behaviour of concrete run-time entities. In this paper we focus on the preservation of timing properties, especially related to worst-case execution time (WCET), period, minimum inter-arrival time (MIAT) and deadline. In our vision, a methodology caters for property preservation by building on the following ingredients:

1. an analysis framework to statically analyse the system;
2. a programming model that enforces the assumptions of the analysis and permits to express exclusively and precisely the semantics supported by the analysis;
3. the means for run-time enforcement of properties.

The Ravenscar Profile (RP) [3] was one of the outputs of the 8th Internation Real-Time Ada Workshop (IRTAW), and it was subject to changes and clarifications during the 9th and 10th IRTAW. For years considered only a *de facto* standard, starting from the 2005 revision of the Ada language it has become a standard part of the language.

The RP well fits the needs for item 1 and 2 above, since it specifies a reduced tasking model for the Ada language [9] which excludes the language constructs that are exposed to unbounded execution time or non-determinism. Systems abiding by the RP are amenable to static analysis in the time dimension and they can be analysed (for example) with theories based on Response Time Analysis [10]. An additional benefit of the adoption of the RP is that Ravenscar kernels have a small footprint, incur small time overhead and, owing to

their reduced complexity, are better candidates for qualification in the target domain. A Ravenscar compilation system is expected to reject programs that include semantics forbidden by the profile.

Unfortunately however, the language constructs allowed by the RP do not suffice for the fulfilment of item 3. The recent revision of the Ada language (a.k.a. Ada 2005) however does offer some interesting language constructs that we consider of extreme interest for the achievement of item 3; namely: Execution Time Timers; Group Budget Timers; and Timing Events.

The definition of the RP (as in [9]) is equivalent to a set of pragma Restrictions, in particular:

```
pragma Restrictions( [...]
   No_Local_Timing_Events,
   No_Dependence =>
      Ada.Execution_Time.Group_Budgets,
   No_Dependence =>
      Ada.Execution_Time.Timers,
   [...]);
```

We can therefore notice that the RP disallows the use of execution time timers and group budget timers and allows the use of timing events only if declared at library level. In this paper we reason about the adoption of those additional language constructs in a Ravenscar system. We focus on the advantages that they deliver in the regard of run-time enforcement of timing properties and on the consequences of their adoption. Finally we try to understand if the combined adoption of the RP and those language constructs is sufficient to cover all kinds of timing faults that a high-integrity real-time system might incur.

The rest of the paper is organised as follows: in section 2 we outline the requirements that we want to fulfil; in section 3 we give a brief description of the language constructs of Ada 2005 we are interested in; in section 4 we outline two candidate architectures that may leverage such constructs; in section 5 we elaborate on the possible timing faults which may be incurred by a system and how those language constructs can be used to monitor and react to them; and finally in section 6 we draw some conclusions.

2 Support for temporal isolation

We commence by summarizing the activities required to warrant run-time preservation of timing properties and outline what part of it can already be achieved within the RP.

Preservation of timing properties at run time consists in three distinct activities:

1. enforcing controllable (constant) timing properties;
2. monitoring timing properties determined by system execution (hence variable by nature);
3. coping with violations of asserted properties.

In the first class of activities we include the enforcement of the period for cyclic tasks and the MIAT for sporadic tasks, as stipulated during the schedulability analysis of the system.

Activities of the second class involve the monitoring of the worst-case execution time (WCET) of tasks and task deadlines. The programming language should provide explicit mechanisms to detect a violation of those properties (WCET overrun and deadline miss).

In the third class we include the activities that are used to react to a detected violation of properties of the second class, that is to provide mechanisms to specify and execute a determined policy upon violation of those properties.

The adoption of appropriate task archetypes [2] and the use of the **delay until** construct make it possible to accurately enforce the period or the MIAT under the RP. Concerns related to the precision of the clock, and thus to the amount of wake-up jitter tasks are subject to, were reported for example in [18].

As regards the monitoring of timing properties, the RP only provides Timing Events (restricted to library level only) that can be used to detect deadline misses. The handler of the Timing Event could trigger some

logging or recovery actions. However the sole use of Timing Events is inconclusive to determine the root cause of the deadline miss and thus to isolate the problem. In fact, that kind of violation can be determined by several factors (most likely WCET overruns of either the task at hand or a higher-priority task). The reaction to this violation can thus be only coarse-grained. Finally, the RP does not provide any mechanism to monitor and react to WCET overruns.

We thus contend that the RP alone does not provide sufficient constructs for the monitoring of timing properties and the recovery from timing faults. In the following we briefly present a few language constructs of the Ada 2005 language and later we introduce their possible use in combination with the RP and the possible policies to react to various classes of timing faults.

3 Ada 2005 language constructs for monitoring real-time properties

The Ada 2005 standard [9] has recently introduced a set of new features to support the programming of high integrity real-time systems. We address here what we consider the three most important new features with respect to time monitoring, namely *execution time timers*, *group budgets* and *timing events*.

3.1 Execution time timers

Ada 2005 associates an execution time clock to each task to monitor its execution time, that is the time spent by the system executing such task and services on its behalf. Taking a step further, the execution time monitoring mechanism can be leveraged to trigger an event if the execution time exceeds a specified real-time value. This is achieved by means of Execution Time Timers. The `Ada.Execution_Time.Timers` package introduces Timer as an execution-time clock for a single task, which expires when the task consume a given amount of CPU time. A `Timer_Handler` protected procedure will be executed when the timer expires. An execution-time timer is thus relative to a specific task and is defined by an allotted time, either an absolute value or a real-time span, and a handler. Setting an execution-time timer allows, for example, to trigger a predefined action upon a WCET overrun. No countermeasure would be automatically taken on the overrunning task, only those determined by the handler procedure.

3.2 Group budget timers

The `Ada.Execution_Time.Group_Budgets` package extends the idea of execution-time timers to monitoring groups of tasks, instead of a single one. A Group Budget is essentially an abstraction of an execution-time budget: several tasks can be associated to a single time budget so that the Group Budget is accounted for their collective execution time. Similarly to execution-time timers, a `Group_Budget_Handler` protected procedure handler is defined to be executed at budget exhaustion. However, no further action is automatically taken on the associated tasks (i.e., they may keep on executing). A Group Budget is thus defined by an allotted time (a real-time span) and a handler.

3.3 Timing events

Whereas Execution-Time Timers and Group Budgets are concerned with monitoring the CPU time of tasks, Timing Events are a programming abstraction that allows the program to explicitly define code to be executed at a specific time instant, without the need to employ a dedicated task. Timing events, as defined by the `Ada.Real_Time.Timing_Events` package, are managed like interrupts but are triggered by the progression of the system clock. The code to be executed is provided through a `Timing_Event_Handler` protected procedure which is executed at the specified time. A timing event is thus defined by an allotted time, either an absolute value or a real-time span, and a handler.

3.4 Summary

Table 1 below provides a summary of Ada language support to timing properties enforcement and monitoring, with a view to Ravenscar-compliance.

Language constructs	Timing properties			within RP
	Period or MIAT	Deadline miss	WCET overrun	
delay until	∗	○	○	yes
Timer	○	○	•	no
Group_Budgets	○	○	•	no
Timing_Events	○	•	○	yes(*)

Symbols: ∗ = property enforcement; • = property monitoring;
○ = unable to enforce/monitor such property.
(*) only declared at library level

Table 1: Language support for timing property preservation.

4 Time-monitoring architectures

The coexistence of different criticality levels in one and the same system requires a mechanism of isolation between the most critical applications and the less critical ones; a failure in a lower-criticality application might unacceptably affect a higher-criticality application. Therefore, isolation should be provided (at least) along three orthogonal dimensional axes: time, space and communication.

Several approaches [5, 13] in the literature exploit the advanced Ada 2005 timing features to provide time-monitoring of the application and, in turn, to attain temporal isolation between different criticality levels. We describe here two architectural paradigms that aim at assuring temporal isolation when multiple applications share the same processor, based on *execution-time servers* [6] and on the recent but promising concept of *priority bands* [13]. We are mainly interested in evaluating the time-monitoring mechanisms, rather than hierarchical scheduling issues.

4.1 Server-based Architecture

Execution-time servers [6, 13] are special entities that act as local schedulers in hierarchical scheduling approaches. They are responsible for allocating a fixed amount of processor time to their associated application so that when the respective tasks execute, their execution time is subtracted from the server budget. At budget exhaustion the application tasks will be prevented from executing further until the next budget replenishment. Servers can be classified on the basis of their replenishment and budget consumption policies. Typically, the budget replenishment is scheduled periodically (with periodic servers), but it can be triggered by specific events as well (with aperiodic servers). With respect to budget consumption, when there is no ready task associated, the server may either preserve (with bandwidth-preserving servers) or notionally consume (with polling servers) its residual budget. Common server implementations are the Periodic [14], Deferrable [16] and Sporadic [15] servers.

As reported in [5], Ada 2005 does not explicitly support execution-time servers but it does provide the primitive constructs from which servers can be implemented. Such primitive constructs are those introduced in Section 3: `Group_Budgets` and `Timing_Events`. The former construct permits to define an execution-time budget, a group of associated tasks and a handler to be triggered on budget exhaustion; the latter, for example, permits to define a time-triggered event to replenish the budget.

However, the execution-time server approach is usually related to the hierarchical scheduling paradigm in a flavour such that budget exhaustion is considered as a natural event, inherently related to system scheduling, as it would be with soft real-time applications. Conversely, from our standpoint, servers are mainly intended to provide time-monitoring of (a group of) tasks. Accordingly, budget exhaustion must be considered as the result of a critical timing misbehaviour which should never occur. Such situation must therefore be not only readily intercepted, but also handled appropriately.

It is worth noticing that the choice of a proper replenishment time and period are fundamental to warranting a prompt detection (and reaction) to WCET overruns. Furthermore, no budget replenishment policy can be assumed to be completely safe and only the addition of `Execution_Time.Timers` would warrant reliable

monitoring [11].

The RP does not include group budgets. However, as stated in [17], such mechanisms could be safely and efficiently used in high-integrity real-time systems, provided that they are only declared at library level. In case of budget exhaustion, we should also apply some sort of corrective actions and possibly preventing further misbehaviours, i.e., all registered tasks should be prevented from further executing and some kind of corrective operation should be performed. Apart from the corrective actions, which are arguably application dependent, trying to prevent tasks from further executing is a hard issue under RP constraints. In fact, we can neither block the registered tasks in a hold state, as provided by the `Ada.Asynchronous_Task_Control` package, nor degrade them to a background priority, as provided by the `Ada.Dynamic_Priorities` package. We will further discuss this issue in Section 5.

4.2 Priority-Band Architecture

An alternative architectural approach to temporal isolation, named *Priority-band architecture* [13], builds on the new Ada 2005 scheduling features. Among several novel features, the new Ada standard introduced the `Priority_Specific_Dispatching` configuration pragma. Using this pragma, the range of priorities allowed by the run-time system can be split in contiguous and disjoint subsets termed priority bands. Each distinct priority band can be considered as a local scheduler that has its local scheduling policy, chosen among those defined in the language standard: FIFO within priorities, non-preemptive FIFO, Round Robin and Earliest Deadline First. Hence, from the hierarchical scheduling standpoint, the global scheduling policy is Fixed Priority Preemptive Scheduling (FPPS), whereas local schedulers are realized with the selection of a specific dispatching policy for each priority band.

As suggested in [13], a logical partition can be implemented as a set of tasks which are assigned priorities within the same priority band. Therefore the definition of a set of priority bands and a proper assignment of task priorities permits to split a system into a set of logical partitions. Hence each task is assigned to a logical partition on the basis of the priority band its base priority belongs to.

However, that scheduling mechanism does not suffice to warrant temporal isolation between partitions. In fact, scheduling assumptions may be violated by run-time misbehaviours and one or more tasks may miss their deadlines, for example owing to WCET overruns. We can cope with that misbehaviour by exploiting `Execution_Time.Timers`.

As discussed in Section 3, execution-time timers build on execution-time clocks which are associated to each task to monitor its execution time. Hence these clocks can be used to detect WCET overruns: when the execution time of a task exceeds the WCET specified value, an alarm is triggered and a specific handler is executed to take the desired countermeasures. In contrast with the server-based approach, the WCET violation is intercepted with a finer granularity level, thus allowing an immediate and reactive detection. In fact, using Group Budget Timers alone we are just aware that a WCET overrun has been detected but we are aware of neither who is the culprit nor when the overrun has actually occurred.

The considerations about WCET overruns made with respect to execution-time servers, obviously still hold true: they must be considered as the result of a critical misbehaviour which should never occur. In case of WCET overrun, we should both prevent the culprit task from executing any further and apply some sort of corrective action, which will be discussed in the next section.

The RP allows neither dispatching policies other than FIFO within priorities nor priority specific dispatching. A complete study on the static analysability of Priority Specific Dispatching has not been performed yet; however such study could leverage on existing work on EDF within priorities [8] extended to address also Fixed-priority non-preemptive and Round Robin local scheduling.

The restrictions of the RP do not allow execution-time timers, though they are instrumental to preserving temporal isolation among subsystems. Similarly to group budgets, they could be safely and efficiently used, provided that they are only declared at library level and there is at most one execution-time timer per task [17], so as to avoid an overly complex implementation of them.

4.3 Summary

Table 2 provides a summary evaluation of server-based and priority-band architectures in terms of the involved language constructs and the respective effectiveness.

Sample architectures and language constructs	Expected Behaviour	within RP
Server-based		
`delay until`	*Period/MIAT enforcement*	yes
`Execution_Time.Group_Budgets`	*Coarse-grained WCET monitoring*	no
`Execution_Time.Timing_Event`	*Budget replenishment*	yes(*)
Priority-band		
`delay until`	*Period/MIAT enforcement*	yes
`Priority_Specific_Dispatching`	*Local scheduling*	no
`Execution_Time.Timer`	*Fine-grained WCET monitoring*	no

(*) only declared at library level

Table 2: Evaluation of sample architectures.

5 Taxonomy of temporal faults

As briefly mentioned in section 2, we are mainly interested in the detection of deadline misses and WCET overruns and the possible policies to react to or recover from those timing faults, since task period or MIAT can be accurately enforced by adopting adequate code patterns and constructs.

5.1 Fault detection

Deadline monitoring can be performed via Timing Events. A timing event is set at each release of a task. In the case the task is able to complete its activation, the timing event is cleared (using the `Cancel_Handler` procedure). In the case the deadline is missed, the timing event handler is executed. Unfortunately, a detected deadline miss alone is not very useful to determine the cause of the problem, so we argue that no other useful operations can be performed in addition to logging the deadline miss for the purpose of information integration over time.

On the contrary, WCET monitoring is far more useful to understand and try to remedy timing faults. WCET overruns may be detected using Execution-Time Timers, as explained in previous sections. Each task is coupled with a timer that monitors the actual CPU time consumed by the task and executes a `Timer_Handler` procedure whenever the task exceeds its allotted CPU time. Upon each task activation the execution-time timer is set to the task WCET bound. In contrast with deadline misses, detected WCET overruns are always ascribed to the overrunning task.

A particular case of WCET overrun may occur when the execution of a critical section of a shared resource takes more than estimated. The longest execution time of each critical section has to be estimated in order to calculate for each task its *blocking time*, that is the longest time the task of interest can be prevented from executing due to resource contention by *lower* priority tasks. In fact, the worst-case response time of a task τ_i is determined by three combined contributions: the WCET of τ_i itself (C_i); the worst-case interference due to the execution of higher-priority tasks or system operation like kernel overheads (I_i); and the blocking time due to the use of a synchronization protocol to arbitrate shared resource contention (B_i).

$$R_i = C_i + B_i + I_i$$

The `Ceiling_Locking` policy provides a minimised bound to the blocking time and guarantees each task can be blocked at most once per activation. Furthermore, if blocking occurs, it is prior to task execution after release.

It is worth noting that if a task executes inside a critical section for longer than estimated, this does not imply it will violate its own WCET (because for example the remaining part of the execution uses less time); however, this situation can affect the schedulability of other tasks. Moreover, in contrast with task WCET violations, which may only affect the schedulability of the faulty task or of lower priority tasks, the overrun in a critical section may cause a missed deadline even for higher priority tasks whose priority is lower than or equal to the ceiling of the used shared resource.

One straightforward way to monitor the overall blocking time for a task, would consist in using a timer to measure the time a task is prevented from executing due to priority inversion, in a flavour similar to [7]. However owing to the lack of direct or indirect Ravenscar support for it, that approach is currently infeasible.

An alternative approach could be the monitoring of the execution time in each critical section; that is monitoring the task that is currently holding the shared resource instead of those waiting for it. Unfortunately none of the surveyed mechanisms seems suitable. Using an `Execution_Time.Timer` to monitor each critical section is not possible since they can be associated to only one task, upon timer creation and thus lacking a way to reassign them to each task entering the critical section. `Group_Budgets` could be in principle coupled with each shared resource; however, according to [9], D.15.2 :

13/2 [...] *A task can belong to at most one group.* [...]

thus making the approach not usable in the case of nested accesses to shared resources or if group budgets are already used in the system architecture. This kind of approach would also add considerable time and space overhead to the monitoring framework of the architecture.

5.2 Fault handling

Once a timing fault has been detected, proper countermeasures should be taken in order to recover the system from the erroneous state, or at least to limit the amount of possible harm. The way a system should react to a timing fault may depend on the nature of the timing fault itself. Since the simple detection of a deadline miss in the general case is not informative enough to allow the identification of its cause, we will focus on handling WCET overruns.

The nature of a WCET overrun can be of two kinds. In fact it can be:

- *transient*, if the task exhibits a slight and intermittent overrun with respect to the calculated or estimated value;
- *permanent*, if the task exhibits a permanent overrun, caused by a major design or analysis fault (i.e., every time the task is elected for execution, it executes longer than expected and, possibly, forever).

A *transient* overrun can occur for example if there has been an inaccurate estimation of the WCET. Since the WCET assumed at system start up are the values that are used during the schedulability analysis of the system, this kind of violation can potentially make the system infeasible. In fact, under fixed-priority preemptive scheduling policy, all tasks that have a priority equal to or lower than the overrunning task may consequently miss their deadlines.

The policies to react to this situation are application specific; they mostly depend on the criticality of the affected subsystems and whether an occasional deadline miss may be acceptable. The possible policies include:

- *Error logging*: the overrun is logged. Although this does not remedy the problem, the log can be used to inform (sub)system-wide fault handling policies.
- *Second chance*: as proposed in [11] and [12], it is possible to calculate for each task the amount of time it can execute exceeding its assumed WCET while still preserving the feasibility of the overall system. If the overrun is not greater than this additional *slack time*, the execution time budget for the task may be extended until the termination of the current job.
- *Period or MIAT change*: according to the activation pattern of the task, it can be useful to increase the period or the MIAT of the task itself. If a sustainable schedulability analysis theory [1] was used (as

for example Response Time Analysis), then this relaxation of the system parameters preserves system feasibility and still permits to mitigate the effects of the transient overrun on the affected tasks.

- *Inhibition*: this policy can be used when the occurrence of transient overruns is deemed unacceptable. If the task structure is realised as the composition of a synchronisation protocol agent (OBCS), a thread, and a functional container (OPCS), as in figure 1, which mirrors HRT-HOOD [4], then it is possible to prevent further execution of the overrunning task. Threaded entities in that model entirely delegate their provided interface (PI) to the OBCS for access control: each invocation is enclosed in a request descriptor that gets enqueued at the OBCS. At each execution, the thread fetches a request descriptor from the OBCS, interprets it and invokes the corresponding procedure in the OPCS. If the thread is sporadic, the call it makes to fetch request descriptors resolves to an entry in the OBCS, where it blocks if the pending queue is empty; if the thread is cyclic instead, it queries the OBCS queue with a protected procedure, which always returns a request descriptor (a default one, in case no other requests had been posted since the last access). We can therefore prevent any further execution of an overrunning task by setting to *false* the guard of the entry of the OBCS. The RP prescribes that a guard of an entry is composed exclusively by a simple Boolean variable. We can therefore set the guard to *false* using an ad-hoc protected procedure. The advantage of this solution is that it is reversible and it directly applies to sporadic tasks; in fact, it can also apply to cyclic tasks if the code pattern was changed accordingly and the cyclic OBCS was equipped with an entry that might occasionally be closed by the fault handling authority.

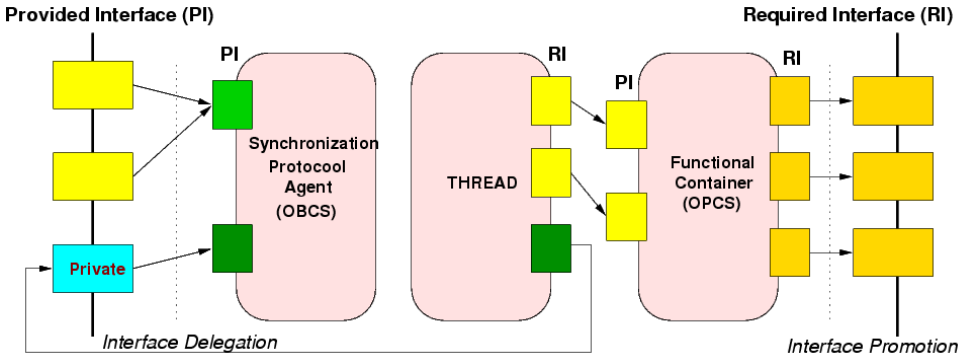

Figure 1: Compositional task structure.

The feasibility of the last two policies depends on the system requirements and in particular if the system can operate either with degraded performance or without the functional behaviour of the inhibited tasks. In several high-integrity real-time systems in fact, the only admissible recovery operation prescribes to safely terminate the program and restart it. If a degraded mode is allowed, attention should be paid to evaluating the producer-consumer relationships in which the inhibited task was involved.

Irrespective of that however, those two policies should be considered as basic ingredients to support mode changes in a Ravenscar-compliant system.

A permanent overrun is a much more severe situation to cope with. In the extreme case, under this kind of fault, a task executes forever as long as it is elected for execution (we may for example say it is stuck in an infinite loop). This kind of fault is critical since the RP does not provide any construct that addresses it. Due to fixed-priority preemptive scheduling, all tasks that have a priority equal to or lower than the overrunning task will never be able to execute again and they will thus miss all their deadlines. It is worth noticing that closing the guard of the OBCS is not useful in this case, since the overrunning task will never terminate its current activation, and thus it will never be enqueued again in the entry of its OBCS.

The solution possibly permitted in an ordinary RTOS would be to kill the overrunning task. Task termination is disallowed in the RP and with good reason, since this greatly reduces the complexity of the implementation and makes the system more fragile.

Hence we only see two remedial solutions:

- *Dynamic priorities*: the priority of the running task is reduced to a background priority to allow other tasks to eventually have opportunity to execute.
- *Kernel API*: the kernel exposes a specific API that permits the selective inhibition of the task. Ideally, with this API it would be possible to flag a task (or a set of tasks) to be considered *non-executable*. Flagging a task that is currently executing would cause an immediate task dispatching point in which all *non-executable* tasks would not be considered. Additionally, the *non-executable* flag shall be reversible. Obviously, the 'immediacy' of the relegation must not jeopardize protected resources, which may require either deferring the measure until the resource is freed or else forcing release if the application was able to restore a safe state for the affected resources.

For what concerns the priority lowering of the overrunning task, we can argue that it is effective from the point of view of system schedulability; in fact it preserves the schedulability of lower-priority tasks. Nonetheless, this solution is not completely satisfactory from the point of view of data integrity, since the faulty task is continuing its execution and thus may, for example, produce incorrect data which could be later used by healthy tasks. Additionally, this solution would require the kernel to provide a full implementation of dynamic priorities, which would increase its complexity and thus both footprint and time overhead. The full availability of dynamic priorities to applications, although potentially useful, for example, in mode changes, might also greatly increase system complexity from the timing analysis point of view.

For those considerations, we prefer the dedicated API solution, which we deem to have a limited impact on the kernel complexity, a predictable execution cost and does not conspicuously deviate from the RP rationale. The idea behind such a solution is clearly quite close to dynamic priorities. The procedure for the selective inhibition of tasks could be, in principle, included in the Ada `Dynamic_Priorities` package, provided that it could be somehow implemented and used without incurring the whole package overhead.

Table 3 summarises both the effectiveness and the compliance to the Ravenscar profile of the discussed techniques against WCET overruns.

Techniques	Transient WCET Overrun	Permanent WCET Overrun	Ravenscar Compliance
Error Logging	•	○	yes
Second chance	•	○	yes
Period/MIAT change	•	○	yes
Inhibition via OBCS	•	○	yes
Task termination	○	•	no
Dynamic priorities	○	•	no
Kernel API	○	•	no

Symbols: • = possible remedy to such temporal fault;
○ = unable or inappropriate to cope with such temporal fault.

Table 3: Techniques against WCET overruns.

6 Conclusions

The restricted subset of the Ada language constructs allowed by the Ravenscar Profile is insufficient both to monitor the timing properties of a system, and to effectively react to detected violations of such properties. We contend that a new profile that augments the RP with the new Ada 2005 language constructs for time monitoring (i.e., `Timers` and `Group_Budgets`), even though declared at library level, as suggested in [17], would allow the run-time enforcement of the timing properties of a system.

We illustrated and evaluated two architectures of interest that facilitate the monitoring of timing properties, abiding by the RP, extended with timers and group budgets.

Although `Timers` alone are actually able to exhaustively detect WCET overruns, we maintain that also `Group_Budgets` should be allowed. In fact they provide a system-level definition of logical partitions and thus offer, in case of misbehaviour, a straightforward way to propagate corrective actions within partitions. Furthermore, in some cases, a task overrun might be tolerated as long as it does not exceed the slack budget of the group it belongs to.

Given run-time mechanisms to detect violation of timing properties, one must still decide how to usefully react to such events. In particular, we distinguished between transient and permanent WCET overruns. Whereas several techniques can be adopted to handle the former, no Ravenscar-compliant technique could be identified to cope with the latter class of faults. Provided that task termination is disallowed in the RP, we suggested to implement a dedicated kernel API to selectively inhibit faulty tasks.

We still maintain that property preservation is of utmost importance for the reliable verification of systems in the target the domain and thus it cannot be disregarded in the definition of a restricted subset of the Ada language constructs. In future work we plan to investigate preservation of properties also in other non-functional dimensions, especially with respect to communication and space.

References

[1] S. Baruah and A. Burns. Sustainable Scheduling Analysis. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 159–168, 2006.

[2] M. Bordin and T. Vardanega. Automated Model-Based Generation of Ravenscar-Compliant Source Code. In *Proc. of the 17th Euromicro Conference on Real-Time Systems*, 2005.

[3] A. Burns, B. Dobbing, and G. Romanski. The Ravenscar Tasking Profile for High Integrity Real-Time Programs. In *Reliable Software Technologies - Ada Europe*, 1998.

[4] A. Burns and A. J. Wellings. *HRT-HOOD: A Structured Design Method for Hard Real-Time Ada Systems*. Elsevier, 1995.

[5] A. Burns and A. J. Wellings. Programming Execution-Time Servers in Ada 2005. In *Proc. of the 27th IEEE Real-Time Systems Symposium*, 2006.

[6] R. I. Davis and A. Burns. Hierarchical Fixed Priority Pre-emptive Scheduling. In *Proc. of the 26th IEEE Real-Time Systems Symposium*, pages 389–398, 2005.

[7] O. M. dos Santos and A. J. Wellings. Blocking Time Monitoring in the Real-Time Specification for Java. In *The 6th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 135–143, 2008.

[8] M. González Harbour and J. C. Palencia. Response Time Analysis for Tasks Scheduled under EDF within Fixed Priorities. In *Proc. of the 24th IEEE Real-Time Systems Symposium*, pages 200–209, 2003.

[9] ISO SC22/WG9. Ada Reference Manual. Language and Standard Libraries. Consolidated Standard ISO/IEC 8652:1995(E) with Technical Corrigendum 1 and Amendment 1, 2005.

[10] M. Joseph and P. K. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390–395, 1986.

[11] J. A. Pulido. *Arquitectura de Software para Sistemas de Tiempo Real Particionados*. PhD thesis, Universidad Politécnica de Madrid, July 2007.

[12] J. A. Pulido, J. A. de la Puente, J. Hugues, M. Bordin, and T. Vardanega. Ada 2005 Code Patterns for Metamodel-based Code Generation. *Ada Letters*, XXVII(2), 2007.

[13] J. A. Pulido, S. Urueña, J. Zamorano, T. Vardanega, and J. A. de la Puente. Hierarchical Scheduling with Ada 2005. In *Reliable Software Technologies - Ada-Europe*, 2006.

[14] L. Sha, J. P. Lehoczky, and R. Rajkumar. Solutions for Some Practical Problems in Prioritized Preemptive Scheduling. In *Proc. of the 7th IEEE Real-Time Systems Symposium*, pages 181–191, 1986.

[15] B. Sprunt. Aperiodic Task Scheduling for Real-Time Systems. In *Ph.D. Dissertation, Dept. of Electrical and Computer Engineering, Carnegie Mellon University*, 1990.

[16] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-Time Environments. *IEEE Transactions on Computers*, 44(1):73–91, 1995.

[17] J. Zamorano, A. Alonso, J. A. Pulido, and J. A. de la Puente. Implementing Execution-Time Clocks for the Ada Ravenscar Profile. In *Reliable Software Technologies - Ada-Europe*, pages 132–143, 2004.

[18] J. Zamorano, J. F. Ruiz, and J. A. de la Puente. Implementing Ada.Real_Time.Clock and Absolute Delays in Real-Time Kernels. In *Reliable Software Technologies - Ada-Europe*, pages 317–327, 2001.