Weakly Hard Real-Time Systems

Guillem Bernat, *Member*, *IEEE*, Alan Burns, *Senior Member*, *IEEE*, and Albert Llamosí

Abstract—In a hard real-time system, it is assumed that no deadline is missed, whereas, in a soft or firm real-time system, deadlines can be missed, although this usually happens in a nonpredictable way. However, most hard real-time systems could miss some deadlines provided that it happens in a known and predictable way. Also, adding predictability on the pattern of missed deadlines for soft and firm real-time systems is desirable, for instance, to guarantee levels of quality of service. We introduce the concept of weakly hard real-time systems to model real-time systems that can tolerate a clearly specified degree of missed deadlines. For this purpose, we define four temporal constraints based on determining a maximum number of deadlines that can be missed during a window of time (a given number of invocations). This paper provides the theoretical analysis of the properties and relationships of these constraints. It also shows the exact conditions under which a constraint is *harder* to satisfy than another constraint. Finally, results on fixed priority scheduling and response-time schedulability tests for a wide range of process models are presented.

Index Terms—Real-time systems, specification, temporal constraints, scheduling algorithms, schedulability analysis, transient overload.

1 Introduction

Real-time system are those in which the temporal aspects of their behavior are part of their specification. The correctness of the system depends not only on the logical results of the computation, but also on the time at which the results are produced. This is due to the fact that these systems interact with their environment, which also changes with time. Real-time systems can be found in many different areas from control systems to multimedia video conferencing.

Most real-time programs are not intended to terminate, but to loop endlessly, examining input data and reacting accordingly. Therefore, the specification of real-time systems relies on this cyclic nature and is expressed in terms of operations—also called tasks or jobs—that have to be performed repetitively and finish within a given deadline.

Traditionally, real-time systems (or tasks) are classified as being *hard* or *soft*. For hard real-time tasks, it is imperative that no deadline be missed, while, for soft tasks, it is acceptable to miss some of the deadlines occasionally. It is still valuable for the system to finish the task, even if it is late, or just not finish that particular invocation. There is another class of tasks called *firm* tasks. Firm tasks are also allowed to miss some of their deadlines, but, as opposed to soft tasks, there is no associated value if they finish after the deadline.

In practical engineering contexts, the occasional loss of some deadline can be tolerated. This is either because the consequences of the loss can be negligible (due to the fact that the robustness of the involved control algorithms implies the ability to react properly at the next invocation step without serious consequences) or because the effect of a single missed deadline is not noticeable by a user (as it would be in the case of a missed audio packet in a video conferencing system). In this case, the quality of the service (QoS) provided would be lower.

Nevertheless, the term *occasional* is so ambiguous that it has no practical meaning for a specification. The extent to which a system may tolerate missed deadlines has to be stated precisely. In addition, the way that these missed and met deadlines are distributed is important. Some systems, for instance, are very sensitive to the consecutiveness of the missed deadlines. A missed deadline may correspond to a gap in the output signal and, if these gaps occur consecutively, the quality of the output is lower than if those gaps are nonconsecutive.

Not having to meet *every* deadline allows the capacity of the system's resources to be smaller than it would be for meeting all of them. This permits the creation of simpler and more cost effective systems that make better use of the available resources while guaranteeing, in the worst case, a minimum level of service. Moreover, as tasks generally run in less time than their worst case execution time, it is expected that, at run-time, some deadlines that were assumed to be missed will be eventually met.

The main contribution of this paper is introducing temporal constraints for specifying tasks that can tolerate a well-defined degree of missed deadlines and to present and study their properties and relationships.

1.1 Weakly Hard Real-Time Systems

We present an appropriate conceptual framework for specifying real-time systems that can tolerate occasional losses of deadlines. We call this type of system *weakly hard real-time system*. Formally:

Definition 1 (weakly hard). *A weakly hard real-time system is a system in which the distribution of its met and missed deadlines during a window of time w is precisely bounded.*

- *G. Bernat and A. Burns are with the Real-Time Systems Research Group, Department of Computer Science, University of York, Heslington, York, YO10 5DD UK. E-mail: {bernat, burns}@cs.york.ac.uk.*
- *A. Llamosí is with the Department of Mathematics and Computer Science, University Illes Balears, Cra. Valldemossa Km 7.5, 07071-Palma, Spain. E-mail: llamosi@clust.uib.es.*

Manuscript received 23 July 1999; accepted 23 Oct. 2000.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 110299.

Systems that must meet all of their deadlines are a particular case of our definition and will be referred to as *strongly hard real-time systems* whenever the distinction is relevant.

The effect of missed deadlines can be different depending on whether these deadlines are missed consecutively or nonconsecutively. For instance, in a computer controlled system, missed deadlines result in loss of control performance. This performance loss depends on the total number of missed deadlines over a period of time. On the other hand, in digital audio systems, the quality of the output produced is more sensitive to the number of consecutive missed deadlines. If the same number of missed deadlines occur nonconsecutively, its effects may not be noticeable. Consequently, it is required to introduce two classes of weakly hard constraints that specify whether the deadlines can be missed consecutively or nonconsecutively.

For symmetry purposes, it is also interesting to be able to specify the number of met deadlines as opposed the number of missed deadlines. For instance, in a certain control system, after a deadline is missed, it may be necessary to meet at least three deadlines in a row so that the state of the system is correctly updated. This introduces a further classification of weakly hard constraints.

To ease the notation, the tolerance to missed deadlines is established within a window of m consecutive invocations, which also corresponds to a window of $w = Tm$ units of time, where T is the period of the task.

The combination of the two considerations, 1) consecutiveness vs. nonconsecutiveness and 2) missed vs. met deadlines, leads us to four types of constraints. The simplest one is the $\binom{n}{m}$ (read *any* n in m). We say that a task meets *any* n in m deadlines if in any window of m consecutive invocations of the task, there are at least n invocations in any order that **meet** the deadline. This constraint and the other three: $\overline{\binom{n}{m}}$, $\langle \binom{n}{m} \rangle$, and $\langle \overline{\binom{n}{m}} \rangle$ are formally defined in Section 3.3. Although other types of constraints could be also defined, with these four constraints, and combinations thereof, it is possible to represent all real scenarios. See Section 3.8 for examples of the application of these constraints.

Note that the tolerance to missed deadlines in a real-time system cannot be adequately specified with a single parameter, for example, with the percentage of the number of deadlines to be met or missed (although it is common practice to do so). This is because, in general, a requirement like less than 10 percent of deadlines can be missed only represents average information over a large period of time. For example, it may mean that one deadline is missed every 10 task invocations or it may mean that 100 deadlines may be missed followed by 900 deadlines met, which is clearly not the same. To capture this situation, another parameter describing the window of time within which the number of deadlines must hold should be specified.

Depending on the application model, the notion of "missed deadline" can be associated to:

- **Delayed completion:** The task invocation runs until completion even though it finishes after the deadline.
- **Abortion:** The task invocation is terminated before finishing its computation either because it will not end its computation by the deadline or because another (more important) task needs the resource.
- **Rejection:** The task invocation is not accepted into the system.
- **Skip:** The task invocation is not released and the whole invocation is not executed.

1.2 Motivation Examples

Before going further in the definition and analysis of weakly hard real-time systems, it is important to argue what the effect of missing a clearly specified number of deadlines is and why it should happen in a predictable way. Different studies have addressed the topic under different assumptions, such as:

- Deadlines cannot be missed,
- Task periods are fixed and cannot be modified,
- Deadline equals period ($D = T$),
- Average response time of soft tasks has to be minimized.

Period and deadline values of a real-time task are defined at the specification stage. These values can be the result of a deep analysis of the system under control, as is the case of a computer-driven automatic control system; a desired sampling rate, as in the monitoring of a noncritical signal; or they may derive from perceived quality of service (QoS) requirements, as in a digital telephone system.

In order to argue in detail about these common assumptions and to analyze what the effects of missing deadlines in a real-time system are, we need to study in detail the semantics associated with a task and where the period and deadline parameters come from. We do this by studying three real-time scenarios, namely real-time control systems, real-time monitoring, and multimedia systems.

1.2.1 Computer-Driven Automatic Control Systems

In a computer-driven automatic control system, tasks periodically sample input signals, perform some computation according to some control algorithm, and send commands to some actuators. Control systems theory analyzes a system according to its physical characteristics and derives the parameters that guarantee both system stability and that show a steady state error within the tolerance bounds. Among these parameters, the sampling period, T (or sampling frequency $\omega = \frac{1}{T}$), plays an essential role in the schedulability of the system [1].

Ideal control systems assume that sampling and output occur *at exact* time instants $t_0 + kT$ ($k \in \mathbb{N}$). In fact, due to different delays (A/D, D/A conversion, sampling jitter, computation time and interference from other tasks), sampling and output occurs at any time between the activation time and the activation time plus the associated deadline. This effect produces an error similar to a quantization error (which can be also included in the

analysis of the system [2]). To overcome the errors associated with digital control, figures quoted in the literature [3] suggest that control system sampling rates should be between 5 and 10 (maybe 20 or even 40) times the bandwidth frequency (i.e., oversampling by a factor of 5 to 40). Another heuristic used for determining the minimum sampling rate is to guarantee that there are between 4 and 10 [4] samples during the rise time of the response of the system to a step input. Note that it is not always the case that increasing the sampling frequency of an already oversampled system will increase the control performance [4]. In fact, it may be just the opposite, large oversampling factors may reduce the control performance. This can be the case in a PID controller because of the very small differences between two consecutive samples.

This oversampling means that an occasional loss of a deadline can generally be tolerated. However, it is very important to guarantee that a certain number of deadlines are not lost in a row otherwise the system may become unstable.

The assumption that $T = D$ is inadequate for digital control. Automatic control systems require that $D \ll T$ so that both sampling and actuation occur at intervals as regular as possible [2].

1.2.2 Monitoring System

Another example of a real-time task is a monitor. A task periodically performs a set of monitoring actions. The sampling period may be carefully computed (and probably overestimated) or decided as a rule of thumb. Again, missing an occasional deadline means that the action from monitoring will be delayed by some bounded period of time. Provided that the effect of such a delay can be tolerated, deadlines can be missed.

As in the previous section, periods are generally defined as a rough estimation of the desired sampling rate, therefore, they may be reduced if design convenience requires.

1.2.3 Multimedia Systems

A multimedia system is one that exhibits different media simultaneously. From the real-time point of view, we are interested in media that changes as a function of time, for instance, digital audio and video. These systems are generally considered soft systems as missing a deadline has no catastrophic consequences unless very high fidelity has to be certified.

Consider, for example, a digital video system. The system decodes some video streams and plays the frames at some rate (say, 30 frames per second). If it does not have enough time to finish decoding a frame, it is either skipped or only partially displayed. Not displaying a frame on time is considered a missed deadline. In this case, a missed deadline results in a lower quality of the service provided by the computer. If it is not known how missed deadlines are distributed over time, several missed deadlines may occur in a row, which will be manifested by a frozen screen for a short while.

Another common problem in systems that can tolerate missed deadlines is related to using a buffering mechanism. Assume a buffer of some size and producer and consumer tasks that periodically add and remove elements from the buffer. If the consumer task misses its deadline, it may mean that the buffer has grown in size temporarily. The buffer will not overflow if there is a guaranteed maximum number of deadlines that can be missed in a row.

As a final note, real-time scheduling algorithms that schedule a mixture of hard and soft tasks usually partition the task in these two classes: hard and soft. As it cannot be guaranteed how many deadlines soft tasks may miss, the criteria commonly used for comparing their performance is the minimum average response time of soft (or firm) tasks, generally ignoring completely that those soft (firm) tasks do have a deadline as well. This actually leads to a situation in which hard tasks suffer high delays and high output jitter in order to give preference to soft (firm) tasks. This is an unfair mechanism because soft (firm) tasks are given preference over hard ones even though hard tasks may have shorter deadlines. A more sensible mechanism is to consider all tasks in the system to be weakly hard (some of them would still be strongly hard) and where firm and soft tasks have weakly hard constraints. A guaranteed scheduling algorithm may schedule all tasks according to their weakly hard constraints with the secondary objective, if required, of minimizing the average response time of the tasks (or a subset of them).

The cases analyzed in this section show that hard real-time systems are not that hard because some degree of missed deadlines can be tolerated by the algorithms themselves. On the other hand, soft real-time systems are not that soft as it is generally required to specify upper bounds on the number and pattern of deadlines missed during a period of time. The notion of weakly hard real-time system captures all of these scenarios in a unified way.

1.3 Scope of the Paper

The study of weakly hard real-time systems can be done from three points of view: the specification, the implementation, and the analysis.

The specification relates to setting weakly hard constraints on tasks which describe their desired or required temporal properties. For this purpose, it is necessary to study in detail the different temporal constraints and how they relate to each other, for example, by identifying when one weakly hard constraint is harder to satisfy than another. The semantics associated with setting weakly hard constraints are also an important issue which depends on the particular application domain.

At the implementation, the specified tasks are mapped into implementation threads which will be scheduled with a given scheduling algorithm (or even implemented as a cyclic executive). The scheduling algorithm can be any traditional fixed priority or dynamic priority algorithm or be based on weakly hard constraints like the enhanced dual priority scheduling [5], [6].

The analysis allows us to determine whether weakly hard constraints will be satisfied. This depends on the scheduling algorithm itself. There are two main scheduling strategies for the implementation of a real-time system. The scheduling strategy can be *guaranteed* or *best-effort*. In guaranteed systems, an offline analysis technique exists that actually assures that the temporal constraints will be

always satisfied. On the other hand, best-effort scheduling is not based on obtaining a priori guarantees that the constraints are going to be satisfied. Instead, it is based on maximizing (or minimizing) some criteria, for example, maximizing the number of weakly hard constraints satisfied.

For guaranteed systems, offline checks are required to determine the schedulability of the system. Section 4 describes such a check for a fixed priority scheduler. For best-effort scheduling algorithms, simulation is most often applied for obtaining average response times.

1.4 Previous Work

There have been some previous approaches to the specification and design of real-time systems that can tolerate occasional losses of deadlines.

Hamdaoui and Ramanathan in [7] present the notion of (m,k)-firm deadlines to specify tasks that are desired to meet m deadlines in any k consecutive invocations. They use this concept in the context of scheduling messages in a communication system. They present a scheduling algorithm based on raising the priority of a class of messages if it is likely that m messages of the class out of k consecutive messages are not going to meet their deadlines. This approach has, on average, a better behavior than a simple EDF approach. Their approach is a best-effort scheduling algorithm. It does not provide any guarantee on the number of deadlines a task may miss. They also assume that all tasks have the same m, k parameters. They use the concept of (m,k)-firm deadlines as a desired level of service as opposed to a guaranteed level of service. Another scheduling algorithm specifically addressed to (m,k)-firm deadlines is the work on window-constrained schedulers [8].

Another approach is the introduction by Koren and Shasha [9] of the *skip factor*, s. If a task has a skip factor of s, it will have one invocation skipped out of s. They introduce the D-over dynamic priority scheduling algorithm, which is based on deciding which task invocations to skip. Later, in [10], the same authors introduce two scheduling algorithms that exploit skips: the Red Tasks Only (RTO) and Blue When Possible (BWP). One interesting result is that making optimal use of skips is an NP-hard problem.

Koren and Shasha deal with overloaded systems. They reduce the overload by skipping some task invocations; the drawback of this approach is that, every s task invocations, one is skipped, although it may meet the deadline if it were scheduled. This work has been followed by Caccamo and Buttazzo in [11], where they schedule hybrid task sets consisting of skippable periodic and soft tasks. They present a scheduling algorithm based on EDF which exploits skips to minimize the average response time of the soft tasks. An improvement of the approach is to demote task instances to background priority instead of skipping them. This enables some skipped tasks to meet their deadlines.

It is known that, under overload conditions, deadline-based scheduling systems perform badly due to the domino effect [12]. This has led to the definition of scheduling strategies which rely on the value of the task rather than the deadline under overload conditions. For example, Buttazzo et al. in [13] study the value-based scheduling under overloads.

Other works are just based on mechanisms that integrate hard and firm tasks and that are able to deal with transient overloads, which implies that some deadlines (of the firm tasks) will be missed. The work of Spuri et al. [14] is an example. However, even though some invocations are rejected, there is no mechanism to quantify how many of them are actually being lost, neither to distribute the number of lost deadlines among the tasks.

Wedde and Lind [15] present a distributed operating system, called Melody, in which they introduce the notion of criticality and sensitivity as a measure of the number of consecutive deadlines a task may miss based on the history of missed deadlines of the task and associated scheduling policies. The scheduling approach, however, is only best-effort as it cannot guarantee that these constraints are satisfied.

There has been increasing interest in the real-time community for quality of service (QoS) based systems. QoS is generally used in multimedia and telecommunication systems. Different strategies allow a negotiated level of service to be offered for each of the tasks in the system. In most of the cases, current tasks compete for the available resources and a mechanism is necessary to distribute the resources between them. In these systems, it is generally assumed that deadlines can be missed which results in a reduction on the quality of the service provided (as in [16]).

The common feature of all these works is that they introduce the idea of systems which tolerate some missed deadlines, but they do that with a particular constraint and with a scheduling algorithm generally only applicable to these constraints, therefore, making it difficult to generalize their ideas and to be able to apply their work in a more general framework. We present a general framework for specifying weakly hard real-time systems. We later show that the approaches for specifying tolerance of missed deadlines shown above are particular cases of our definitions of weakly hard constraints. Based upon the analysis performed in this paper, several online scheduling algorithms can be devised.

The main emphasis of the paper is on the specification of weakly hard constraints and on the study of the properties and relationships between tasks with such constraints. However, we also discuss the issues of schedulability analysis. Therefore, the paper is organized as follows: The next section introduces the process model used for specifying the constraints. We first introduce a pure periodic model and then complete it with the specification of sporadic tasks. After this section, the formal definitions of weakly hard constraints are introduced and their properties and relationships studied in detail. We finally present the schedulability checks for fixed priority scheduled systems and the conclusions.

2 Process Model

In the real world, events happen in parallel and a system that interacts with this real world has to mimic this parallel behavior. This is usually done by building the system as a set of cooperating sequential operations that interact with each other and with the environment and that execute

continuously. Each of these operations models a small part of the system and can be either *periodic* or *nonperiodic*.

A periodic operation consists of a computation that is executed repeatedly in a regular and cyclic pattern. The duration interval between the start of one execution, the invocation time, and the next one is a constant, called the period (denoted by T). Nonperiodic operations are usually executed in response to asynchronous events. In order to be able to analyze the effect of nonperiodic operations, a measure of the minimum separation between arrivals is usually needed, which is also denoted by T.

These operations are usually called tasks or jobs. This term should not be confused with the same term when it is used in the implementation. A task in a specification context means a goal that must be achieved periodically. A task in an implementation context means an execution thread. Although a specification task can be mapped on an implementation thread, other alternatives exist (cyclic executives). In this work, a *task* refers to the operation concept when in the context of a specification and to the thread concept in the context of the implementation.

2.1 Periodic Tasks

The basic process model, from the specification point of view, consists of a set, Π, of N periodic tasks, where each task, τ_i is characterized by the period, T, deadline, D, and a weakly hard constraint, λ, which describes the tolerance to missed deadlines (the different weakly hard constraints are introduced formally in Section 3).[1]

$$\Pi = \{\tau_i = (T_i, D_i, \lambda_i)\}_{1 \leq i \leq N}.$$

All periodic tasks are assumed to be released together at time $t = 0$. In this case, the tasks will be released together again at the least common multiple of the periods of the tasks; this period of time is called the *hyperperiod* and it is denoted by $\mathbf{H}$:

$$\mathbf{H} = \mathrm{lcm}\{T_i \mid \tau_i \in \Pi\}.$$

We denote by $A_i = \frac{\mathbf{H}}{T_i}$ the number of activations of a task τ_i in the hyperperiod.

In some circumstances, however, the starting times of some of the tasks may not occur at $t = 0$. In this case, we also consider that a task may have an *offset*, denoted by O_i, which is the time elapsed before the first invocation of that task.

For a given task τ_i, we denote by $S_i(k)$, $k \geq 1$, the invocation time of the kth release. $S_i(k) = (k-1)T_i + O_i$.

We also use R_i to denote the worst case response time of the task under the chosen implementation schema. We study the behavior of the tasks at different invocations; for this purpose, we introduce $F_i(k)$ as an upper bound on the worst case finalization time of task τ_i at invocation k, that is, the latest time when the task may finish its kth invocation considering interference from other tasks. As the tasks are periodic, the interference received at invocations other than at the critical instant (when all the tasks are released together) is considerably lower. These values can only be computed after a scheduling strategy is chosen and worst case computation times of the tasks obtained. We also define $R_i(k)$ as the worst case response time of the task at invocation k: $R_i(k) = F_i(k) - S_i(k)$.

In general, $R_i(k) \neq R_i(l)$ for $k \neq l$ because the amount of interference a task may suffer from other tasks depends on the phasing with other tasks which is different at each invocation.

Note that, whenever it is clear from the context, the subindex i is sometimes omitted.

When we investigate schedulability tests for fixed priority systems in Section 4, the definition of the hyperperiod at level i will be needed. A task can only suffer interference from higher priority tasks and potentially a blocking due to lower priority tasks. We can therefore define the *hyperperiod at level i* as the hyperperiod only considering tasks of equal or higher priority than task τ_i. We denote the hyperperiod at level i by h_i and the number of invocations of a task in its hyperperiod by $a_i = \frac{h_i}{T_i}$.

2.2 Nonperiodic Tasks: Repeating and Isolated Tasks

We also consider that the system can contain nonperiodic tasks. Traditionally, hard nonperiodic tasks are called *sporadic*, while soft nonperiodic tasks are usually called *aperiodic*.[2] In our context, these definitions are not enough; we need to further distinguish between two classes of nonperiodic tasks: *repeating* and *isolated*.

On the one hand, repeating tasks are nonperiodic tasks which are continuously invoked, although this does not happen at fixed periods of time, but on a regular basis. Repeating tasks model quasi-periodic activity, such as the case of signals generated by an angular encoder in which the rate of arrivals of signals would depend on the angular velocity, which changes in time, or a video streaming system in which video and audio packets are sent through a network. The receiver node sees a stream of signals which are continuously arriving, but cannot be modeled as periodic due to the different sources of delay.

On the other hand, isolated tasks are nonperiodic tasks that are not expected to be invoked consecutively (even though they may have a minimum interarrival time for analysis purposes). Isolated tasks can be seen as the tasks attached to alarm or rare events in the system.

Periodic and repeating tasks may have weakly hard constraints assigned to them; however, it makes no sense to set weakly hard constraints to isolated tasks.

In the rest of this paper, τ denotes a periodic task, whereas σ denotes a nonperiodic task. A task which may be either periodic or nonperiodic is denoted by ρ. For repeating tasks, the vector $R_i(k)$ cannot be computed in the same way as for periodic tasks because arrival instants, $S_i(k)$ are not fixed.

1. Note that we do not include the worst case computation time C_i in this process model as it is not necessary from the requirements point of view. It it will be only necessary for checking the schedulability of the system.

2. This terminology is unfortunate and confusing as it does not represent the notion behind the tasks. However, they are accepted terms, so we will use them.

TABLE 1
Example Task Set

τ_i	T_i	D_i	C_i	R_i
τ_1	200	5	4	4
τ_2	25	25	2	6
τ_3	25	25	6	12
τ_4	40	40	3	15
τ_5	50	50	3	18
τ_6	50	50	5	23
τ_7	59	59	8	39
τ_8	80	80	9	75
τ_9^*	80	**80**	2	**97**
τ_{10}^*	100	**100**	5	**139**
τ_{11}	200	200	1	145
τ_{12}	200	200	3	148
τ_{13}	200	200	1	149
τ_{14}	200	200	1	150
τ_{15}	200	200	3	199
τ_{16}	1000	1000	1	200
τ_{17}	1000	1000	1	295

Tasks τ_9 and τ_{10} miss their deadlines in the worst case.

3 Specification of Weakly Hard Real-Time Systems

This section studies the way a bounded distribution of met and lost deadlines can be specified. For illustrative purposes, we use a task set scheduled under fixed priority scheduling. After this, we introduce the concept of μ-pattern and, based on this, we define the weakly hard constraints.

3.1 Example

The following case study, based on the one described by Locke et al. in [17] will be useful to illustrate our concepts. The parameters of the task set are shown in Table 1. For this task set, for all tasks τ_i, $O_i = 0$.

The reader should realize that the table also includes the columns C_i, holding the worst case computation time, and R_i, which holds the worst case response time of the tasks assuming fixed priority scheduling computed using response time analysis techniques [18], [19].

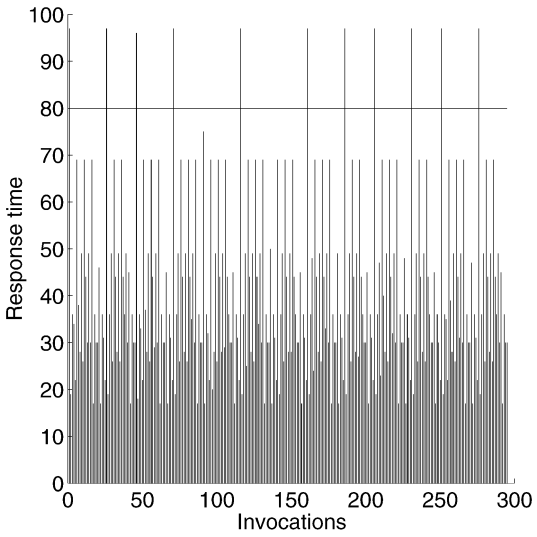

Fig. 1. Response times of task τ_9 having $D_9 = 80$ for all invocations in the hyperperiod at level 9. The task misses at most 11 deadlines in every 295 invocations.

Under fixed priority scheduling, the traditional worst case response time schedulability analysis would conclude that the system is not schedulable because tasks τ_9 and τ_{10} miss their deadlines in the worst case. However, even though tasks τ_9 and τ_{10} miss their deadlines at the critical instant, they meet most of the rest of their deadlines.

Fig. 1 shows $R_9(k)$, the worst case response time at each invocation k of task τ_9 (the way $R_i(k)$ is computed will be discussed in Section 4). Each vertical line represents the worst case response time of the task at invocation k. The deadline of the task is $D_9 = 80$. The hyperperiod is $\mathbf{H} = 118000$, whereas $h_9 = 23600$. This means that the task is invoked $a_9 = 295$ times within its hyperperiod at level 9. Table 2 shows the exact distribution of the missed and met deadlines for invocations 1 to 295 of task τ_9. The pattern is repeated cyclically every a_9 invocations. A 0 represents a missed deadline and a dot a deadline met.

The task does not meet the deadline at the critical instant at $t = 0$. However, the deadline is missed at most 11 times during its hyperperiod (11 times every 295 invocations);

TABLE 2
Exact Distribution of Missed and Met Deadlines

```
  1- 40: 0.... .....   ..... .....   ..... 0....   ..... .....
 41- 80: ..... 0....   ..... .....   ..... .....   0.... .....
 81-120: ..... .....   ..... .....   ..... .....   ..... 0....
121-160: ..... .....   ..... .....   ..... .....   ..... .....
161-200: 0.... .....   ..... .....   ..... 0....   ..... .....
201-240: ..... 0....   ..... .....   ..... .....   0.... .....
241-280: ..... .....   0.... .....   ..... .....   ..... 0....
281-295: ..... .....   .....
```

also note that the invocations when the task may miss its deadline are spread out regularly along the hyperperiod. The minimum separation between two consecutive deadlines missed is 20 invocations. This is because the task only misses the deadline when it overlaps with the invocation of a subset of higher priority tasks that produce enough interference to make the task miss its deadline. Note that this situation is very common in real-time systems made up of periodic tasks.

This example illustrates some typical properties of task executions: Even though some tasks may miss their deadline in the worst case scenario, the response times are bounded; in addition, the points in time when those deadlines may be missed are spread out and occur at predictable points in time.

3.2 μ-Patterns

As we showed in the previous section, we can characterize the way a task achieves and misses its deadlines by a binary sequence that we call the task's μ-pattern.

Definition 2. *The task's* μ-pattern *is the worst case pattern of met and lost deadlines of task* τ_i. *It is denoted by* μ_i *and it is defined as:*

$$\mu_i(k) = \begin{cases} 1 & if\ R_i(k) \le D_i \\ 0 & otherwise \end{cases}$$

$$where\ k \ge 1\ is\ the\ invocation\ count.$$

The analysis of μ-patterns will require the use of some elementary concepts of strings on the alphabet $\Sigma = \{0, 1\}$. As usual, we will denote as Σ^* the set of words over the alphabet Σ. We will call *sequences* the elements of $\Sigma^* \cup \Sigma^\infty$. Note that a μ-pattern is a sequence.

Given a sequence α, the length of the sequence, $|\alpha|$, is the number of symbols the sequence has; α_i is the ith symbol of α starting with $i = 1$. A slice or subsequence $\alpha(i..j)$ is the word obtained from α by getting the symbols starting at position i and ending at position j (both included). It is only defined for $1 \le i \le j \le |\alpha|$. Also, $\alpha(i, p)$ denotes the slice of length p starting at i, $\alpha(i..i + p - 1)$. A negative value of p in $\alpha(i, -p)$ denotes a slice that *ends* at i and has length p. This corresponds to $\alpha(i - p + 1..i)$. The set of slices of length p of a given sequence or word α is denoted by $W^p(\alpha)$:

$$W^p(\alpha) = \{\alpha(i, p)\} \qquad 1 \le i \le |\alpha| - p + 1. \tag{1}$$

3.3 Weakly Hard Temporal Constraints

Some systems are sensitive to the consecutiveness of lost deadlines while others are only sensitive to the number of deadlines missed. For each one, we have considered the points of view of both missed and met deadlines. This gives us four types of constraints:

Definition 3. *A task* τ *"meets any* n *in* m *deadlines"* ($m \ge 1$ *and* $0 \le n \le m$) *and it is denoted by* $\tau \vdash \binom{n}{m}$ *if, in any window of* m *consecutive invocations of the task, there are at least* n *invocations in any order that meet the deadline.*

Formally, the task satisfies $\binom{n}{m}$ if its μ-pattern satisfies:

$$\forall \alpha \in W^m(\mu),\ \alpha \text{ has at least } n \text{ 1s}.$$

Definition 4. *A task* τ *"meets row* n *in* m *deadlines"* ($m \ge 1$ *and* $0 \le n \le m$) *and it is denoted by* $\tau \vdash \langle {n \atop m} \rangle$ *if, in any window of* m *consecutive invocations of the task, there are at least* n *consecutive invocations that meet the deadline.*

Formally, the task satisfies $\langle {n \atop m} \rangle$ if its μ-pattern satisfies:

$$\forall \alpha \in W^m(\mu),\ \text{the word } 1^n \text{ is a substring of } \alpha.$$

Definition 5. *A task* τ *"misses any* n *in* m *deadlines"* ($m \ge 1$ *and* $0 \le n \le m$), *and it is denoted by* $\tau \vdash \overline{\binom{n}{m}}$ *if, in any window of* m *consecutive invocations of the task, no more than* n *deadlines are missed.*

Formally, the task satisfies $\overline{\binom{n}{m}}$ if its μ-pattern satisfies:

$$\forall \alpha \in W^m(\mu),\ \alpha \text{ has at most } n \text{ 0s}.$$

Definition 6. *A task* τ *"misses row* n *in* m *deadlines"* ($m \ge 1$ *and* $0 \le n \le m$), *and it is denoted by* $\tau \vdash \overline{\langle {n \atop m} \rangle}$ *if, in any window of* m *consecutive invocations of the task, it is never the case that* n *consecutive invocations miss their deadline.*

Formally, the task satisfies $\overline{\langle {n \atop m} \rangle}$ if its μ-pattern satisfies:

$$\forall \alpha \in W^m(\mu),\ \text{the word } 0^n \text{ is not a substring of } \alpha.$$

Notice that, for the $\overline{\langle {n \atop m} \rangle}$ n m constraint, the parameter m is not required because the satisfaction of the constraint does not depend on the size of the window. Therefore, we may also write $\overline{\langle n \rangle}$.

In this context, we will use λ to denote a constraint and Γ as the set of all possible temporal constraints of these four types. We will also refer to the n and m parameter of a constraint λ as $\lambda.n$ and $\lambda.m$. We also call the $\lambda.m$ the *window* of the constraint. Please note that the constraints are to be satisfied *for all subsequences* of length m. If the sequence has length shorter than $\lambda.m$, then we say the satisfiability is *undefined* (more details are given in next section). The constraints are summarized as follows:

	Met deadlines	Missed deadlines
Any order	$\binom{n}{m}$	$\overline{\binom{n}{m}}$
Consecutive	$\langle {n \atop m} \rangle$	$\overline{\langle {n \atop m} \rangle} \equiv \overline{\langle n \rangle}$

These constraints model other notations introduced previously by other researchers. Some examples are:

- The notion of (m, k)-firm deadlines of [7] maps directly to the $\binom{m}{k}$ constraint.
- The skip factor s of [9] corresponds to the $\overline{\binom{1}{s+1}}$ (or $\binom{s}{s+1}$).
- The criticality c and sensitivity s of [15] correspond to $\overline{\langle c \rangle}$ and $\overline{\langle s \rangle}$, respectively.

3.4 Weakly Hard Constraints for Repeating Tasks

Until now, we have only considered weakly hard constraints for periodic tasks. However, these constraints cannot be applied directly to nonperiodic tasks because of some fundamental differences between periodic and sporadic tasks. For example, consider a system in which an isolated task misses its deadline if it is activated at the critical instant. In the worst case, this isolated task may never meet a deadline as it might arrive at the multiples of the hyperperiod. Repeating tasks do not show this problem. For repeating tasks, the definition of weakly hard constraint has to be reformulated to specify the exact window of time in which the constraint has to hold.

For periodic tasks, we have considered the satisfiability of a given number of deadlines over a number of task invocations, that is, for a window of Tm time units. For repeating tasks, as the arrival instants of the tasks are not known, we cannot specify the size of the window as the number of invocations. Instead, it is necessary to specify a window of time in which the constraint has to be checked. This window of time has to be large enough to have at least m invocations of the task. This leads us to rewrite the weakly hard constraint for repeating tasks as follows:

Definition 7 (Weakly hard constraint for repeating tasks). *Given an interval of time w and a weakly hard constraint λ, we denote by λ_w a weakly hard constraint for a window time w. A given task ρ satisfies λ_w (denoted by $\rho \vdash \lambda_w$) if all the possible μ-patterns of task invocations within a window w satisfy λ.*

For periodic tasks, this definition is equivalent to Definitions 3 to 6 by considering $w = Tm$.

For a given time t, we consider the window of time $[t - w, t]$ and we obtain the slice of the μ-pattern that corresponds to the task invocations that have finished within this interval. Therefore, the repeating task will satisfy the constraint if, for all possible time instants t, the corresponding slice satisfies the constraint. Fortunately, we do not need to check all possible time instants, it is only required to check the time instants that correspond to the invocation or finalization of a repeating task. If μ is the μ-pattern of a repeating task ρ, where $\mu(k)$ corresponds to the kth invocation, then, at a given time t, the slice, denoted by $\mu^w(t)$, is given by all invocations of the task that have finished within this interval:

$$\mu^w(t) = \mu(l..r), \tag{2}$$

where $l = \min\{s \mid F(s) \geq t - w\}$ and $r = \max\{s \mid F(s) \leq t\}$. Therefore, the satisfiability of the constraint can be written as:

$$\rho \vdash \lambda_w \quad \Leftrightarrow \quad \forall t,\ \mu^w(t) \vdash \lambda_w.$$

A natural question arises as to whether the constraint is satisfied when the length of the slice is shorter than $\lambda.m$. This situation corresponds to the case when the task has been invoked less than $\lambda.m$ times in $[t - w, t]$. There are two possible answers depending on the way we evaluate the future. We borrow the terms of *lazy* and *eager* evaluation from functional languages theory.

Under this situation, there are not enough symbols in the slice to decide whether the constraint is satisfied. Under a *lazy* evaluation we would say that the constraint is *undefined*. However, with fewer than m symbols in the μ-pattern, it may be possible to determine, assuming some future behavior, whether the constraint will be satisfied. We call this evaluation an *eager* evaluation.

Given a constraint λ_w (let $m = \lambda_w.m$) and μ-pattern μ of a repeating task. If t is such that $p = |\mu^w(t)|$, where $p < m$, then, under an eager evaluation:

- The constraint, λ, is *satisfied* at t if, even assuming that the next $m - p$ deadlines are missed, λ will be satisfied:

$$\mu^w(t) \vdash \lambda_w \quad \stackrel{\text{def}}{\Leftrightarrow} \quad \mu^w(t) \cdot 0^{m-p} \vdash \lambda.$$

- The constraint, λ, is *unsatisfied* at t if, even assuming that the next $m - p$ deadlines are met, λ will be unsatisfied:

$$\mu^w(t) \nvdash \lambda_w \quad \stackrel{\text{def}}{\Leftrightarrow} \quad \mu^w(t) \cdot 1^{m-p} \nvdash \lambda.$$

- If it cannot be determined whether the constraint is satisfied or unsatisfied, then the constraint is *undefined* at t.

The previous concepts are illustrated in Fig. 2. It shows a pattern of the behavior of a repeating task that misses some of its invocations (the fourth, seventh, eighth, and 18th) with a $\binom{3}{4}$ constraint and with a window of nine time units. The graph shows the state of the constraint for both the lazy and eager evaluation modes.

The *lazy* bar shows the status of the constraint assuming that it is undefined unless there $\mu^w(t)$ has at least four invocations. At $t2$, four tasks are finished and it is known that the fourth invocation missed the deadline, however, as the previous three were met, then the constraint is satisfied. This holds until time $t3$, when $\mu^w(t3)$ does not hold four instances of the task. At $t4$, the task missed the deadline and the weakly hard constraint is not satisfied as two deadlines were missed. This holds until $t5$, where the constraint is again satisfied as $\mu^w(t5) = 01111$.

The *eager* bar is able to decide sooner whether the constraint is either satisfied or missed. For instance, at $t1$, as three deadlines were met, independently of the fourth invocation, $\binom{3}{4}$ is satisfied. In the same way, at $t4$, as two deadlines were missed in the last three, the constraint is not satisfied anyway.

However, updating $\mu^w(t)$ at finalization and invocation times of the task introduces considerable overhead. It would require time stamping each symbol of the μ-pattern and then determining which are the last symbols in w at t. As we seek intuitive and simple mechanisms for checking weakly hard constraints, we propose a relaxation of the satisfiability of the constraint. We assume that, after a task is started (i.e., after the first m invocations have finished), the last m symbols of the μ-pattern are the relevant history of the task. This assumption holds if the window size is larger than m times the average interarrival times of the task and the task does not have long periods of inactivity. If so, then mechanisms for determining whether the constraint is undefined should be used.

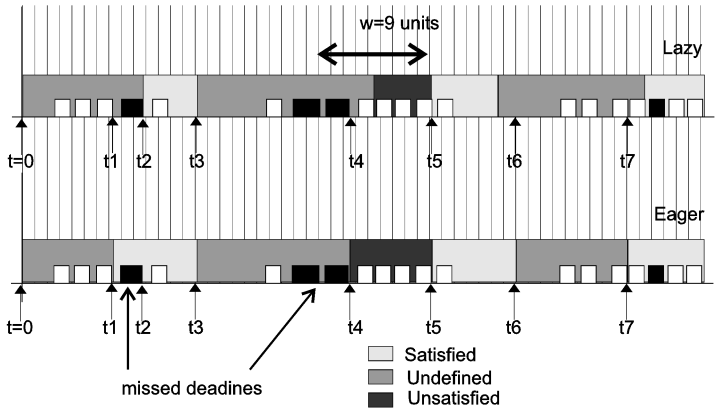

Fig. 2. State of constraint $\binom{3}{4}_{w=9}$ for a repeating task under lazy and eager evaluation.

3.5 An Algebra of Temporal Constraints

A natural question arises as to whether different types of constraints are comparable. For this purpose, we shall define the relation between sequences. As our sequences are binary, we can naturally extend to them the logical operators (i.e., applied symbol-wise). As a 1 in the μ-pattern represents a met deadline, we introduce the following definition to be able to compare μ-patterns.

Definition 8. *Given either two words u, v of the same length or two infinite sequences u, v (i.e., $u, v \in \Sigma^l$, $l > 0$ or $u, v \in \Sigma^\infty$), we say that u is harder than v (v is easier than u) if u has, at least, the same number of 1s and in the same positions as v. Formally,*

$$u \preceq v \quad \Leftrightarrow \quad u \wedge v = v. \tag{3}$$

For example, let $u = 11011$, $v = 10001$, and $w = 11100$, then $u \preceq v$ because $u \wedge v = 10001 = v$. However, nothing can be ascertained between u (or v) and w as $u \wedge w = 11000$.

It is easy to prove that the relation $\preceq$ is a partial order over sequences. Additionally, $(\Sigma^l, \preceq, \wedge, \vee)$ for all $l \geq 1$ is a bounded lattice. For any two words $u, v \in \Sigma^l$, their infimum is $u \wedge v$ and their supremum is $u \vee v$. The minimal element is the word 1^l and the maximal element is the word 0^l. The same applies to Σ^∞.

Given a constraint λ, it is also of interest to consider the set of sequences that satisfy it:

Definition 9 (Satisfaction set). *The satisfaction set of a constraint λ, denoted by $\mathcal{S}(\lambda)$, is the set of all $\alpha \in \Sigma^\infty$ that satisfy λ.*

For example, the set of μ-patterns of length 3 and 4 that satisfy the $\binom{2}{3}$ constraint are:

$$\mathcal{S}\left(\binom{2}{3}\right) = \{011, 101, 110, 111,\ 0110, 0111, 1011, \ldots\}.$$

With this definition, we can then introduce the comparison between constraints.

Definition 10. *Given two constraints, $\lambda, \lambda' \in \Gamma$, we say that λ is harder than λ' (λ' is easier than λ), denoted by $\lambda \preceq \lambda'$ if all sequences that satisfy λ also satisfy λ'. Formally,*

$$\lambda \preceq \lambda' \quad \Leftrightarrow \quad \mathcal{S}(\lambda) \subseteq \mathcal{S}(\lambda'). \tag{4}$$

For example, $\binom{2}{3} \preceq \binom{1}{3}$, for instance, $a = 011011 \vdash \binom{2}{3}$, and, therefore, $a \vdash \binom{1}{3}$.

The relation $\preceq$ induces naturally the notion of constraint equivalence as

$$\lambda \equiv \lambda' \quad \Leftrightarrow \quad \lambda \preceq \lambda' \ \wedge \ \lambda' \preceq \lambda. \tag{5}$$

It is easy to prove that $\equiv$ is an equivalence relationship. There are two classes of particular interest, namely, $[\binom{1}{1}]$ and $[\binom{0}{1}]$. These two classes are the *hardest* and the *weakest* classes of constraints, respectively. If a constraint $\lambda \in [\binom{1}{1}]$, then the only sequence that satisfies it is 1^*. On the other hand, if a constraint $\lambda \in [\binom{0}{1}]$, then any sequence, even 0^*, satisfies λ.

Now, we shall relate the relation between constraints and the relation between sequences. This is achieved by two monotonicity properties.

If a sequence α satisfies a constraint λ, any other sequence which is harder than α will also satisfy the constraint. Formally,

Theorem 1. *Given $\alpha \in \Sigma^\infty$ and $\lambda \in \Gamma$ such that $\alpha \vdash \lambda$, then $\forall \alpha' \in \Sigma^\infty$, $\alpha' \preceq \alpha \Rightarrow \alpha' \vdash \lambda$.*

For example, let $a = 10101010$, which $a \vdash \binom{2}{4}$, and let $b = 10111110 \preceq a$, then it follows that b also satisfies $\binom{2}{4}$.

Likewise, if a constraint λ is satisfied by a sequence α, any other constraint which is weaker than λ will also be satisfied by that sequence. Formally,

Theorem 2. *Given $\lambda \in \Gamma$ and $\alpha \in \Sigma^\infty$ such that $\alpha \vdash \lambda$, then $\forall \lambda' \in \Gamma$, $\lambda \preceq \lambda' \Rightarrow \alpha \vdash \lambda'$.*

For example, let $a = 10101010$ which satisfies $\binom{2}{4}$. As $\binom{2}{4}$ is harder than $\binom{1}{4}$, it follows that $a \vdash \binom{1}{4}$.

The proofs of these theorems are straightforward. Another interesting property is "fragility," which is a measure of how close a particular sequence is to missing its weakly hard constraint.

Definition 11. *A sequence $\alpha \in \Sigma^\infty$ is p-fragile for a constraint λ if $\alpha \vdash \lambda$ and p is the smallest number that satisfies that $\exists \alpha'$, obtained from α by switching p 1s in α to 0 so that α' no longer satisfies λ.*

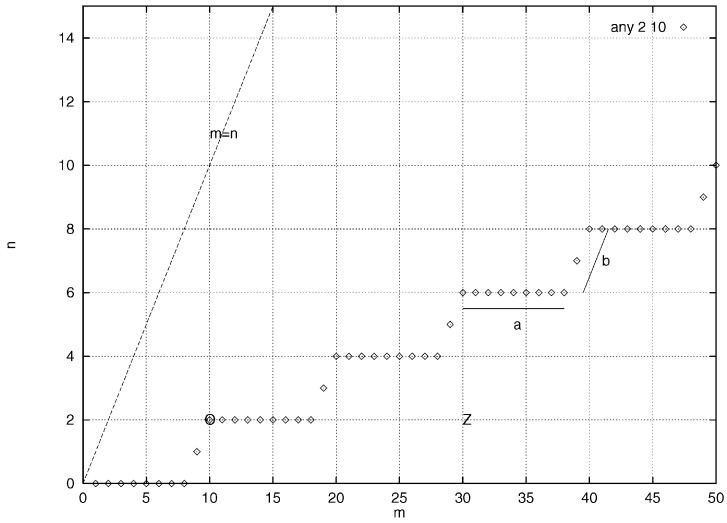

Fig. 3. Graphical interpretation of the $\binom{n}{m} \preceq \binom{p}{q}$ relationship for $n = 2$, $m = 10$. All pairs (p, q) in region Z satisfy the constraint $\binom{2}{10} \preceq \binom{p}{q}$.

In particular, a 1-fragile sequence will simply be called a fragile sequence. For example, $a = 10111$ is 3-fragile for $\binom{1}{4}$ because if we switch any two symbols, for instance, the first and third symbols of a, we have $a' = \mathbf{0}0\mathbf{0}11$, which still satisfies $\binom{1}{4}$. However, a' is fragile because if we switch one symbol more, we have a sequence that is no longer satisfied.

3.6 Constraint Relationship Properties

After having introduced weakly hard constraints and having analyzed their properties, the next step is obtaining expressions that directly relate two particular constraints as being equivalent, harder, or weaker. In this section, we present some expressions that allow us to obtain the relationship between two constraints as a function of their n and m parameters. The proofs of these theorems can be found in [20], [6]. We first start with some equivalences between constraints.

Theorem 3. $\binom{n}{m} \equiv \overline{\binom{m-n}{m}}$.

Meeting n deadlines out of m, in any order, is equivalent to specifying that no more than $m - n$ deadlines can be missed out of m.

Theorem 4. $\overline{\langle \begin{smallmatrix} n \\ m \end{smallmatrix} \rangle} \equiv \overline{\langle \begin{smallmatrix} n \\ m' \end{smallmatrix} \rangle} \quad \forall m, m' \geq n.$

The theorem states that specifying that no more than n deadlines in a row can be missed is independent of the size of the window m. We now analyze relationships between constraints of the same type.

Theorem 5. $\binom{n}{m} \preceq \binom{p}{q} \Leftrightarrow \quad p \leq \max\left\{ \left\lfloor \frac{q}{m} \right\rfloor n, q + \left\lceil \frac{q}{m} \right\rceil (n - m) \right\}.$

The proof is based on building, for a given pair m, n, the weakest sequence that satisfies the constraint and then deriving the parameters p and q of the constraints which also satisfy this sequence. Fig. 3 shows the points (p, q) that satisfy the inequality for the pair $(n, m) = (2, 10)$. The diamonds represent the boundary of the region of points (p, q) that are weaker than $\binom{2}{10}$. For instance, $\binom{2}{10} \preceq \binom{4}{30}$. Each section a, b corresponds to the parts of the max function.

The two following corollaries give us insight on constraint properties:

Corollary 1. $\binom{n}{m} \preceq \binom{p}{m} \Leftrightarrow p \leq n.$

If two constraints have the same window size m, the higher the number of deadlines to be met n, the harder the constraint.

Corollary 2. $\binom{n}{m} \preceq \binom{n}{q} \Leftrightarrow q \geq m.$

If two constraints require the same number of deadlines to be met, the wider the window, the weaker the constraint.

Theorem 6. $\langle \begin{smallmatrix} n \\ m \end{smallmatrix} \rangle \in [\langle \begin{smallmatrix} 1 \\ 1 \end{smallmatrix} \rangle] \Leftrightarrow 2n > m.$

The previous theorem states that specifying that more than $\frac{m}{2}$ deadlines have to be met in a row out of m is equivalent as specifying that *all* deadlines have to be met. This can be seen clearly with an example. Take, for instance, $\langle \begin{smallmatrix} 3 \\ 5 \end{smallmatrix} \rangle$ and consider the subsequence $\alpha = 11011$ of $1 \ldots 11011 \ldots 1$, which has only one zero. α does not have three ones in a row and, therefore, $\langle \begin{smallmatrix} 3 \\ 5 \end{smallmatrix} \rangle$ is not satisfied. The next theorems show the relationship between *row* constraints which are not strongly hard.

For

$$\left\langle \begin{matrix} n \\ m \end{matrix} \right\rangle, \left\langle \begin{matrix} p \\ q \end{matrix} \right\rangle \notin \left[\left\langle \begin{matrix} 1 \\ 1 \end{matrix} \right\rangle \right], \quad \left\langle \begin{matrix} n \\ m \end{matrix} \right\rangle \preceq \left\langle \begin{matrix} p \\ q \end{matrix} \right\rangle \Leftrightarrow$$
$$(q < m \ \wedge \ p \leq n - \left\lceil \frac{m - q}{2} \right\rceil) \ \vee \ (q \geq m \ \wedge \ p \leq n).$$

The proof is also based on building, for a given pair m, n, the weakest sequence that satisfies the constraint. The graphical interpretation of the theorem is shown in Fig. 4. All the pairs (p, q) in Z satisfy the constraint.

Note that not all pairs are comparable. There are constraints λ, λ' for which neither $\lambda \preceq \lambda'$ nor $\lambda' \preceq \lambda$. Also, Corollaries 1 and 2 have their corresponding corollaries for the "row" constraint:

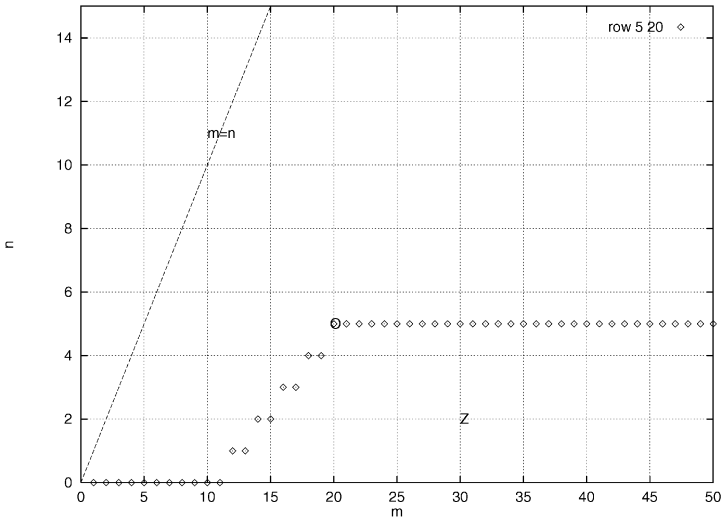

Fig. 4. Graphical interpretation of the $\langle {}^{n}_{m} \rangle \preceq \langle {}^{p}_{q} \rangle$ relationship. All pairs (p, q) in region Z satisfy $\langle {}^{5}_{20} \rangle \preceq \langle {}^{p}_{q}$.

Corollary 3. *For* $\langle {}^{n}_{m} \rangle, \langle {}^{p}_{m} \rangle \notin [\langle {}^{1}_{1} \rangle]$ $\quad \langle {}^{n}_{m} \rangle \preceq \langle {}^{p}_{m} \rangle \Leftrightarrow p \leq n.$

Corollary 4. *For* $\langle {}^{n}_{m} \rangle, \langle {}^{n}_{q} \rangle \notin [\langle {}^{1}_{1} \rangle]$ $\quad \langle {}^{n}_{m} \rangle \preceq \langle {}^{n}_{q} \rangle \Leftrightarrow q \geq m.$

The properties between *row* and *any* constraints are harder. The following theorem presents a lower bound on the constraint parameters:

Theorem 8. $\langle {}^{n}_{m} \rangle \preceq \binom{p}{m} \Rightarrow \quad p \leq 4n - m - 2,\ p \leq m,\ p \geq 0.$

Finally, a similar property as of Corollary 1 can be obtained for the *miss row* constraint:

Theorem 9. $\overline{\langle n \rangle} \preceq \overline{\langle p \rangle} \Leftrightarrow n \leq p.$

3.7 Closed Sequences

We have considered the satisfiability of weakly hard constraints in terms of infinite sequences of task invocations. However, for most systems, the μ-pattern is made up of the cyclic repetition of a word. In such a case, we say that the μ-pattern is closed. Formally,

Definition 12 (closed sequence). *Given a sequence* $\alpha \in \Sigma^{\infty}$, α *is closed with cycle* c *if:*

$$\forall k, r \ \ k \geq 1, r > 0, \ \ \alpha(k) = \alpha(k + rc). \tag{6}$$

For closed sequences, the check of whether $\alpha \vdash \lambda$ can be done by checking only the slices of size m starting within a single cycle (plus the size of the window), as the next theorem shows.

Theorem 10. *If a sequence* $\alpha \in \Sigma^{\infty}$ *is closed, then, for all constraints with window size* m*,*

$$\alpha \vdash \lambda \ \ \Leftrightarrow \ \ \forall \omega \in W^{m}(\alpha(1..c + m - 1)), \ \ \omega \vdash \lambda.$$

For closed sequences, the check of whether $\alpha \vdash \lambda$ can be computed by elementary algorithms whose cost is linear with respect to the length of the slice. Closed sequences are of special interest, as we shall see in the next sections. In most practical environments, periodic task μ-pattern are closed.

3.8 Using Weakly Hard Constraints

The $\preceq$ relationship is also a partial order over weakly hard constraints. It is therefore meaningful to set different weakly hard constraints on the same task. In fact, we can define an algebra of weakly hard constraint combining them with Boolean operators. For example, we can specify that a given task has to satisfy $(\binom{5}{10}$ and $\overline{\langle 3 \rangle})$ or $\binom{8}{12}$. However, the formal treatment is left for a further work.

Returning to the example in Section 3.1, a careful analysis shows that task τ_9 satisfies, for instance, the constraints: $\binom{9}{10}$, $\overline{\langle 1 \rangle}$, $\langle {}^{19}_{100} \rangle$, but it does not satisfy $\binom{29}{30}$ nor $\binom{20}{100}$. It is straightforward, using the theorems presented before, to determine the weakest set of constraints that are satisfied for a given task μ-pattern.

There are two different contexts in which weakly hard constraints are used. Here, we have concentrated our efforts on *off-line* analysis of weakly hard constraints (i.e., given the μ-pattern of the worst-case execution of a task with a particular scheduling algorithm, determining whether it satisfies its weakly hard constraints). However, the same framework allows us to define new scheduling algorithms that specifically address the problem of satisfying *on-line weakly hard constraints*. This can be done either in a best-effort or in a guaranteed way. For instance, a similar notion of p-fragility can be used as the priority of the task to determine how close a given task is from missing its weakly hard constraint. This is the subject of ongoing research.

4 Fixed Priority Schedulability Analysis

After having presented weakly hard constraints and studied their properties and relationships, we now obtain off-line schedulability checks for some process models under fixed priority basis.

A real-time system can be implemented in several ways, no matter whether its specification is strongly or weakly hard. However, the techniques for analyzing whether a specification is met are different.

In this section, we will develop only the implementation consisting of mapping the specification jobs into execution threads. Moreover, we will restrict our analysis to fixed priority scheduling of such threads or tasks. The purpose of doing so is to provide insight about how weakly hard specifications can be usefully related to a particular implementation before extending the specification discussion. We therefore present response time-based schedulability checks for tasks sets scheduled under fixed priority.

Of particular relevance is the assignment of priorities to the tasks. We use P_i as the priority of the task, 1 being the highest priority.

We shall also have, for each task τ_i, the worst case computation time corresponding to a single invocation of the task, denoted by C_i. Moreover, the tasks may communicate data through common variables. If so, we must also take into account the blocking factors due to blocking operations upon these variables under, for instance, the priority ceiling protocol [21]. The maximum blocking time a task may suffer from lower priority tasks is denoted by B_i. With these definitions, each task is therefore characterized by:

$$\tau_i = (T_i, D_i, C_i, B_i, O_i, P_i, \lambda_i).$$

The utilization of the task set is given by: $U = \sum_{\tau_i \in \Pi} \frac{C_i}{T_i}$.

4.1 Analysis of Synchronous Systems

We first present the analysis of systems made of periodic tasks only scheduled with a fixed priority scheduler. This analysis requires two steps. The first one is computing the upper bounds for the response times of each invocation of the tasks. The second one is checking the resulting such μ-pattern satisfies the constraint.

We extend the notion of closed sequences to tasks. We say that a task is closed if its μ-pattern is closed.

In order to check whether the tasks of a system satisfy their weakly hard specifications, we must start computing their μ-pattern. Fortunately, if the utilization is not larger than one, these μ-patterns are closed, so their analysis can be reduced to a finite sequence. However, we shall first prove this property.

Theorem 11. *Given a task set* Π*, made up of independent periodic tasks scheduled under fixed priority, then all tasks are closed if and only if* $U \leq 1$*. Moreover, the cycle of each task* τ_i *is given by* $c_i = a_i$ *invocations.*

Proof. Assume $U < 1$, then we can extend the system by an additional task τ_{N+1} at the lowest priority having $T_{N+1} = \mathbf{H}$ and $C_{N+1} = \mathbf{H}(1-U)$. Note that C_{N+1} is integer because $\mathbf{H}$ divides all periods T_i.

Such a task makes the new system have a utilization rate $U' = 1$. C_{N+1} is exactly the amount of time that the original system left unused in the first hyperperiod. The response time of the only invocation of τ_{N+1} within $\mathbf{H}$ can be computed by the usual expression [18].

$$R_{N+1} = C_{N+1} + \sum_{\tau_j \in A} \left\lceil \frac{R_{N+1}}{T_j} \right\rceil C_j.$$

$\mathbf{H}$ is a solution of the previous equation because

$$C_{N+1} + \sum_{\tau_j \in A} \left\lceil \frac{\mathbf{H}}{T_j} \right\rceil C_j = \frac{\mathbf{H}}{T_{N+1}} C_{N+1} + \sum_{\tau_j \in A} \frac{\mathbf{H}}{T_j} C_j = \mathbf{H}U' = \mathbf{H}.$$

This means that the lowest priority task finishes its computation exactly by the hyperperiod, therefore, the last time unit within the hyperperiod is used by τ_{N+1}. So, all tasks from the original set Π have completed all their invocations within the hyperperiod.

In that case, the next hyperperiod starts with the same conditions as the first one. Therefore, the upper bounds on the response times of the tasks at any invocation have cyclic pattern of period $\mathbf{H}$.

The same argument can be also applied in the case of $U = 1$. In this case, we don't need to add an additional task. At the hyperperiod, all tasks have finished all their invocations within the hyperperiod.

The same argument also applies to the subset of tasks of higher priority than task τ_i. In this case, the cycle of task τ_i is repeated every h_i time units and, therefore, the task is closed with cycle $c_i = \frac{h_i}{T_i} = a_i$. □

Under the operation of a protocol for shared resources, the same result applies as, in the worst case, the task is blocked once at each invocation.

The computation of the μ-pattern for task τ_i requires computing the worst case finalization time of task τ_i at invocation k, $F_i(k)$, within the first hyperperiod at level i. $F_i(k)$ is the sum of three factors. First, the time of computation of the first k invocations of the task. Second, the time that higher priority tasks have been computing before $F_i(k)$. Third, the time the processor has not been used by tasks with priority higher than or equal to τ_i. This is due to the fact that the invocations of these tasks can't start before their invocation times. So, they can leave the processor idle. Therefore, $F_i(k)$ is the minimum $t > 0$ that makes the following equation hold:

$$t = kC_i + \mathrm{Intf}_i(t) + \mathrm{Idle}_i(S_i(k)), \tag{7}$$

where $\mathrm{Intf}_i(t)$ is the interference from higher priority tasks up to time t. The idle time at level i, $\mathrm{Idle}_i(t)$, is the amount of time the processor can be used by tasks of lower priority than τ_i within a period of time $[0, t)$.

The value $F_i(k)$ can be computed with a generalization [6] of the worst case response time formulation. Formally,

$$\begin{cases} t_0 & = 0 \\ t_{n+1} & = kC_i + \mathrm{Intf}_i(t_n) + \mathrm{Idle}_i(S_i(k)). \end{cases} \tag{8}$$

This equation does not have a direct solution for t. But, it can be computed with a recurrence formula. Starting with $t_0 = 0$, the recurrence finishes when $t_{n+1} = t_n$. The recurrence finishes if $U \leq 1$. The interference function $\mathrm{Intf}_i(t)$ is the amount of time higher priority tasks have requested before t and it can be computed by:

$$\mathrm{Intf}_i(t) = \sum_{\tau_j \in \mathrm{hp}(\tau_i)} \left\lceil \frac{t - O_j}{T_j} \right\rceil C_j, \tag{9}$$

where $\mathrm{hp}(\tau_i)$ is the set of tasks of higher priority than task τ_i. $\mathrm{Idle}_i(t)$ can be computed by adding a virtual task $\bar{\tau} = \left(\bar{T} = t, \bar{D} = t, \bar{C}, \bar{O} = 0\right)$ of lower priority than τ_i and

computing the maximum value $\bar{C}$ such that task $\bar{\tau}$ meets its deadline. Formally,

$$\mathrm{Idle}_i(t) = \max\{\bar{C} \mid \bar{\tau} \text{ is schedulable}\}. \quad (10)$$

The finish time for task $\bar{\tau}$ is given by:

$$t = \bar{C} + \sum_{\tau_j \in \mathrm{hep}(\tau_i)} \left\lceil \frac{t - O_j}{T_j} \right\rceil C_j \quad (11)$$

and checking if $t < \bar{D}$. Here, we use $\mathrm{hep}(\tau_i)$, the set of tasks of higher or equal priority than task τ_i, as we need to also account for the interference of task τ_i itself. Equation (11) can be computed by a recurrence on the values of t. The range of values to check for $\bar{C}$ range from 0 to t and can be computed by a bisection search. See [6] for effective upper and lower bounds.

Note that, in (7), $\mathrm{Idle}_i(S_i(k))$ does not depend on t and therefore does not need to be recomputed at each iteration of the recurrence. $R_i(k)$ can also be computed by explicitly building the schedule (for instance, with an event driven simulator), but, as it is shown in [6], the formulation presented here is much more efficient. In addition to that, there are two major problems associated with the analysis of the constraints based on building the schedule:

- The schedule can only be built for perfectly synchronous systems, that is, systems without blocking factors and without sporadic (repeating or isolated) tasks. This restriction limits the applicability of the analysis to a small range of task models.
- The schedule has to be built for all the tasks independently of whether they meet all of their deadlines.

Once all the $R_i(k)$ values have been calculated, the resulting μ-patterns can be checked.

One common argument against analyzing the system up to the hyperperiod is that, in general, the hyperperiod can be extremely large. In theory, this is the case, but we note that, in real engineering scenarios, this pessimism can be overridden by noting that the periods of the tasks are usually adapted to achieve low hyperperiods and, also, they can be generally modified to remove common prime factors.

The other drawback is that the computation of the analysis can take considerable time. However, this is an off-line analysis technique and, therefore, analysis time is not a major concern. Also note that, for instance, the computation cost of performing the analysis is, in general, smaller than the time it takes to compile the whole system.

4.2 Optimal Priority Assignment

One interesting result is that, for task sets with weakly hard constraints, neither Deadline Monotonic nor Rate Monotonic priority assignment are optimal.

This is very easy to see with an example. Assume a task set made of two tasks, as shown in Table 3 and assume the priorities are assigned in deadline monotonic priority ordering. The first task meets all its deadlines and therefore satisfies its weakly hard constraint. However, the second one misses its deadline every time it is invoked because it overlaps with the execution of τ_1. As a consequence, with this priority ordering the task set is not weakly hard schedulable. However, if we swap the priorities of the tasks, it turns out that τ_2 meets all its deadlines and therefore satisfying its constraint. Task τ_1 will miss two deadlines in a row in every 10 invocations. This is due to the fact that, when it coincides with τ_2, it suffers a burst of interference of 18 ticks. As it only misses two deadlines every 10, its constraint is clearly satisfied.

TABLE 3
Nonoptimality of DMA Priority Assignment

τ_i	T_i	D_i	C_i	$\binom{n}{m}$
τ_1	10	10	5	$\binom{5}{10}$
τ_2	100	20	18	$\binom{9}{10}$

Fortunately, an optimal priority assignment can be found by applying the partition method. It is based on the property that if a task is weakly hard schedulable at a priority p, then it is also weakly hard schedulable at any higher priority. The algorithms starts with an initial priority assignment (Deadline Monotonic is a good choice) and checks whether the lowest priority task meets its constraint. If if fails, then another task is selected to be the lowest priority task. If no task is found to be schedulable at that priority level, then the task set is unschedulable. If a task is found, then this task is left at that priority and the set of remaining tasks is considered. This process finishes when all tasks are found schedulable or when no task can be found to be schedulable at a given priority level.

4.3 Analysis of Asynchronous Systems

The previous analysis can be easily extended to schedules in which tasks may suffer blocking due to the operation of a protocol for accessing resources (like ceiling protocol) and nonperiodic tasks. The worst case response time computation can be extended to include the maximum possible blocking time at any particular invocation.

Also, the response time $R_i(k)$ for periodic tasks can account for the worst case interference from nonperiodic tasks. The formulation is updated to consider the worst possible interference from nonperiodic tasks at each periodic task invocation. It is still an open issue whether off-line guarantees can be obtained for repeating tasks.

Note that most of the tasks will meet all of their deadlines and, therefore, weakly hard checks would have to be performed on a small subset of the tasks only. If a task is predicted to meet is first deadline (at the critical instant) then it will meet all deadlines.

5 Conclusion

The new generation of real-time systems requires mechanisms to specify in a clear, predictable, and bounded way that some deadlines can be missed. This is mainly for 1) alleviating the pessimism in the parameters of the system (mainly C_i) and the worst case scenarios (assuming all sporadics arriving simultaneously), 2) providing a mechanism for fair degradation of the quality of the service tasks

provide, and 3) obtaining a fair mechanism for deciding which tasks need to be skipped during transient overloads.

This allows systems to be built with worst case utilization close to 100 percent. This means that, generally, all tasks meet their deadlines, but, even in the worst case, a minimum level of service is given to the tasks.

The other main property of such systems is that it enables one to predict what the behavior of a task will be in the case that a deadline is missed. Current scheduling algorithms do not remember which tasks missed the deadline and how many times. This leads to unfairness between the tasks and to nonguaranteed behavior. Weakly hard real-time systems are based on: 1) clear and intuitive constraints for specifying the number of deadlines missed and met over a period of time, 2) study of the properties of the tasks specified with weakly hard constraints, and 3) offline schedulability checks for determining whether those constraints will be satisfied under traditional scheduling algorithms.

ACKNOWLEDGMENTS

This research has been partially funded by the EPSRC project DIRC.

REFERENCES

[1] S. Bennett, *Real-Time Computer Control: An Introduction.* Prentice Hall Int'l, 1994.
[2] C. Houpis, *Digital Control Systems. Theory, Hardware and Software.* McGraw-Hill, 1985.
[3] C.C. Bissell, *Control Engineering,* second ed. Chapman & Hall, 1998.
[4] M. Törngren, "Fundamentals of Implementing Real-Time Control Applications in Distributed Computer Systems," *Real-Time Systems,* vol. 14, pp. 219-250, 1998.
[5] G. Bernat and A. Burns, "Combining (n m)-Hard Deadlines with Dual Priority Scheduling," *Proc. 18th IEEE Real-Time Systems Symp.,* Dec. 1997.
[6] G. Bernat, "Specification and Analysis of Weakly Hard Real-Time Systems," PhD thesis, Departament de Ciències Matemàtiques i Informàtica. Universitat de les Illes Balears. Spain, http://www.cs.york.ac.uk/~bernat, Jan. 1998.
[7] M. Hamdaoui and P. Ramanathan, "A Dynamic Priority Assignment Technique for Streams with (m,k)-Firm Deadlines," *IEEE Trans. Computers,* vol. 44, no. 12, pp. 1443-1451, Dec. 1995.
[8] R. West and C. Poellabauer, "Analysis of a Window-Constrained Scheduler for Real-Time and Best-Effort Packet Streams," *Proc. 21st IEEE Real-Time Systems Symp.,* Nov. 2000.
[9] G. Koren and D. Shasha, "D-Over: An Optimal On-Line Scheduling Algorithm for Overloaded Real-Time Systems," *Proc. 13th IEEE Real-Time Systems Symp.,* pp. 290-300, Dec. 1992.
[10] G. Koren and D. Shasha, "Skip-Over: Algorithms and Complexity for Overloaded Systems that Allow Skips," *Proc. 16th IEEE Real-Time Systems Symp.,* pp. 110-117, Dec. 1995.
[11] M. Caccamo and G. Buttazzo, "Exploiting Skips in Periodic Tasks for Enhancing Aperiodic Responsiveness," *Proc. 17th IEEE Real-Time Systems Symp.,* pp. 330-339, Dec. 1997.
[12] J.A. Stankovic, M. Spuri, K. Ramamritham, and G. Buttazzo, *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms.* Kluwer Academic, 1998.
[13] G. Buttazzo, M. Spuri, and F. Sensini, "Value vs. Deadline Scheduling in Overload Conditions," *Proc. 16th IEEE Real-Time Systems Symp.,* pp. 90-99, Dec. 1995.
[14] M. Spuri, G. Buttazzo, and F. Sensini, "Robust Aperiodic Scheduling under Dynamic Priority Systems," *Proc. 16th IEEE Real-Time Systems Symp.* pp. 210-219, Dec. 1995.
[15] H. Wedde and J. Lind, "Building Large, Complex, Distributed Safety-Critical Operating Systems," *Real-Time Systems J.,* vol. 13, pp. 277-302, 1997.
[16] L. Abeni and G. Buttazzo, "Integrating Multimedia Applications in Hardware Real-Time Systems," *Proc. 19th IEEE Real-Time Systems Symp.,* pp. 4-14, Dec. 1998.
[17] D. Locke, D. Vogel, and T. Mesler, "Building a Predictable Avionics Platform in Ada: A Case Study," *Proc. 12th Real-Time Systems Symp.,* pp. 181-189, Dec. 1991.
[18] N. Audsley, A. Burns, M. Richardson, and A.J. Wellings, "Applying New Scheduling Theory to Static Priority Pre-Emptive Scheduling," *Software Eng. J.,* vol. 8, no. 5, pp. 284-292, 1993.
[19] A. Burns and A. Wellings, *Real-Time Systems and Their Programming Languages,* second ed. Addison Wesley, 1996.
[20] G. Bernat and A. Burns, "Weakly-Hard Temporal Constraints," Technical Report YCS, Dept. of Computer Science. Univ. of York, 2000.
[21] L. Sha, R. Rajkumar, and J.P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Trans. Computers,* vol. 39, no. 9, pp. 1175-1185, Sept. 1990.

Guillem Bernat graduated in 1992 in computer science from the Universitat de les Illes Balears. He later got his PhD degree from the same university in 1998. He was a lecturer in its Department of Computer Science from 1992 until 2000, when he was appointed a senior research fellow in the Department of Computer Science at the University of York. Since January 2001 he has been a lecturer in the same department. He is interested in a wide variety of aspects of real-time systems, mostly on scheduling, flexible scheduling architectures, and portable worst-case execution time analysis. He is a member of the IEEE.

Alan Burns graduated in 1974 in mathematics from Sheffield University, then took a DPhil degree in the Computer Science Department at the University of York. He has worked for many years on a number of different aspects of real-time systems engineering. After a short period of employment at UKAEA Research Centre, Harwell, he was appointed to a lectureship at Bradford University in 1979. He was subsequently promoted to senior lecturer in 1986. In January 1990, he took up a readership at the University of York in the Computer Science Department. During 1994, he was promoted to a personal chair. In 1999, he became head of the Computer Science Department at York. His research activities have covered a number of aspects of real-time and safety critical systems, including requirements for such systems, the specification of safety and timings needs, systems architectures appropriate for the design process, the assessment of languages for use in the real-time safety critical domain, distributed operating systems, the formal specification of scheduling algorithms and implementation strategies, and the design of dependable user interfaces to safety critical applications.

Professor Burns, together with Professor Wellings, heads the Real-Time Systems Research Group at the University of York—this is one of the largest research groups in this area in the world and has a strong international reputation. He has authored/coauthored more than 300 papers/reports and eight books. Most of these are in the Ada or real-time area. His teaching activities include courses in operating systems, scheduling, and real-time systems. He is a senior member of the IEEE.

Albert Llamosí graduated in history from the Universitat Autònoma de Barcelona. He received industrial engineer and doctor in computer science degrees from the Universitat Politècnica de Catalunya. He has taught different programming-related topics, including compiler design, at the Universitat Politècnica de Catalunya, at the Universitat Rovira i Virgili, and at the Universitat de les Illes Balears, where he has been a full professor since 1991. His research domain has always been parallel programming and real-time systems. He has been a member of the Ada-Europe board for several years.