Preservation of Timing Properties with the Ada Ravenscar Profile

Enrico Mezzetti, Marco Panunzio, and Tullio Vardanega

Department of Pure and Applied Mathematics, University of Padova, Italy
{emezzett,panunzio,tullio.vardanega}@math.unipd.it

Abstract. Modern methodologies for the development of high-integrity real-time systems leverage forms of static analysis that gather relevant characteristics directly from the architectural description of the system. In those approaches it is paramount that consistency is kept between the system model as analyzed and the system as executing at run time. One of the aspects of interest is the timing behavior. In this paper we discuss how the timing properties of a Ravenscar compliant system can be actively preserved at run time. The Ravenscar profile is an obvious candidate for the construction of high-integrity real-time systems, for it was designed with that objective in mind. Our motivation was to assess how effective the Ravenscar profile provisions are to the attainment of property preservation. The conclusions we came to was that a minor but important extension to its standard definition completes a valuable host of mechanisms well suited for the enforcement and monitoring of timing properties as well as for the specification of handling and recovery policies in response to violation events.

1 Introduction

In recent years, methodologies for the development of high-integrity real-time systems have started to adopt styles that leverage forms of static analysis mostly based on an architectural description of the system. One of the core concerns of those development methodologies is to facilitate the early analysis of the design attributes that are critical to the computation, time, space and communication behavior of the system. The adopted architectural description language and the methodology that uses it should therefore permit the required forms of analysis to be performed as early as possible in the development process, typically much earlier than implementation and test on target.

This prerequisite is important, because when design attributes are used as input for system analysis, they later constrain system execution in order that the analysis assumptions can actually (continue to) hold true at run time. Ultimately therefore those design attributes turn into system properties. Preservation of properties at run time then becomes an essential provision to warrant consistency between the system as analyzed and the system during execution. In fact, any deviation that the system execution may incur at run time from the initial stipulations may invalidate the analysis results and cause undesirable consequences. Obviously, the nature of the attributes that the analysis techniques in use want set on the system determine how strict and taxing the preservation measures must be. A simple yet coarse analysis may demand little in the way of

J. Real and T. Vardanega (Eds.): Ada-Europe 2010, LNCS 6106, pp. 153–166, 2010.

run-time preservation capabilities, but it may also result in ineffective design. More sophisticated analysis costs more in both the intellectual gears required of the user and the support needed for preservation, but it also permits finer-grained control of the design.

In this paper we focus on how to ensure the preservation of timing properties at run time. This goal can be achieved with three distinct and complementary provisions:

1. enforcing the timing properties that are to stay constant during execution;
2. monitoring the timing properties that are inherently variable in so far as they are influenced or determined by system execution;
3. trapping and handling the violations of asserted properties.

In class 1 we include the provisions needed to enforce the period for cyclic tasks and the minimum inter-arrival time (MIAT) for sporadic tasks. Those values are stipulated as constants that feed schedulability and performance analysis and they must therefore be obliged as exactly as possible during execution. Needless to say, the granularity and accuracy of the software clock and the absolute vs. relative nature of delays have a tremendous influence on the degree of preservation that can be attained.

Provisions in class 2 concern the monitoring of the execution time of tasks and their deadlines. Both attributes are intrinsically variable, the latter because, while we may set deadlines as relative, when they take effect they are obviously absolute and thus depend on current time.

The ability to monitor those variable attributes is the prelude to being able to detect the violations of their bounds as well as identify the failing party. The bound assumed for task execution time is the worst-case value of it, known as WCET, which should not be overrun. Deadlines should not be missed, which we can detect by observing whether the jobs of tasks always complete before their respective deadline.

Those provisions belong in class 3, together with the ability to act on the violation event following some user-defined policy.

In this paper we discuss a practical strategy to attain the preservation of the timing properties of interest. We want this strategy to be effectively applicable under the constraints of the Ravenscar profile [1], which we regard as the most appropriate run-time infrastructure for high-integrity real-time systems.

We also contend that the ability to monitor the execution time of tasks is crucial in two distinct ways: it helps us adjudge the cause of a timing-related violation event with suitable accuracy and it permits to trigger the designated handling procedure with the least possible latency.

The ability to monitor execution time responds to two important yet basic needs:

– First, we must acknowledge that the worst-case execution time (WCET) of tasks is a most fundamental input to schedulability analysis. Designers are required to feed the analysis equations with a value that is both *safe*, that is, no less than the actual task WCET, and as *tight* as possible so as to avoid overly pessimistic analysis results. Unfortunately, obtaining bounds with both those two characteristics is a tough call for both the scientific community and the current industrial practice. Hence systems are statically analyzed on the basis of bounds that may prove unsafe in some possibly extreme scenarios of execution, whether normal or erroneous, and thus incur WCET overruns that invalidate the assurance of analysis.
  The use of execution-time timers allows to promptly detect such overruns.

– Second, execution-time monitoring serves industrial developers most practical, effective and standard means to measure the execution of tasks in the most accurate and representative settings with respect to both hardware and operational scenarios. These measurements, when obtained through high-coverage verification and validation activities provide useful confirmatory evidence of the WCET bounds used in the analysis.

The remainder of the paper is organized as follows: in section 2 we briefly recall the essentials of the Ravenscar profile and account for its ongoing evolution; in section 3 we show how to enforce static timing properties; in section 4 we discuss how to monitor variable timing properties; in section 5 we propose some policies to detect and handle violation events; in section 6 we draw some conclusions.

2 The Ravenscar Profile

The Ravenscar profile (RP) [1] was one of the most prominent outputs of the 8th International Real-Time Ada Workshop (IRTAW), held in 1997. It was subsequently subject to minor refinements and clarifying interpretations during the 9th and 10th IRTAW in 1999 and 2002 respectively.

The RP has ever since received growing attention by the scientific, user and implementor community alike. Several industrial-quality implementations of it exist to date. Furthermore, with the 2005 revision of the Ada language it has also become a standard part of the language.

The rationale for the RP is to provide a restricted tasking model suited for the development of high-integrity real-time systems.

The Verification and Validation (V&V) activities for that class of systems include the use of static analysis to analyze the behavior of the system in the time and space dimensions. To best serve this need, the profile: excludes all Ada constructs that are exposed to non-determinism or unbounded execution cost; prescribes the use of a static memory model; and constrains task communication and synchronization to the use of protected objects under the ceiling locking protocol. The resulting run-time system can be implemented on top of a real-time kernel of little complexity – which is good for certification – and high space and time efficiency.

In our timing properties preserving architecture we use the following Ada constructs and features:

(i) the `delay until` statement, for the enforcement of the period of cyclic tasks and the MIAT of sporadic tasks (see section 3 for details);
(ii) `Timing_Event` declared at library level, for deadline monitoring (see section 4.1); and
(iii) execution-time timers for monitoring task WCET (see section 4.2).

`Timing_Events` and execution-time timers were introduced in the 2005 revision of the language, together with the standard definition of the RP. `Timing_Events` were – with definite benefit, as we shall see later – included in the Ravenscar profile, but under the restriction that they be declared at library level only [2]. Conversely, although the need

to monitor the run-time behavior of tasks even under the Ravenscar constraints was evident, execution-time timers were excluded, for it was feared that the asynchronous nature of timer events would hamper the predictability of execution and cause a disturbing increase in the run-time overhead of implementations.

Interestingly, the cost-related element of the cautionary argument behind the exclusion of execution-time timers from the RP does not hold anymore: several industrial-quality implementations of the RP have recently been extended with (restricted) experimental support for it, e.g.: MarteOS [3] and ORK 2.1 [4]. The latter implementation is a real-time kernel developed by the Polytechnic University of Madrid, which targets the LEON2[1], a SPARC V8 processor. ORK 2.1 provides a lightweight implementation of execution-time timers, restricted to at most one per task. To facilitate use in high-integrity systems, ORK 2.1 provides a very comprehensive score of upper-bounds to the timing overheads of all its primitive services, including those for execution time monitoring.

Table 1 summarizes the run-time overhead incurred by the timer management procedures of the `Execution_Time` package.

Table 1. ORK 2.1 time overhead of `Execution_Time` procedures (in processor cycles)

Package	Procedure	Execution time
`Execution_Time`	`Clock`	435
`Execution_Time`	`CPU_Time + Time_Span`	58
`Execution_Time`	`CPU_Time − Time_Span`	58
`Execution_Time`	`CPU_Time < CPU_Time`	65
`Execution_Time`	`CPU_Time ≤ CPU_Time`	77
`Execution_Time`	`CPU_Time > CPU_Time`	73
`Execution_Time`	`CPU_Time ≥ CPU_Time`	51
`Execution_Time`	`Split(CPU_Time, Seconds_Count, Time_Span)`	1142
`Execution_Time`	`Time_Of(Seconds_Count, Time_Span)`	80

In spite of the negative effect caused the very poor clock registers provided in the LEON2 processor architecture, the overheads reported in Table 1 arguably demonstrate, from the standpoints of both implementation and execution, that the inclusion of execution-time timers can be afforded in the Ravenscar profile. The subsequent discussion will also show that the other concern, that of permitting asynchronous timer events to unduly occur during execution is defeated under the use that we propose.

Acknowledging this evidence, the 14th IRTAW held in October 2009 formalized an Ada Issue (AI) proposal for the inclusion of *execution-time timers* in the standard definition of the Ravenscar Profile. At the time of writing, that AI has been submitted to the approval of the Ada Rapporteur Group (ARG) for evaluation.

We look very favorably to this possible revision of the RP. As we discuss in the sequel in fact, the availability of execution-time timers is absolutely central to the suite of run-time mechanisms we need for the realization of detection and handling of time-related faults in a Ravenscar-compliant system.

There has been heated (yet amicable) discussion as to whether the RP should stay fixed as sanctioned in the Ada 2005 standard and let any extensions (as opposed to

[1] `http://www.gaisler.com`

general modifications) of it form a distinct profile. Our view in that regard is that if a language feature is deemed useful for the intent and purposes of the RP and its implementation incurs low-enough space and time overhead, then it should incrementally add to the standard RP instead of feeding a separate derivative profile.

3 Enforcement of Timing Properties

The first class of timing properties we described in section 1 comprises constant properties that can be enforced explicitly, like the task period in cyclic tasks and the MIAT in sporadic tasks.

The task period can be straightforwardly enforced with the use of an absolute delay, as supported by the `delay until` statement of Ada, which separates in time successive activations of jobs of that task, each job being represented by the inside of the outermost loop of task body.

To enforce the MIAT in sporadic tasks we need instead to combine the use of the absolute delay with a task structure that captures the software-generated release event associated to the task. It is worth noting that by adopting a two-staged strategy for the handling of hardware interrupts and sporadic tasks for the deferred (or second-level) part we attenuate – but not obliterate – the hazard occurring from interrupts occurring more frequently than stipulated.

For example, reference [5] defines tasks as a composition of four basic blocks (Figure 1) that mirrors HRT-HOOD [6]:

- a provided and required interface, respectively PI and RI;
- an operational control structure (OPCS), which implements the functional (sequential) behavior of each PI service;
- a thread, which implements the task behavior and thus executes PI services of the task, one per activation as required;
- an object control structure (OBCS), which operates as the synchronization protocol agent responsible for delivering the release event to the task, in conformance to the Ravenscar constraint that wants inter-task communication to be asynchronous via a protected object.

In that task structure, the PI of the task is entirely delegated to the OBCS: each invocation of that PI is therefore addressed to the OBCS, which reifies it into a *request descriptor* and posts it in a dedicated application-level queue.

The release event of a sporadic task is determined by the occurrence of two subordinate conditions: the task has woken up from its MIAT-long suspension; and at least one invocation request is pending in the OBCS queue. When the latter condition is true, the guard to the corresponding entry in the OBCS opens and the thread may fetch with mutual exclusion guarantees the request descriptor from the OBCS queue. The thread then interprets the descriptor to determine which PI must be executed, and calls the designated service in the OPCS (see listing 1).

Cyclic tasks can be reconciled with this structure by requiring that at every periodic release they use a protected procedure to fetch a request descriptor from their OBCS. This provision enables the cyclic task structure to allow the execution of commanded

operations instead of just the nominal periodic operation. In the latter case the request descriptor would take a default value, but explicit invocations of the cyclic task PI would cause non-default request descriptors to be deposited in the queue of the corresponding OBCS. Interestingly, the latency of execution of the commanded operation would not undermine its ultimate utility, for it would be bounded by the task period.

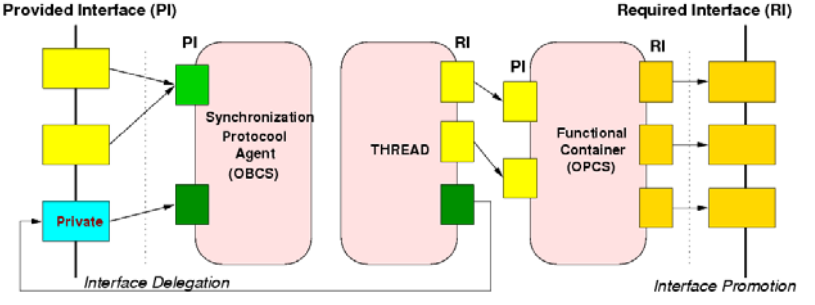

Fig. 1. Compositional task structure

Listing 1. Task structure for the enforcement of period or MIAT

```
1  loop
2    <fetch a request from OBCS and decode it>;
3    <invoke the required service in OPCS>;
4    Next_Time := Next_Time + Milliseconds (Interval);
5    delay until} Next_Time;
6  end loop
```

Much like many other real-time constructs, the efficacy of the `delay until` statement depends on the accuracy of the implementation of hardware and software clocks, and on the precision of the hardware timer. The lack of proper hardware support may negatively affect the accuracy of clocks and thus of absolute delays, as described in [7].

4 Monitoring of Timing Properties

Modern development approaches employ static analysis techniques to predict the timing behavior of the system model. Unfortunately, however, the values set for the task timing attributes that depend on run-time behavior – most notably, the worst-case execution time – may be exceeded. This is because the problem is difficult and prone to inaccurate reasoning or inadequate means. The consequences of a WCET overrun misbehavior may be dire, in that a number of tasks may miss their deadlines.

Let us now focus on the monitoring of deadlines, task WCET and blocking time induced by the use of the `Ceiling Locking` protocol to warrant mutual exclusion for shared critical sections.

4.1 Deadline Monitoring

Schedulability analysis ascertains whether every job of a task can complete its execution before the applicable deadline, which can be either absolute or relative to the release time of the task.

Since the expiration of an absolute deadline intrinsically is a time event, timing events can effectively be used to perform deadline monitoring.

In Ada 2005 the `Timing_Event` is a lightweight mechanism that executes a protected procedure at a specific time instant, without the need to employ a dedicated task. The code to be executed at the given time is specified in a `Timing_Event_Handler`. `Timing_Events` are triggered by the progression of the hardware clock. Implementations may execute the protected procedure directly in the context of the interrupt service routine that acknowledges the clock interrupt. This is in fact an implementation advice in the Ada specification.

Listing 2. Deadline-monitored task

```
1  loop
2     Set_Handler (Deadline_Event,
3                  Milliseconds (Rel_Deadline),
4                  Deadline_Miss_Handler);
5     <task operations>;
6     Next_Time := Next_Time + Milliseconds (Interval);
7     Cancel_Handler (Deadline_Event, isSetHandler);
8     delay until Next_Time;
9  end loop;
```

At each new task release, a timing event can be armed to expire at the absolute deadline of the task invoking the `Set_Handler` procedure (see listing 2). If the task was able to complete its activation before its deadline, the timing event would be cleared using the `Cancel_Handler` procedure. Otherwise, the timing event would be fired and the designated handler would be executed.

Unfortunately, very little can be learned from the detection of a deadline miss, for the violation event is not directly related with the actual cause of it. A deadline miss could in fact be incurred by a WCET overrun of the monitored task itself, or by greater interference from higher priority tasks (each element of it being a possibly marginal WCET overrun), or even by the blocking caused by the resource access protocol, when a lower priority task holds a resource with ceiling priority higher than that of the monitored task. As a consequence, no other useful operations can be performed for the handling of a deadline miss than just logging the event for the purposes of information integration over time.

4.2 WCET Monitoring

The provisions that enable the monitoring of execution time of tasks are probably the single most useful mechanism to ensure the preservation of timing properties. In Ada 2005, `Execution_Time.Timers` provide a simple yet efficient mechanism to monitor the execution time of tasks, to detect WCET overruns and to react in a timely fashion to a violation event.

Execution-time clocks were first introduced in the POSIX 1003.1d standard [8] as a means to cater for information on the run-time behavior of tasks. Execution-time clocks have subsequently been included in the 2005 revision of Ada [2]. The inclusion of execution-time timers in the Ada language standard is very good news indeed. Previously in fact, the industrial need for monitoring the execution time of tasks or for measuring execution time in general could only be responded to by resorting to vendor-specific solutions, realized in whether hardware or software.

The `Execution_Time` package associates a `Clock` to a designated task, which is used for measuring the CPU time actually consumed by the task execution. (To tell the truth, the language standard permits the measured value to include the execution time of interrupt handlers occurred during the execution of the task. This may obviously cause the resulting value to be pessimistically inaccurate. To rectify this discrepancy the 14th IRTAW formulated an implementation advice to treat the execution time of interrupt handlers separately from that of the preempted tasks.)

A `Timer` realizes a mechanism on top of an execution-time clock (thus related to a single task) which triggers a `Timer_Handler` procedure when the task has consumed a given amount of CPU time.

Listing 3. WCET-monitored task

```
1  loop
2     Set_Handler (WCET_Timer,
3                  Milliseconds (WCET),
4                  WCET_Violation_Handler);
5     <task operations>;
6     Next_Time := Next_Time + Milliseconds (Interval);
7     delay until Next_Time;
8  end loop;
```

Every individual task can be attached to a timer that monitors the CPU time that is consumed by the task. At each task activation the timer is set to expire whenever the task exceeds its allotted CPU time (which is meant to be its WCET). In the event of a WCET overrun the `Timer_Handler` procedure is immediately executed. In contrast with deadline monitoring, the handler need not be cancelled on self-suspension because a suspended task does not consume CPU time. Since the timer that has fired a handler on a violation event is directly attached to the overrunning task, a detected WCET overrun is always correctly ascribed to the actual culprit.

Under fixed-priority preemptive dispatching, a WCET overrun may cause a missed deadline not only in the overrunning task itself, but also on lower priority tasks owing to greater interference.

4.3 Blocking Time Monitoring

A task that executes longer than stipulated inside the critical section of a shared resource may cause a subtle case of timing fault. In fact, the *response time equation* (1) of any task τ_i is determined by three additive factors: the WCET C_i of τ_i itself; the worst-case interference due to the execution of higher priority tasks including kernel overheads (I_i); and the *blocking time*, which is computed as the longest time the task of interest

can be prevented from executing by lower priority tasks (B_i) in force of the resource access protocol in use.

$$R_i = C_i + B_i + I_i \tag{1}$$

In order to determine the blocking time factor that applies to every individual task we must therefore compute an estimate for the longest execution time of each critical section and then apply specific filtering scheme that depends on the resource access protocol in use. It is worth noting in this regard that the `Ceiling_Locking` policy, which is prescribed in the Ravenscar profile, provides a minimized bound for the blocking time factor value and guarantees that each task can be blocked at most once per activation and just before release.

This notwithstanding, detecting and diagnosing this kind of timing fault can prove quite cumbersome. The mere fact that a task executes longer than expected inside a critical section does not necessarily incur a WCET overrun in any affected task (the running task and that may suffer blocking from it) as the task execution as a whole may even out this violation. Consequently, to cope with this kind of fault we cannot simply rely on WCET monitoring through `Timers`.

As we inferred earlier on, a blocking-time violation can affect task schedulability in a subtler way than just causing WCET overruns in them. In fact, whereas WCET overruns only affect the schedulability of the faulty task or of lower priority ones, 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.

An interesting study [9] targeting Real-Time Java describes an elegant approach to directly monitor blocking time. The proposed solution leverages the inherent property of the `Ceiling_Locking` policy, which ensures that blocking may occur only once per task activation and just before its release. The essence of the proposal revolves around using a kernel-level timer to measure the time duration that a task is prevented from execution owing to priority inversion. Due to the lack of standard support for it, however that approach is currently not feasible in our context.

An alternative approach consists in measuring the execution time actually spent within shared resources instead of monitoring the blocking time incurred from their use. The worst-case blocking time term B_i in equation (1) depends on the adopted synchronization protocol; with the `Ceiling_Locking` policy prescribed by the Ravenscar profile – which has the same worst-case behavior as the priority ceiling protocol [10]) – the worst-case blocking time B_i induced on task τ_i amounts to the duration of the longest critical section executed by lower priority tasks in a shared resource with a ceiling priority higher than or equal to the priority of τ_i.

Unfortunately, to monitor the time a task executes inside a shared resource we cannot use the Ada `Timers`, for they are associated to only one task and this attachment is statically fixed at timer creation. Hence we cannot define a timer that can be reassigned to the tasks that enter a given critical section.

To circumvent the lack of direct language support, one might possibly resort to using the execution-time clocks on which `Timers` are based. An execution-time clock can in fact be used to measure task execution in any given code region. One could thus query the CPU time consumed by the task of interest before and after the critical section and then calculate the difference (see listing 4).

Listing 4. Time monitoring of execution in shared resource

```
1  Time_In := Execution_Time.Clock;
2  <beginning of critical section CS>;
3  <end of critical section CS>:
4  Time_Out := Execution_Time.Clock;
5  if Time_Out − Time_In > CS_WCET then
6     <violation handling>;
7  end if;
```

If the CPU time spent executing in the critical section is longer than estimated we may have a blocking-time violation for the higher priority tasks that contend for the same resource. This can be determined by comparing the consumed CPU time against the amount of blocking that the overruning task is predicated to induce on higher priority tasks; if that is the case, then we do have a violation event to treat.

Unfortunately however this approach suffers from at least two serious defects. First, the scheme shown in listing 4 would not be able to promptly detect a WCET overrun inside a critical region, but only after the task has finished executing in the shared resource and has released it, which may *not* actually occur in the case of serious programming or execution error. Furthermore, in contrast with the `Timer_Handler` procedure in `Timers`, the handling mechanism would not be executed at interrupt level but at the priority of the overrunning task, which is immediately preempted by higher priority tasks, perhaps even of those it was blocking. This implies that the handling of the fault is further deferred, possibly after the preempting task has already missed its deadline. Finally, this approach also adds considerable time and space overhead to the monitoring framework of the architecture.

5 Handling of Timing Faults

Several policies can be adopted to try to remedy a detected timing fault. The handling type may depend on: the severity and frequency of the timing fault; the criticality of the affected system function; system requirements. In fact, for several high-integrity real-time systems, the only recovery operation admissible on the detection of a severe fault requires to either terminate the program and enter some safe mode, or else switch to a hot redundant computer, if available.

We now enumerate the fault handling policies that can be used when a timing violation is detected at run time. Those policies can be applied in the face of occasional or even recurrent overruns of modest or even important gravity. Some of the proposed treatments simply contribute information for fault identification and diagnosis. Other treatments permit to effect a recovery action, which is able to mitigate or remedy the effects of the timing fault.

However, as we further discuss, all of those policies are unable to remedy *permanent overrun* situations that arise from a task getting stuck executing forever in a loop.

Error logging. This is the simplest and most basic treatment: the WCET violation event is simply logged. Although the logging does not remedy the problem, the log can be used to inform some designated fault handling authority, which can then apply some system- or partition-wide policy.

Integration of WCET. When designers perform schedulability analysis based on Response Time Analysis [11], they determine the worst-case response time of each task and thus earn confidence that tasks meet their deadline. As we mentioned earlier on, the robustness of the analysis results depends on the safeness of the WCET bounds that are fed to the equations.

Sensitivity analysis (for example in the flavor described in [12]) instead is a theory that is able to calculate how long the execution of a task can exceed its WCET while still maintaining the overall system schedulable. In essence, provided that all the remaining timing parameters stay fixed, we are able to statically calculate the maximum amount of tolerance the system can admit for single violations of WCET.

We can leverage this information to realize some WCET overrun handling policies.

Let us call ΔC_x the allowable excess execution time for task τ_x that we determine using sensitivity analysis. When task τ_i should ever incur a WCET overrun, perhaps because the estimated WCET bound C_i was too optimistic, we can permit the task to execute until $C_i + \Delta C_i$ without fearing consequences for system schedulability. The timing fault incurred when execution time exceeds C_i should however be always notified to the fault management authority so that it can set an upper bound on the number of times the violation is allowed before escalating to other handling policies (such as e.g., degraded mode, safe mode, etc...).

If a task should frequently incur WCET overruns that do not exceed ΔC_i, an obvious alternative strategy would consist in directly increasing the WCET bound that is monitored by the execution-time timer of the task. The increment that integrates the WCET can be applied to one and the same task multiple times as long as it does not exceed ΔC_i. Unfortunately, if we wanted to apply this policy to more than one task at a time, we would need to recalculate the ΔC_x increment factor for all tasks τ_x.

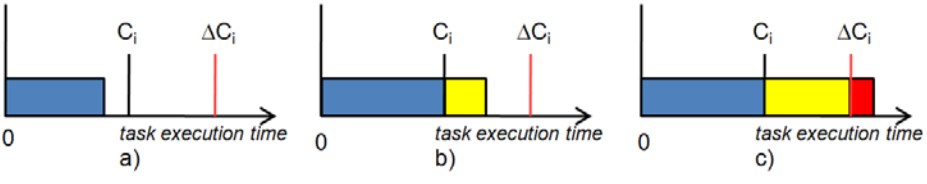

Fig. 2. a) Nominal execution of a task; b) The task overruns the WCET bound used in the schedulability analysis; the overall system however is still schedulable; c) The task overruns the WCET bound and the sensitivity analysis bound; the overall system is not schedulable anymore

The theory presented in [13] has potential for application in our context. That work in fact formalizes the class of "weakly hard real-time systems" and supports it with a suite of schedulability equations that are able to ascertain whether a task set meets "any n in m deadlines" (with $m \geq 0$ and $0 \leq n \leq m$), or "any row n in m deadlines".

In [14] instead, the authors leverage on [13] to determine the ΔC_i of a task in a weakly hard real-time system under EDF. Unfortunately, this theory is only able to predict the ΔC_i of a single task under the assumption that all other system parameters stay fixed. This limitation notwithstanding, an extension of that theory that was able

to account for shared resources and fixed priority preemptive scheduling would be an interesting candidate for application in our context.

Period or MIAT change. Increasing the period or MIAT of a periodic or sporadic task that is frequently overrunning its WCET may help one mitigate the effects on affected tasks. If a sustainable schedulability analysis theory [15] was used (as for example Response Time Analysis), then this relaxation of the system parameters is guaranteed to preserve the overall system schedulability.

Task inhibition via OBCS. If tasks are realized with the compositional structure described in section 3, it is possible to set the guard of the OBCS entry to false so as to prevent any further release of an overrunning task. In order to make the mechanism compliant to the RP, the guard of the entry shall be expressed as a simple Boolean variable; this is simple to achieve as the guard can be set by a designated protected procedure that can be invoked by the fault handling authority. The solution applies directly to sporadic tasks – and cyclic tasks alike – and it is reversible in that the guard can be set to true again anytime the fault handling policy deems it safe.

The applicability of the latter two policies is contingent on the system requirements: the system should be able to operate with degraded performances or without the functions in which the faulty tasks is involved; this assessment includes the evaluation of producer-consumer relationships in which the faulty task is involved. Table 2 recapitulates the possible policies and their essential characteristics.

Table 2. Techniques against WCET overruns

Technique	Recovery Action	Ravenscar Compliance
Error Logging	○	yes
Integration of WCET	∗	yes
Period/MIAT change	●	yes
Inhibition via OBCS	●	yes

Symbols: ● = the technique can be used as (part of) a recovery action;
∗ = the technique can be part of a recovery action in a limited number of situations;
○ = the technique does not remedy the timing fault.

The occurrence of a *permanent overrun* is an extremely severe situation to cope with. For this situation to occur, a task must be stuck executing forever in a loop. In that situation in fact, the rules of fixed-priority preemptive dispatching will never permit lower priority tasks to ever execute again.

A fault of this kind is extremely delicate for Ravenscar systems, since the RP does not provide for any mechanisms that permit to directly cope with it.

Task termination would be no solution, not only because it is explicitly excluded by the RP, but also because of its inherent exposure to massive extents of non-determinism and its disruptive costs to kernel implementation and verification.

The use of dynamic priorities and/or asynchronous task control could be advocated to mitigate or remedy the problem. The former feature would be used to decrease the priority of the offending task to the lowest possible value: this solution is however not

satisfactory for data integrity in so far as the task would stay eligible for execution and may consume data when the CPU has no other tasks to execute. The latter would not be able to force tasks stuck in a critical section to yield as the task in question shall first release the resource before asynchronous task control can take effect.

In the case the system was not able to direct perform ultimate maintenance on itself, patching part of the software (in actual fact, the functional part of offending tasks) while continuing reduced operation may become the only applicable non-disruptive course of action. The inhibition of designated tasks by setting the corresponding OBCS entry guard to false would permit to safely replace the faulty OPCS.

As a conclusion, there is still an open area of investigation for a practical and effective handling policy for this kind of severe faults. However it should be clear that due to the high-integrity nature of the real-time systems in which the RP is used (and thus to the extensive V&V campaigns they are subject to), we can assume that the probability of occurrence of permanent overruns is negligible.

6 Conclusion

In this paper we discussed the importance of preservation of properties at run time for state-of-the-art development methodologies. As the analysis of systems is applied in earlier design stages, it becomes imperative to ensure that the system at run time does not deviate from what was predicated by analysis. In our work we focused on run-time preservation of timing properties. We centered our approach on the adoption of a subset of the Ada language known as the Ravenscar profile (RP), which facilitates the design of systems that are by definition amenable to static analysis in the time and space dimensions. We described a framework for the enforcement and monitoring of timing properties which also allows to perform a set of fault handling and recovery actions.

The framework requires only three time-related constructs of the Ada language: the *delay until* statement, timing events and execution-time timers. The first two constructs already belong in the RP. The inclusion of execution-time timers in the RP, which at the time of this writing, is under evaluation by the Ada Rapporteur Group, would make the RP satisfactorily fit for monitoring WCET as well as providing the mechanisms to react to violation of that property.

Our study singled out two areas that need further investigation:

1. the monitoring of blocking time, which currently has no practical and satisfactory solution in Ada; it would be interesting to investigate the feasibility of a solution inspired to the proposal described in [9]. Alternatively, we might want to allow execution-time timers to be used for measuring the duration of critical sections.
2. permanent WCET overruns (caused for example by a task stuck in an endless loop) are critical in a Ravenscar system, since the profile does not provide any effective mechanism to cope with this situation; however, thanks to the intensive V&V campaigns that are routinely required by the high-integrity nature of the systems of our interest, we can assume that the occurrence of these faults has a negligible probability to occur.

In conclusion, we contend that in the context of high-integrity real-time systems the Ravenscar Profile is an excellent candidate to be used as the cornerstone of a development methodology. The RP in fact guarantees the development of statically analyzable systems and provides adequate means to ensure property preservation from design to implementation and eventually at run time.

References

1. Burns, A., Dobbing, B., Romanski, G.: The Ravenscar Tasking Profile for High Integrity Real-Time Programs. In: Asplund, L. (ed.) Ada-Europe 1998. LNCS, vol. 1411, p. 263. Springer, Heidelberg (1998)
2. 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)
3. Aldea Rivas, M., González Harbour, M.: MaRTE OS: an Ada Kernel for Real-Time Embedded Applications. In: Strohmeier, A., Craeynest, D. (eds.) Ada-Europe 2001. LNCS, vol. 2043, p. 305. Springer, Heidelberg (2001)
4. Universidad Politécnica de Madrid: GNATforLEON cross-compilation system, `http://polaris.dit.upm.es/~ork`
5. Bordin, M., Vardanega, T.: Automated Model-Based Generation of Ravenscar-Compliant Source Code. In: Proc. of the 17th Euromicro Conference on Real-Time Systems (2005)
6. Burns, A., Wellings, A.J.: HRT-HOOD: A Structured Design Method for Hard Real-Time Ada Systems. Elsevier, Amsterdam (1995)
7. Zamorano, J., Ruiz, J.F., de la Puente, J.A.: Implementing Ada.Real_Time.Clock and Absolute Delays in Real-Time Kernels. In: Strohmeier, A., Craeynest, D. (eds.) Ada-Europe 2001. LNCS, vol. 2043, p. 317. Springer, Heidelberg (2001)
8. IEEE Standard for Information Technology: Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) - Amendment 4: Additional Real-time Extensions (1999)
9. dos Santos, O.M., Wellings, A.J.: Blocking Time Monitoring in the Real-Time Specification for Java. In: The 6th International Workshop on Java Technologies for Real-Time and Embedded Systems, pp. 135–143 (2008)
10. Sha, L., Lehoczky, J.P., Rajkumar, R.: Solutions for Some Practical Problems in Prioritized Preemptive Scheduling. In: Proc. of the 7th IEEE Real-Time Systems Symposium, pp. 181–191 (1986)
11. Joseph, M., Pandya, P.K.: Finding Response Times in a Real-Time System. The Computer Journal 29(5), 390–395 (1986)
12. Bini, E., Di Natale, M., Buttazzo, G.: Sensitivity Analysis for Fixed-Priority Real-Time Systems. Real-Time Systems 39(1-3), 5–30 (2008)
13. Bernat, G., Burns, A., Llamosí, A.: Weakly Hard Real-Time Systems. IEEE Trans. Computers 50(4), 308–321 (2001)
14. Balbastre, P., Ripoll, I., Crespo, A.: Schedulability Analysis of Window-Constrained Execution Time Tasks for Real-Time Control. In: Proceedings of the 14th Euromicro Conference on Real-Time Systems, pp. 11–18 (2002)
15. Baruah, S., Burns, A.: Sustainable Scheduling Analysis. In: Proceedings of the 27th IEEE Real-Time Systems Symposium, pp. 159–168 (2006)