A Framework for Composing Real-Time Schedulers

Giuseppe Lipari and Enrico Bini
Scuola Superiore S. Anna
Pisa, Italy
{*lipari,e.bini*}*@sssup.it*

Gerhard Fohler
Mälardalen University
Västerås, Sweden
gfr@mdh.se

January 8, 2003

Abstract

In this paper, we present a framework for integrating real-time components in the same system, where each component can have its own scheduling algorithm. There are two main reasons for this research: allowing maximum flexibility in the design of systems with many different real-time activities; reusing already existing applications without changing their scheduling policy. After defining the concept of component in our context, we present our methodology that is based on a two-level hierarchical scheduling paradigm: at the global level, a scheduler selects which component must be executed at each instant; the selected component then chooses which task has to be scheduled depending on its own scheduling strategy.

1 Introduction

In software engineering, a software component is described by the services it requires and the services it provides to the other components in the system. For example, in object oriented programming, an object is characterized by its programming interface whereas the required services are often specified at the design level with some specification language.

Component-based design and development techniques are now being applied in real-time embedded systems. However, such techniques must be adapted to the particular needs of this domain. Until now, little work has been done on the characterization of the quality of service of a component from a temporal point of view. This would be especially useful in the real-time domain, where components consist of concurrent cyclic tasks with temporal constraints (i.e. deadlines). In fact, when we integrate all the components in the final system, we must be able to analyse the schedulability of the system (i.e. to check if the deadlines are respected).

The schedulability analysis depends on the adopted scheduling algorithm. There are many real-time scheduling algorithms: in practice, fixed priority scheduling is the most widely adopted paradigm, because it is provided by all real-time operating systems.

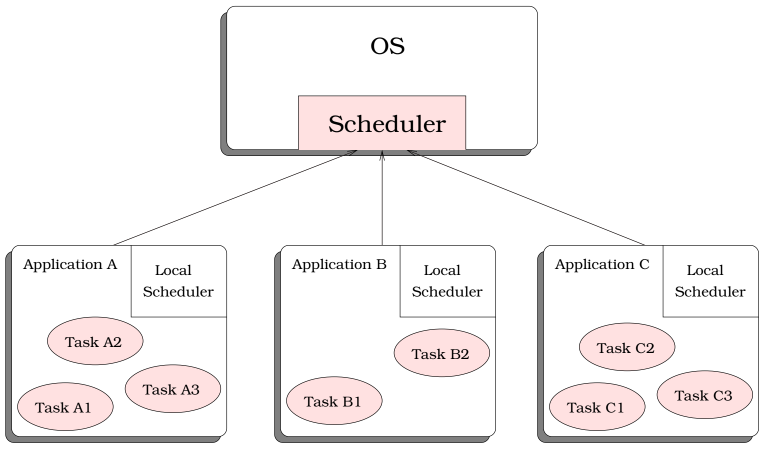

Figure 1: A hierarchical scheduler.

In some specific domain, like higly critical hard real-time systems, the Time Triggered paradigm is used instead. The Earliest Deadline First algorithm is used when we want to maximize the resource utilization. In general, there is not a catch-all: a scheduling algorithm is best suited in some case and is less suited in others.

To allow maximum flexibility and separation of concerns, each components should be designed assuming its own scheduling algorithm: during the integration phase, it would then be necessary to put togheter all the components, each one with its own scheduling algorithm. The structure of such hierarchical system is shown in Figure 1.

As an example, consider an engine control system in the automotive domain. In a standard controller, we can distinguish two sets of activities:

- Some activity is triggered by the *crank angle sensor*. The time between two activations is variable and depends on the rotations per minute of the engine. Therefore, the activity is triggered by an interrupt at a variable rate. Probably, the best way to schedule such an activity is by a fixed priority scheduler.

- Many other activities in the same system are periodic, and must be activated by a timer. For example, most of these systems are conneted to other electronic boards in the car through a Time Triggered network, which requires precise timing. Therefore, this second set of activities are best handled by a time triggered paradigm.

If we want to use a unique scheduling paradigm for the whole system we must either reduce the two set of activities to periodic tasks, and force them to be scheduled by a time triggered paradigm; or re-program the second set of activities to be handled by a fixed priority scheduler. In both case we have to do some effort in the programming; moreover, we will probably loose something in term of resource usage. The best choice would be to program the two set of activities as two distinct components, each one with its own scheduler, and then integrate the two components in the same system.

A similar problem arises when dealing with real-time systems with mixed constraints. In many systems, some activity may be critical and must be treated as a *hard real-time* (i.e. no deadline must be missed); some other activity is less critical and nothing catastrophic happens if some constraint is not respected. However, the quality of service provided is inversely proportional to the number of missed deadlines (*soft real-time* tasks). Different scheduling paradigm are used for hard and soft real-time activities: one way of composing such activities would be to implement them as different components, each one with its own scheduling algorithm.

Another motivation for composing schedulers is the possibility to re-use already existing components. Suppose we have a component that consists of many concurrent real-time tasks, which has been developed assuming a fixed priority scheduler. Now, we want to re-use the component in a new system in which we want to use the Earliest Deadline First algorithm, but we cannot go back to the design phase (for example for cost reasons). Therefore, we need a way of using the same component in the new system *without changing its scheduling algorithm.*

In this paper, we present a framework for integrating real-time components, each one with its own scheduling algorithm, guaranteeing the correctness of the system from a schedulability point of view. In this paper we will not consider problems related to the functional properties of the components, but only to their temporal properties.

After defining the concept of component in our context, we will propose a two level hierarchical scheduling strategy. At the global level, a system-level scheduler select which component has to be executed at each instant. The selected component then chooses which task has to be executed depending on its own scheduling algorithm.

2 Definition of component

A component is a piece of software consisting of:

- a set of one or more concurrent tasks (i.e. flows of execution);
- a scheduler (i.e. an algorithm that chooses which task must be executed at every instant);
- a set of output data ports;
- a set of input data ports.

2.1 Tasks

A task τ_i is a sequential program that is activated multiple times by external or internal events. At each activation (also called *job*) a piece of code is executed, and at the end the task is blocked waiting for the next activation. A task τ_i is characterized by a worst-case execution time C_i and a minimum interarrival time T_i between two consecutive activation. A task is *periodic* if it is activated exactly every T_i. Since we are interested in real-time applications, every job is assigned a deadline d_i. In most cases, this deadline is equal to the activation time r_i plus a relative deadline D_i. Summarizing, in the following a task τ_i is described by a t-uple (C_i, T_i, D_i).

Components that do not have a proper task (i.e. do not posses an independent flow of execution) are called "passive components". Passive components are data structures that can be protected by mutex semaphores. In this work, we will not consider passive components.

2.2 Communication

A component provides services to the other components and requires services from the external world. There are several methods for communicating between components, and they can be classified according to their synchronization paradigm. We can have synchronous services (i.e. a component invokes a service and waits for the response), asynchronous services (i.e. a component invokes a service and continues executing without waiting), loosely synchronous services, etc. In this paper, we will only consider a particular communication paradigm that completely de-couples the temporal behaviour of one component from the others.

A component $\mathcal{C}_i$ provides *output data ports* O_{ik} to the other components, and reads data from its *input data ports* I_{ik}. During integration, input and output data ports are connected to cyclic asynchronous buffer buffers (CABs) [4]. A CAB is a data structure that can be read and written at any instant and that always maintains the last written value. Read and write operation are never blocking. Such data structure can be implemented with a multiple buffers structure so that two concurrent tasks never access the same buffer at the same time.

We choose this communication paradigm because it completely de-couples the temporal behaviour of one component from the others, and greatly simplifies our integration analysis. As a future work, we plan to introduce other communication paradigms, like for example asynchronous procedure calls.

In a real-time system it is important to compute the maximum end-to-end delay that it takes to the system to elaborate one output, from the time the data is read from the external world (sensors), to the time that it is delivered to the external world (actuators). To do this computation, we must be able to characterize the flow of data from a temporal point of view.

For example, if component $\mathcal{C}_1$ requires fresh sensor data every 10msec from component $\mathcal{C}_2$, but the latter produces the data on the output port only every 20msec, there is a clear mismatch between the two components. To be able to check these mismatches, we have to introduce temporal constraints associated to the required data and temporal profiles associated to the provided data.

We propose here three additional parameters to associate to each provided data port O_{ik} of component $\mathcal{C}_i$:

- the maximum delay from the input δ_{ik}; this is the maximum time for the component to take the original data from the input (for example, a sensor or another component), elaborate and write the result in the output data port;

- the period π_{ik}; this is a average measure between two consecutive output instants in the data port;
- the maximum jitter ϕ_{ik}; this is the maximum amount of deviation (in positive or in negative) from the expected delivery.

Bound for these parameters can be obtained from the component tasks by analyzing their worst case temporal behavior. We do not report here the methodology for computing these bounds as it depends also on the scheduling algorithm.

Conversely, each input data port I_{ik} has only one requirements: the maximum allowed delay Δ_{ik} from the source, that is the maximum time it takes the data from the original source (i.e. a sensor) to the component reading.

We will use such parameters during the integration phase to check that the composition respects all the temporal constraints.

2.3 Temporal profile of a component

To be able to analise the whole system during the integration phase, we need to summarize the temporal characteristics of as component in a compact and effective way. However, every scheduling algorithm has its own schedulability test. Therefore, we need to *abstract* away from the particular scheduler and from the task set structure.

When different components are present in the system, the processor execution time must be allocated to the different components so that every task meet its deadlines. Therefore, we first need to describe the *worst-case temporal requirements* of every component. On the other side, the global scheduler must allocate to each component the less execution time that is possible without compromising any deadline. In other words, we need to compute the *minimum amount of execution time* that component $\mathcal{C}$ must receive in every interval to guarantee that all deadlines are met. We call this function *the temporal profile* $\gamma_{\mathcal{C}}(t)$ of component $\mathcal{C}$.

Example. Consider a component $\mathcal{C}_1$ consisting of only one task τ_1 with $C_1 = 3$msec, $D_1 = 5$msec, $T_1 = 10$msec. It is clear that the minimum amount of time that the component must receive in every interval of lenght $t = 5$msec is 3msec, otherwise the task could miss its deadline. For all intervals of lenght $5 \leq t < 15$, the amount of time is still 3, whereas for intervals of lenght $15 \leq t < 25$ the amount of time is 6. The corresponding function $\gamma_{\mathcal{C}}(t)$ is depicted in Figure 2.

When considering components with more that one task, the profile function depends also on the adopted scheduling algorithm. In this paper we consider three different schedulers.

Earliest Deadline First. This algorithm schedules the job with the earliest deadline at every instant. EDF is optimal, in the sense that if a task set is schedulable by any other algorithm, then it is schedulable by EDF. Baruah et al. [2] found a schedulability

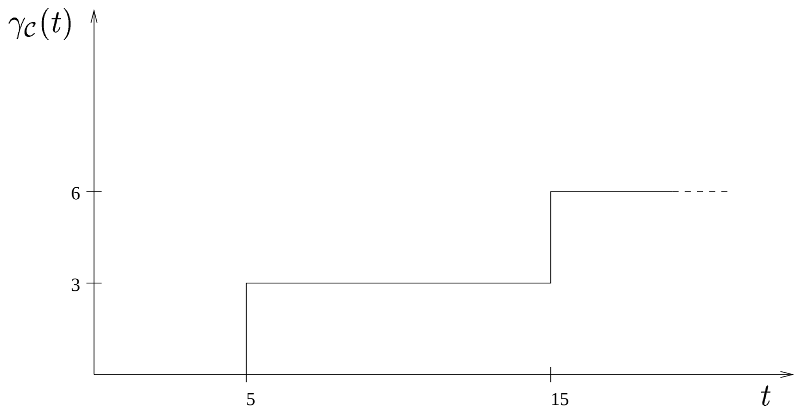

Figure 2: Function $\gamma_{\mathcal{C}}(t)$ for the component in the example.

analysis for EDF based on the computation of the *demand bound function*:

$$DBF(t) = \sum_{i=1}^{n} \max\left\{0, \left(\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right) C_i\right\}$$

A set of real-time tasks is schedulable by EDF if in every interval of time the demand bound function is not greater than the available execution time:

$$\forall t > 0 \qquad DBF(t) \leq t$$

It is straigthforward to show that, the profile function $\gamma_{\mathcal{C}}(t)$ for a component $\mathcal{C}$ with a EDF scheduler is its demand bound function.

Time Triggered Scheduling. According to this paradigm, a schedule is pre-computed off-line and stored in a table. On-line, a dispatcher reads the table as the time passes and schedules the corresponding job. Fohler [5] presented an algorithm, called *Slot Shifting*, for scheduling aperiodic requests in a Time Triggered Architecture. The algorithm first reduces the table driven schedule in a sequence of *target windows*: each job J_{ij} of task τ_i is assigned a target window $[r_{ij}, d_{ij}]$ with the meaning that the job must be executed after r_{ij} and must terminate before d_{ij}. The table driven schedule can then be transformed in a EDF schedule of jobs $\{J_{ij}\}$, where r_{ij} is the release time of job J_{ij} and d_{ij} is its deadline. Hence, it is straightforward to apply the same analysis as EDF, and compute a profile function as follows:

$$\gamma_{\mathcal{C}}(t) = \max_{t_1, t_2}\{D(t_1, t_2)\}$$

where

$$D(t_1, t_2) = \sum_{t1 \leq r_{ij} \quad d_{ij} \leq t_2} C_i.$$

Fixed Priority. In this paradigm, every task is assigned an integer that represents its priority. At any instant, the job corresponding to the task with the highest priority is executed. Many schedulability tests have been devised for a fixed priority scheduler. It has been proved [7] that the Deadline Monotonic priority assignment (i.e. every task is assigned a priority inversely proportional to its relative deadline D_i) is optimal.

Bini and Buttazzo [3] presented a necessary and sufficient test for fixed priority scheduling that can be easily generalized for our purpouses. Lipari and Bini [6] presented a method for computing an entire class of profile functions that bound the requirement of a fixed priority task set. These functions are of the form:

$$\xi_{\alpha,\Delta}(t) = \Delta + \alpha t$$

where α and Δ are two parameters that depend on the particular task set. All the functions $\xi_{\alpha,\Delta}(t)$ have the property that if the component is assigned at least $\xi_{\alpha,\Delta}(t)$ units of execution time in every interval of time t, then every deadline in the task set is met. The interested reader can contact the authors for more details on this methodology.

3 Global scheduling

Now, we need a global algorithm for scheduling the processor execution time among the system components. Many algorithms have been proposed until now, which can be roughly divided in two classes: static slot allocation, and dynamic scheduling. In the first class, there are algorithms that statically divide the time-line in slots, and then assign slots to the different components. In the second class, there are scheduling algorithms that consider each component as an *aperiodic real-time task*: these algorithms use classical techniques of the real-time scheduling theory commonly referred as *servers*. In this paper we will consider a particular server algorithm, called *Constant Bandwidth Server* (CBS) [1]: however, the methodology described in this paper can be applied to other kind of server algorithms as well.

A server is characterized by two parameters (Q, P), where Q is the *maximum budget*, and P is the server *period*, with the meaning that the component can execute for Q units of time every P units. The server dynamically maintains two internal variables q and d that are updated according to the server rules [1]. The system consists of a set of servers scheduled by EDF: the server with the earliest d is scheduled at each instant.

The previous algorithm has the following property [1]:

Property 1 *Given a system consisting of n servers $S_i = (Q_i, P_i)$, with $\sum_{i=1}^{n} \frac{Q_i}{P_i}$, then every server S_i that is continuosly backlogged is guaranteed to receive at least Q_i units of execution time every P_i units of time.*

However, we do not know exactly *when* the component will receive execution, because it depends also on the presence of other servers in the system. Therefore, it

[1]Due to space constraints, we cannot describe here the entire server algorithm. Please, refer to [1] for a complete description of the algorithm.

may be possible that the component receive all the needed computation time at the beginning of the interval $[kP, (k+1)P]$, and then executes in $[kP, kP+Q]$; or it may happen that it receive all computation time at the end of the interval $[kP, (k+1)P]$, that it in $[(k+1)P-Q, (k+1)P]$; or that the execution is scattered along the interval.

3.1 Characteristic function of a server

Given a component, we need to find the server parameters (Q, P) that make its task set feasible. We already know how to compute the temporal profile of a component $\gamma_C(t)$, which is the minimum amount of execution time that is needed in every interval of time for guaranteeing that every deadline is met. Now we need to compute the minimum amount of time $Z_{S_i}(t)$ that is assigned to a server S_i, with parameters (Q_i, P_i) in every interval of time of lenght t. We call function $Z_{S_i}(t)$ *characteristic function* of server S_i. Finally, to guarantee the schedulability of the component, we need to check that

$$\forall t \quad \gamma_C(t) \leq Z_{S_i}(t).$$

Lipari and Bini [6], provided a method for computing the characteristic function of a server.

Theorem 1 *Given a server algorithm defined by the rules of Section 3, and with parameters (Q, P), its characteristic function is the following:*

$$Z_S(t) = \begin{cases} 0 & 0 < t \leq P-Q \\ (k-1)Q & kP-Q < t \leq (k+1)P-2Q \\ t-(k+1)(P-Q) & (k+1)P-2Q < t \leq (k+1)P-Q \end{cases}$$

where k is an integer greater than zero and equal to $\left\lceil \frac{t-(P-Q)}{P} \right\rceil$.

The function $Z_S(t)$ for a server with $Q = 5$ and $P = 8$ is plotted in Figure 3.

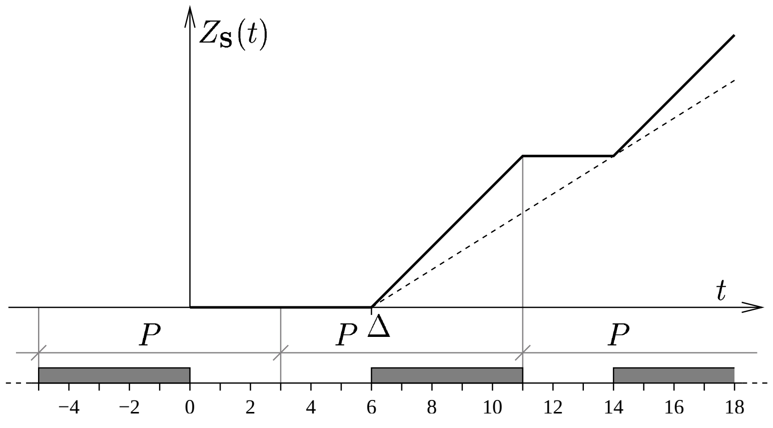

Figure 3: Characteristic function of server $S = (5, 8)$.

4 Integration

When one new component $\mathcal{C}$ must be inserted in the system, we have to proceed as follows:

Temporal profile. First, we have to characterize the temporal profile $\gamma_{\mathcal{C}}(t)$ of our component. This depends on the number of tasks in the component, on their parameters and on the scheduling algorithm. In case the scheduling algorithm is EDF or TT, we have one choice; in the case of fixed priority, we have a class of bounding functions. Remember that the profile function is the maximum amount of time that the component requires in every interval of time t in order to guarantee the deadlines of all its tasks.

Server parameters. The component must be *encapsulated* in a *server* with the proper parameters. The server is described by its *characteristic function* $Z_S(t)$, which represents the minimum amount of execution that in every interval of time t the server can guarantee to the component. In order to make all tasks in the component schedulable, we must compute the parameters (Q, P) (and hence the $Z_S(t)$) such that

$$\forall t > 0 \qquad Z_S(t) \geq \gamma_{\mathcal{C}}(t)$$

In general, several pairs (Q, P) satisfy the previous inequality: we will obtain a class of parameters (Q, P) among which we can choose the best pair according to some user-defined metrics. The procedure is a slightly more complex for fixed priority schedulers, as there are several possible $\gamma_{\mathcal{C}}(t)$ for the same component. Due to space limitations, the exact procedure for computing (Q, P) is not reported here: the interested reader can refer to [6] or contact the authors.

System load. Then, we have to guarantee that the new server fits well in the system

$$\sum_{i=1}^{N} \frac{Q_i}{P_i} \leq 1$$

where N is the number of components. In case the above condition is not respected, our system may be unschedulable, i.e. some task may miss some deadline. Therefore, we have to go back and change something. We can either eliminate one component or reduce its computational requirement.

Communication. Now we consider the interaction between the different components. We have to check that, for every input port, the maximum delay constraint is respected. We first compute the maximum delay for the required data as follows:

$$\sum_{j=1}^{m} \delta_{jk} + \pi_{jk} + 2\phi_{jk}$$

where $\mathcal{C}_1, \ldots, \mathcal{C}_m$ are all components that elaborate the data from the original source to the considered component $\mathcal{C}_i$. Expression $\pi_{jk} + 2\phi_{jk}$ takes into account the delay that it takes to pass the data from component $\mathcal{C}_j$ to component $\mathcal{C}_{j+1}$.

Hence, we have to check that

$$\Delta_{ik} \geq \sum_{j=1}^{m} \delta_{jk} + \pi_{jk} + 2\phi_{jk}$$

If all the previous steps were succesfully completed, our system is feasible: all tasks of all components will meet their deadlines and all constraints on end-to-end delay are respected. If something went wrong, the only possibility is to go back and change something. For example, we could use a simpler component that requires less computation time; or we can enforce better constraints on the data.

5 Conclusions

In this paper we consider the problem of integrating real-time components, each one with its own scheduler in the same system. The framework permit to define and characterize each component in isolation from the rest of the system, and then use an integration procedure for computing the global scheduler parameters and check that the final system respects all temporal constraints.

The definition of component used in this work is very simple: in particular, components can only communicate through asynchronous buffers. This simplifies the integration analysis as the component temporal profiles do not depend on each other. Currently we are exploring extensions different models of communications, that include synchronous and asyncronous procedure calls.

References

[1] Luca Abeni and Giorgio C. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the* 19th *IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[2] S.K. Baruah, A.K. Mok, and L.E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 182–190, December 1990.

[3] Enrico Bini and Giorgio C. Buttazzo. The space of rate monotonic algorithm. In *Proceedings of the* 23rd *IEEE Real-Time Systems Symposium*, December 2002.

[4] G. C. Buttazzo. Hartik: A real-time kernel for robotics applications. In *IEEE Real-Time Systems Symposium*, December 1993.

[5] G. Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *Proceedings of the 16th Real Time System Symposium, Pisa, Italy*, December 1995.

[6] Giuseppe Lipari and Enrico Bini. Resource partitioning among real-time applications. Submitted to Euromicro Conference on Real-Time Systems 2003.

[7] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):40–61, 1973.