

# Semantics-Preserving and Memory-Efficient Implementation of Inter-Task Communication on Static-Priority or EDF Schedulers\*

S. Tripakis, C. Sofronis, N. Scaife and P. Caspi  
Verimag laboratory, Centre Equation, 2, avenue de Vignate, 38610 Gières, France

## ABSTRACT

In previous work, we have proposed a method of preserving the functional semantics of model-based designs by the use of static checks and a double-buffer protocol [12]. However, this is restricted to static, fixed-priority scheduling and for high-priority to low-priority communications requires a double buffer to be stored for each pair of communicating tasks. In this paper we extend the method to dynamic-priority scheduling in the form of *earliest-deadline-first* (EDF) scheduling and show that, although scheduling is dynamic, a static buffering scheme can still be used. We also suggest some memory optimizations of our protocol which still preserve the original functional semantics. Finally, we show how model checking can be used to prove correctness of the scheme.

**Categories and Subject Descriptors:** D.2.2 [Design Tools and Techniques], D.2.3 [Coding Tools and Techniques].

**General Terms:** Design, Reliability, Verification.

**Keywords:** Model-based design, Embedded software, Semantical preservation, Process communication, Scheduling.

## 1. INTRODUCTION

Model-based design is being established as an important paradigm for modern embedded software development. The main principle of the paradigm is to use models (with formal semantics) all along the development cycle, from design, to analysis, to implementation. Using models rather than, say, building prototypes is essential for keeping the development costs manageable. However, models alone are not enough. They need to be accompanied by powerful tools for analysis (e.g., model-checking) and implementation (e.g., code generation). Automation here is the key: high system complexity

\*This work has been partially supported by the European projects RISE (IST-2001-38117) and ARTIST2 (IST-004527).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

EMSOFT'05, September 19–22, 2005, Jersey City, New Jersey, USA.

Copyright 2005 ACM 1-59593-091-4/05/0009 ...\$5.00.

and short time-to-market make model reasoning a hopeless task, unless it is largely automatized.

Design is facilitated by high-level models, which allow the designer to focus on essential functionality and algorithmic logic, while abstracting from implementation concerns. This is also important in order to make designs platform-independent. High-level models, therefore, often assume “ideal” semantics, such as concurrent, zero-time execution of an unlimited number of components. This is the case for very popular tool-boxes like “Simulink/Stateflow”<sup>1</sup>, as well as synchronous languages [2].

Naturally, these assumptions break down upon implementation. This often results in implementations which do not preserve the original semantics. In turn, the results obtained by analyzing the model (e.g., model satisfies a given property) may not hold at the implementation level. In order not to lose the benefits of using high-level models, then, the following issue needs to be addressed: how can the semantics of the high-level model be preserved while relaxing the ideal semantical assumptions?

*Related works.* In the past, several answers have been brought to this question:

- When time-triggered systems are derived from continuous processes, the solution is usually based on numerical analysis, stability and jitter considerations [1].
- In [5], the Esterel [4] programming language is coupled with the Kronos timed-automaton model-checker [7] so as to check whether the actual timing is compatible with the synchronous hypothesis. The approach is endowed with a compiler, a simulator and a debugger.
- In [8], a different approach is taken which departs from the synchronous assumption by considering “logical execution times”. This approach can be valuable in some cases but requires mixing implementation details with the modelling process. It is also relatively untested and its domain of interest needs to be assessed more thoroughly.
- The Simulink/Stateflow documentation claims that the RealTimeWorkshop code generator is able to provide implementations that reproduce the deterministic behavior’ of the model.<sup>2</sup> However, no references are given in support of this claim.

<sup>1</sup>Simulink and Stateflow are trademarks of The Mathworks Inc.: <http://www.mathworks.com>.

<sup>2</sup> Quoting from Section “Mapping Model Execution to the

*Our contributions.* In previous work [12] we have partially addressed the question, by proposing an inter-task communication scheme which preserves the ideal, zero-time semantics of a set of tasks running under static-priority, preemptive scheduling.

This paper continues this work, presenting two main contributions. First, we extend the scheme to the case of EDF (earliest-deadline first) scheduling. Second, we propose a set of optimizations of the memory requirements of the scheme for the frequent case of multi-periodic tasks.

More precisely, regarding the first point, we show that a “static” buffering scheme like the one used under static-priority scheduling can also be used under EDF scheduling. By static we mean that the buffers are determined at compile-time and do not change at run-time. This may seem surprising, knowing that the buffering scheme used under static-priority scheduling is different depending on the relative priorities of the communicating tasks. Since, under EDF, the relative priorities of two tasks can change dynamically, one might conclude that the buffering needs to be dynamic as well. We show that this is not the case. In particular, we show that, when task  $\tau_i$  communicates data to task  $\tau_j$ , if the (relative) deadline of  $\tau_i$  is smaller than the one of  $\tau_j$  then the high-to-low priority buffering scheme can be used; otherwise, the low-to-high priority buffering scheme can be used.

Concerning the optimizations, we note that for low-to-high priority communications we need a double buffer for each writer so there is no additional overhead for writes to multiple readers. The opposite is true for high-to-low communications, we need a double buffer for each reader so one-to-many communications have a much higher space complexity. In this paper, we consider various situations in which we can ameliorate this problem for the popular *multi-periodic* case, when static-priority, rate-monotonic scheduling is used [9]. There are various special cases which we could consider and there are also implementation “tricks” which we could use to save space under these conditions but we restrict ourselves to a small set of commonly-occurring scenarios and retain the original *abstract* treatment of the problem. We do, however, consider the atomicity of communications which can have a significant impact upon the overall space-complexity of our method.

We show that, in the general multi-periodic case,  $n + 1$  single buffers are both sufficient and necessary, in the worst case, for each writer task communicating with  $n$  reader tasks. This is to be compared to the  $2n$  requirement of the general scheme. We also provide an algorithm which, given a set of multi-periodic tasks, computes the minimal number of buffers required by this set (this number is often less than  $n + 1$ ). The algorithm also computes the indexing scheme, that is, which buffer a task should read from or write to at any given time. We also consider the special case of *harmonic* multi-periodic systems, those for which the periods are in consecutive powers-of-two, i.e., task  $\tau_i$  has period  $2^{i-1}$ . We show that, without assumptions on the atomicity of reads and writes, the requirement is reduced from  $n + 1$

to  $n$  buffers. Assuming atomicity, on the other hand, permits to further reduce this to  $n/2$ . Such reductions can be significant as the number of writers increases. In particular, for  $n$  writers each communicating to all lower-priority tasks (i.e., writer  $i$  communicates to tasks  $i + 1, \dots, n$ ), the optimizations permit us to reduce the number of buffers from the  $n(n-1)$  required by the generic scheme to  $n(n+1)/2 - 1$ , to  $n(n-1)/2$ , to  $n(n-1)/4$ , in the multi-periodic, harmonic and atomic-harmonic cases, respectively. Notice that these are worst-case requirements which can be further optimized using the algorithm discussed above.

The rest of the paper is organized as follows. Section 2 presents the task model with ideal semantics. Section 3 discusses the execution under static-priority of EDF scheduling and issues of semantical preservation during implementation. Section 4 presents the semantics-preserving buffering schemes in both scheduling cases. Section 5 presents the buffer optimizations. Section 6 discusses a correctness proof method based on model-checking. Section 7 presents the conclusions and future work directions.

## 2. AN INTER-TASK COMMUNICATION MODEL

We consider a set of *tasks*,  $\mathcal{T} = \{\tau_1, \tau_2, \dots\}$ . The set need not be finite, which allows the modelling of, for example, dynamic creation of tasks.

To model inter-task communication, we consider a set of *links* of the form  $(i, j, p)$ , with  $i, j \in \{1, 2, \dots\}$  and  $p \in \{-1, 0\}$ . If  $p = 0$  then we write  $\tau_i \rightarrow \tau_j$ , otherwise, we write  $\tau_i \xrightarrow{-1} \tau_j$ . A link  $(i, j, p)$  means that task  $\tau_j$  receives data from task  $\tau_i$ . If  $p = 0$  then  $\tau_j$  receives the last value produced by  $\tau_i$ , otherwise, it receives the one-before-last value (i.e., there is a “unit delay” in the link from  $\tau_i$  to  $\tau_j$ ). In both cases, it is possible that the first time that  $\tau_j$  occurs<sup>3</sup> there is no value available from  $\tau_i$  (either because  $\tau_i$  has not occurred yet, or because it has occurred only once and  $p = -1$ ). To cover such cases, we will assume that for each task  $\tau_i$  there is a *default output* value  $y_0^i$ . Then, in cases such as the above,  $\tau_j$  uses this default value.

*Zero-time semantics.* We associate with this model an “ideal”, *zero-time* semantics. For each task  $\tau_i$  we associate a set of *occurrence times*  $T_i = \{t_1^i, t_2^i, \dots\}$ , where  $t_k^i \in R_{\geq 0}$  and  $t_k^i < t_{k+1}^i$  for all  $k$ . Because of the zero-time assumption, the occurrence time captures the release, start and finish times of a task. In the next section, we will distinguish these three times. We make no assumption on the occurrence times of a task. This allows us to capture all possible situations, namely, where a task is *periodic* (i.e., released at multiples of a given period) or where a task is *aperiodic* or *sporadic*.

We assume that  $T_i \cap T_j = \emptyset$  for all tasks such that  $\tau_i \rightarrow \tau_j$  and  $\tau_j \rightarrow \tau_i$ . That is, if task  $\tau_i$  receives data from task  $\tau_j$  and vice versa, without unit delay, then the two tasks cannot occur at the same time.

Given time  $t \geq 0$ , we define  $n_i(t)$  to be the number of times that  $\tau_i$  has occurred before  $t$ , that is:

$$n_i(t) = |\{t' \in T_i \mid t' \leq t\}|.$$

Target Environment” of [10]: “A correctly executing application will generate deterministic results that are identical to the results produced by the model in simulation. To achieve correct execution, the model’s sample rates must be mapped into corresponding tasks executing in the target environment.”

<sup>3</sup> As we shall see shortly, we define an “ideal” zero-time semantics where a task executes and produces its result as the same time it is released. We can thus say “task  $\tau_i$  occurs”.

We denote inputs of tasks by  $x$ 's and outputs by  $y$ 's. Let  $y_k^i$  denote the output of the  $k$ -th occurrence of  $\tau_i$ . Given a link  $\tau_i \rightarrow \tau_j$ ,  $x_k^{i,j}$  denotes the input that the  $k$ -th occurrence of  $\tau_j$  receives from  $\tau_i$ . The ideal semantics specifies that this input is equal to the output of the last occurrence of  $\tau_i$  before  $\tau_j$ , that is:

$$x_k^{i,j} = y_l^i, \text{ where } l = n_i(t_k^j).$$

Notice that if  $\tau_i$  has not occurred yet then  $l = 0$  and the default value  $y_0^i$  is used.

If the link has a unit delay, that is,  $\tau_i \xrightarrow{-1} \tau_j$ , then:

$$x_k^{i,j} = y_l^i, \text{ where } l = \max\{0, n_i(t_k^j) - 1\}.$$

### 3. EXECUTION ON STATIC-PRIORITY OR EDF SCHEDULERS

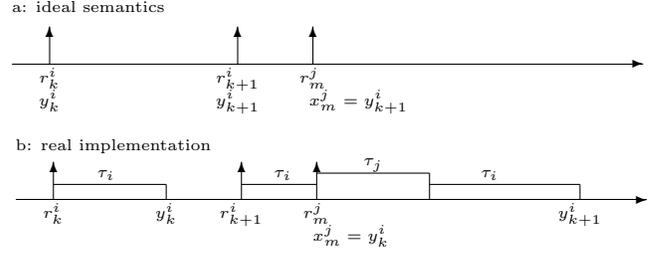
We consider the situation where tasks are implemented as stand-alone processes executing on mono-processor execution platform equipped with an operating system. The latter implements a given scheduling policy to determine which of the ready tasks (i.e., tasks released but not yet completed) is to be executed at a given point in time. We consider two scheduling policies:

- *Static-priority*: each task is assigned a unique *priority* (to avoid ambiguities, we assume no two tasks have the same priority); the task with the highest priority among the ready tasks executes.
- *Earliest-deadline first* or EDF: each task is assigned a unique *deadline* (to avoid ambiguities, we assume no two tasks have the same deadline); the task with the earliest deadline among the ready tasks executes.

In the ideal semantics, task execution takes zero time. In reality, this is obviously not true. A task is released and becomes ready. At some later point it is chosen by the scheduler to execute. Until it completes execution, it may be preempted a number of times by other tasks. To capture this, we distinguish the release time of a task  $\tau_i$  from the time  $\tau_i$  begins execution and from the time  $\tau_i$  ends execution. For the  $k$ -th occurrence of  $\tau_i$ , these three times will be denoted  $r_k^i$ ,  $b_k^i$  and  $e_k^i$ , respectively.

*Problems with a “naive” implementation.* Our purpose is to implement the set of tasks so that the ideal semantics are preserved by the implementation. It is worth examining a few examples in order to see that a straightforward implementation does not preserve the ideal semantics. Let us then consider a simple implementation scheme where, for each link  $\tau_i \rightarrow \tau_j$ , there is a buffer  $B_{i,j}$  used to store the data produced by  $\tau_i$  and consumed by  $\tau_j$ . A first concern is data integrity: a task writing on this buffer might be preempted before it finishes writing, leaving the buffer in an inconsistent state. To avoid this, we will also assume that the simple implementation scheme uses *atomic* reads and writes, so that a task writing to or reading from a buffer cannot be preempted before finishing. We will also assume that all reads happen at the beginning and all writes at the end of execution of a task.

Finally, we will assume that the set of tasks is *schedulable*. This means that no task ever violates its (absolute) deadline. In the static-priority case, we assume that the absolute deadline is the next release time of the task, that is,



**Figure 1:** In the semantics,  $x_m^j = y_{k+1}^i$ , whereas in the implementation,  $x_m^j = y_k^i$ .

the absolute deadline of the  $k$ -th occurrence of  $\tau_i$  is  $r_{k+1}^i$ . In the EDF case, if  $d_i$  is the (relative) deadline of  $\tau_i$ , then the absolute deadline of the  $k$ -th occurrence of  $\tau_i$  is  $r_k^i + d_i$ .<sup>4</sup>

Even with the above provisions, the ideal semantics are not always preserved. Consider, as a first example, the case  $\tau_i \rightarrow \tau_j$ , where static-priority scheduling is used and  $\tau_i$  has lower priority than  $\tau_j$ . Consider the situation shown in Figure 1. We can see that, according to the semantics, the input of the  $m$ -th occurrence of  $\tau_j$  is equal to the output of the  $(k+1)$ -th occurrence of  $\tau_i$ . However, this is not true in the implementation, because  $\tau_j$  preempts  $\tau_i$  before the latter has time to finish, thus, before it has time to write its result.

In fact, there is no solution to this problem<sup>5</sup> unless we require that, whenever  $\tau_i$  has lower priority than  $\tau_j$  and  $\tau_j$  receives data from  $\tau_i$ , a unit delay is used between the two tasks, in other words, the link must be:  $\tau_i \xrightarrow{-1} \tau_j$ .

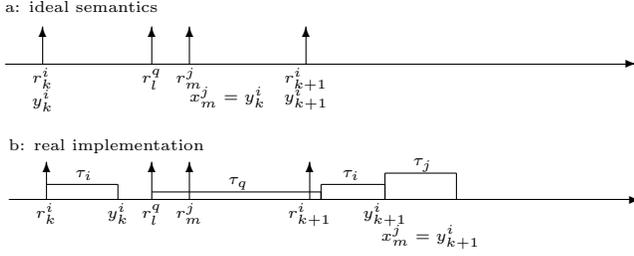
Even when the above requirement is satisfied, the simple implementation scheme is not correct. An example which shows this in the low-to-high priority case is given in [12]. Next, we give an example for the high-to-low priority case.

Consider again that  $\tau_i \rightarrow \tau_j$ . Assume static-priority scheduling and suppose that  $\tau_i$  has higher priority than  $\tau_j$ . Consider also a third task  $\tau_q$  with higher priority than both  $\tau_i$  and  $\tau_j$ . Consider the situation shown in Figure 2. We can see that, according to the semantics, the input of the  $m$ -th occurrence of  $\tau_j$  is equal to the output of the  $k$ -th occurrence of  $\tau_i$ . However, this is not true in the implementation, because  $\tau_q$  “masks” the order of arrival of  $\tau_j$  and  $\tau_i$  ( $r_m^j < r_{k+1}^i$ ). As a result, the order of execution of  $\tau_j$  and  $\tau_i$  is reversed.

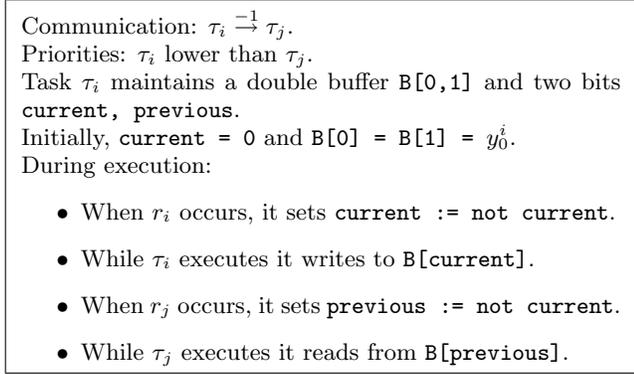
These two examples show that a simple implementation scheme like the one above will fail to respect the ideal semantics. Note that the problems are not particular to static-priority scheduling. Similar situations can happen with EDF scheduling, depending on the deadlines of the tasks. In particular, the situation shown in Figure 1 can occur un-

<sup>4</sup> Obviously, schedulability depends on the assumptions made on the release times and execution times of tasks. A large amount of work exists on schedulability analysis techniques for different sets of assumptions, see, for instance [9] for the multi-periodic case. Notice, however, that our assumption of schedulability is not related to a specific schedulability analysis method: it cannot be, since we make no assumptions on release times and execution times of tasks.

<sup>5</sup> There are solutions where the receiving task “blocks” and waits for the sending task to finish, even though the latter has lower priority. In this work, we are interested in *wait-free* solutions because they are easier to implement.



**Figure 2:** In the semantics,  $x_m^j = y_k^i$ , whereas in the implementation,  $x_m^j = y_{k+1}^i$ .



**Figure 3:** Low-to-high priority communication scheme

der EDF scheduling if  $r_m^j + d_j < r_{k+1}^i + d_i$ . The situation shown in Figure 2 can occur under EDF scheduling if  $r_l^q + d_q < r_{k+1}^i + d_i < r_m^j + d_j$ .

## 4. SEMANTICS-PRESERVING IMPLEMENTATION

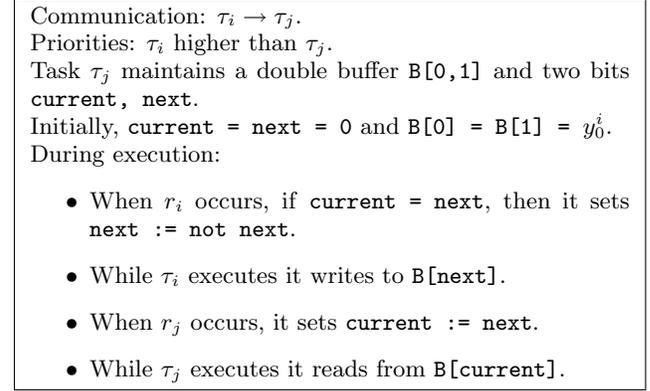
An implementation scheme which preserves the ideal semantics under static-priority scheduling was first proposed in [12]. In this section, we extend this scheme to deal with EDF scheduling as well. The extensions rely on the original scheme, thus, we first recall the latter.

### 4.1 Semantics-preserving implementation under static-priority scheduling

The main feature of the inter-task communication scheme is that, contrary to the “naive” implementation scheme of the previous section, it relies on actions being taken not only while tasks execute but *also when they are released*. These actions are very simple (and inexpensive) bit manipulations. They can therefore be provided as operating system support.

Assuming that  $\tau_i$  is the writer and  $\tau_j$  the reader, there are two buffering schemes, depending on the relative priorities of  $\tau_i$  and  $\tau_j$ . The two schemes are described in Figures 3 and 4. Notice that in the low-to-high case we assume a unit-delay between writer and reader. In the low-to-high scheme, a double buffer and two Boolean variables are maintained by the writer. Note that the same buffer and variables can be used for all possible readers of this writer. In the high-to-low scheme, a double buffer and two Boolean variables

are maintained by the reader. In this case, the reader must maintain one such triplet for each writer it reads from.



**Figure 4:** High-to-low priority communication scheme

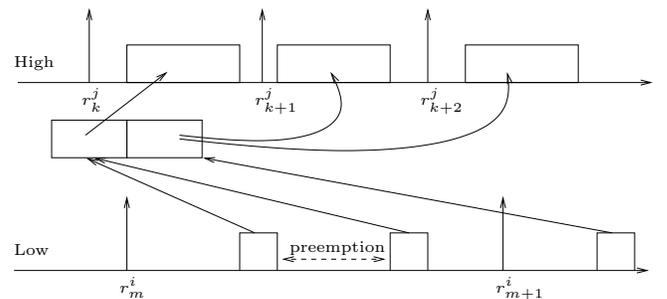
Typical execution scenarios are illustrated in Figures 5 and 6. One time axis is shown for each task: notice that the low-priority task is preempted in both cases by the second occurrence of the high-priority task.<sup>6</sup> The double buffer is shown in the middle. The arrows indicate where each task writes to or reads from. It can be checked that the semantics are preserved.

### 4.2 Semantics-preserving implementation under EDF scheduling

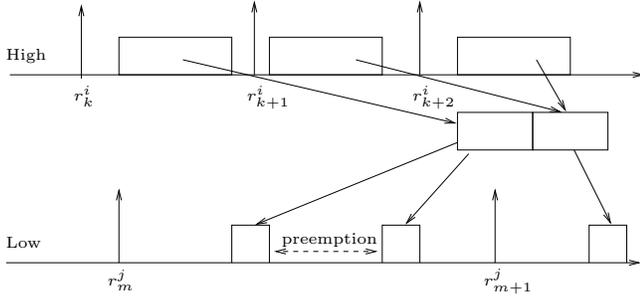
From the previous section we see that, under static-priority scheduling, two different buffering schemes are necessary, one for the low-to-high and the other for the high-to-low priority cases. On the other hand, we know that EDF is a dynamic priority algorithm, where for two tasks  $\tau_i$  and  $\tau_j$ , it is possible that sometimes  $\tau_i$  preempts  $\tau_j$  and sometimes  $\tau_j$  preempts  $\tau_i$ .

The above two facts could lead one to conclude that, under EDF scheduling, the buffering scheme for a case  $\tau_i \rightarrow \tau_j$

<sup>6</sup> It is worth noting that the beginning of execution of the high-priority task does not coincide with its release. This is because, in general, there may be other tasks with even higher priority and they may delay the beginning of the task in question (in fact, they may also preempt it, but this is not shown in the figures).



**Figure 5:** A typical low-to-high communication scenario



**Figure 6: A typical high-to-low communication scenario**

needs to be dynamic, “simulating” either the high-to-low or the low-to-high case, depending on the release times and deadlines of the involved tasks. We show that, fortunately, this is not the case: the buffering scheme can be decided statically. In particular, the buffering scheme depends on the relative deadlines  $d_i$  and  $d_j$  of tasks  $\tau_i$  and  $\tau_j$ , respectively:

- if  $d_i > d_j$  then the low-to-high buffering scheme is used; here we assume a unit delay between  $\tau_i$  and  $\tau_j$ , as in the low-to-high static-priority case, in order to avoid the problem of Figure 1;
- if  $d_i < d_j$  then the high-to-low buffering scheme is used.

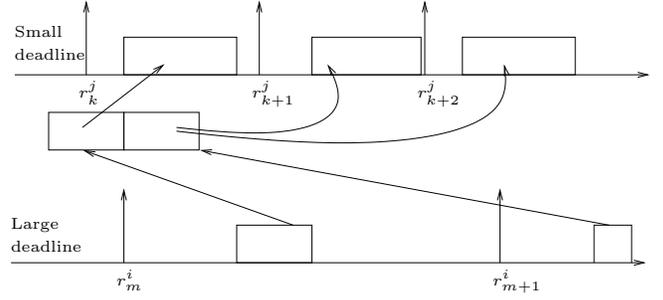
In the discussion that follows, we explain the intuition behind the correctness of this choice. The examples that are provided below do not constitute a proof of correctness, they merely aim at conveying an informal argument. A proof using model-checking is presented in Section 6.

The case  $d_i > d_j$  implies that, if  $\tau_j$  is released before  $\tau_i$  then  $\tau_i$  cannot preempt  $\tau_j$ , neither can it start before  $\tau_i$  ends. Indeed,  $r_k^j \leq r_m^j$  and  $d_j < d_i$  implies  $r_k^j + d_j < r_m^j + d_i$ , that is, the absolute deadline of  $\tau_j$  is smaller than that of  $\tau_i$ . Therefore, we have a situation which is “almost the same” as the low-to-high priority case. The difference is that in the low-to-high priority case  $\tau_j$  *always preempts*  $\tau_i$ , whereas in the EDF case this might not happen. Therefore, in order to guarantee the correctness of the scheme, we must examine this last possibility, to ensure that nothing goes wrong.

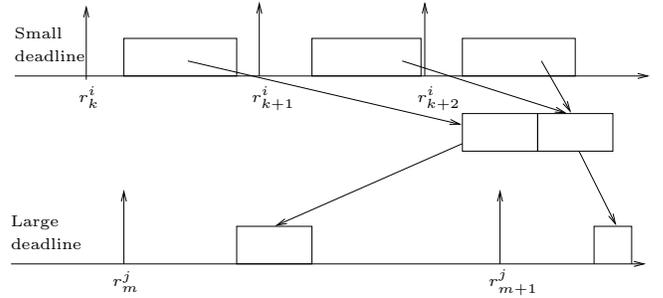
Figure 7 illustrates what might happen when  $\tau_j$  does not preempt  $\tau_i$  as it normally would in the low-to-high static-priority scenario. One can see that this poses no problems for the buffering scheme. In fact, the situation is as if the  $(k+1)$ -th instance of  $\tau_j$  was released after the  $m$ -th instance of  $\tau_i$  finished.

Let us now turn to the case  $d_i < d_j$ . This case implies that, if  $\tau_i$  is released before  $\tau_j$  then  $\tau_j$  cannot preempt  $\tau_i$ , neither can it start before  $\tau_i$  ends. Indeed,  $r_k^i \leq r_m^j$  and  $d_i < d_j$  implies  $r_k^i + d_i < r_m^j + d_j$ , that is, the absolute deadline of  $\tau_i$  is smaller than that of  $\tau_j$ . Therefore, we have a situation which is “almost the same” as the high-to-low priority case. The difference is that in the high-to-low priority case  $\tau_i$  *always preempts*  $\tau_j$ , whereas in the EDF case this might not happen. As before, we must examine this possibility.

Figure 8 illustrates what might happen when  $\tau_i$  does not preempt  $\tau_j$  as it normally would in the high-to-low static-



**Figure 7: The scenario of Figure 5 possibly under EDF:  $\tau_i$  is not preempted**



**Figure 8: The scenario of Figure 6 possibly under EDF:  $\tau_j$  is not preempted**

priority scenario. Again, this poses no problems to the buffering scheme. The situation is as if the  $(k+1)$ -th instance of  $\tau_i$  was released after the  $m$ -th instance of  $\tau_j$  finished.

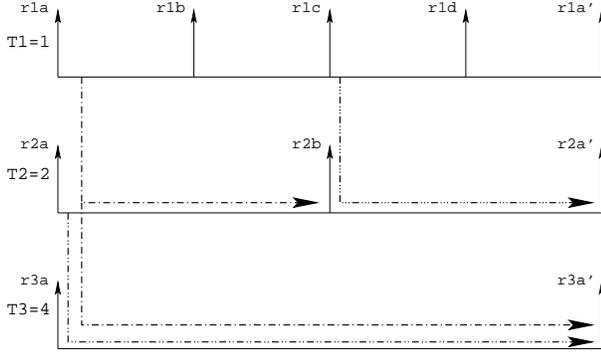
## 5. MEMORY OPTIMIZATIONS FOR THE MULTI-PERIODIC CASE

In this section we consider some optimizations in buffer usage for the special case where we have periodic tasks where period equals deadline. We assume static-priority, rate-monotonic scheduling. The optimizations we propose are for the high-to-low priority scheme. This is the “expensive” scheme, since it requires each reader to maintain a double buffer for each writer. The low-to-high scheme, on the other hand, only requires one double buffer per writer, which can be used by all readers.

### 5.1 Periods in consecutive powers of two, non-atomic case

Consider  $n$  tasks,  $\tau_1, \dots, \tau_n$ , such that task  $\tau_i$  has period  $T_i = 2^{i-1}$  and a priority which is in inverse order of period. Suppose each task sends data to all lower-priority tasks. For three tasks of periods  $T_1 = 1$ ,  $T_2 = 2$  and  $T_3 = 4$ , this gives the situation shown in Figure 9.

We assume that for simultaneous occurrences, the higher-priority task takes precedence and thus transmits its data to the co-incidentally occurring lower-priority task. Thus the first emission of  $\tau_1$  (**r1a**) is required by both  $\tau_2$  and  $\tau_3$ . Furthermore, the data is required to persist until the end of the period of  $\tau_3$ , i.e. until **r3a**<sup>?</sup>. However, the same emission



**Figure 9: Multi-periodic events with consecutive periods in powers of 2**

is required by  $\tau_2$  and this buffer can be shared with  $\tau_3$ . Thus we can implement this system using three single buffers (one for  $r1a$  to  $r2a$  and  $r3a$ , one for  $r2a$  to  $r3a$  and one for  $r1c$  to  $r2b$ ). Our original scheme would require three double buffers.

In fact, it is immediately apparent that for the highest-priority task out of  $n$  tasks we require  $n - 1$  single buffers. Thus, for each writer task  $i$ , we require  $n - i$  single buffers, one for each of the (lower-priority) reader tasks. When every task is a writer, this gives a total of  $n(n - 1)/2$  single buffers. If we used the general scheme of Section 4 we would require  $n(n - 1)/2$  double buffers, that is, double the memory space.

To implement this scheme in practice requires a buffer indexing mechanism. Generating the indexing pattern implied by Figure 9 is quite simple. We must first observe that each alternate emission of the top-level task is unused by lower-priority tasks. Given this, for one writer and  $n$  reader tasks, for emission  $i = \{0, 2, 4, \dots\}$  the writer utilizes buffer:

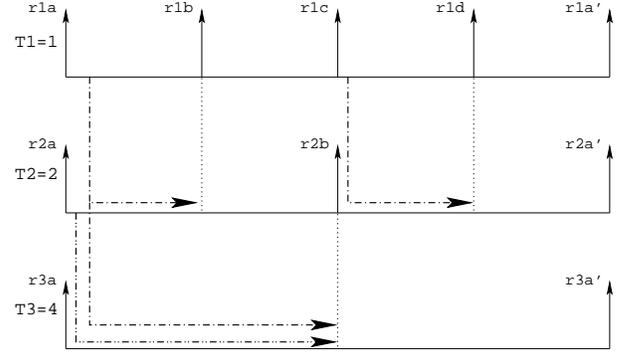
$$B(i) = \min \left\{ j : 0 \leq j \leq n - 1, i \bmod 2^{n-j} = 0 \right\} \quad (1)$$

The writer at time  $i$  writes to buffer  $B(i)$ , any reader occurring at time  $i$  reads from buffer  $B(i)$ . For example, for  $n = 3$  we need 3 buffers indexed 0, 1, 2 and used according to the pattern  $B(0) = 0$ ,  $B(2) = 2$ ,  $B(4) = 1$ ,  $B(6) = 2$ ,  $B(8) = 0$ , and so on. A potentially useful value is the residence time of the data in the buffer which can be computed as:  $R(i) = 2^{n-B(i)}$ .

## 5.2 Periods in consecutive powers of two, atomic case

In Section 5.1 we made no assumptions about the atomicity of data transfers and we were constrained to allow buffers to be occupied for the entire period of a given task. However, we know from the fixed-priority scheduling constraints that a low-priority task  $\tau_j$  must begin execution before the mid-point of its period, i.e.,  $2^{j-2}$ . If it has not managed to do so, this means that the cumulative execution time of higher-priority tasks is at least half the period of  $\tau_j$  and, since all these higher-priority tasks will also be executed at the second half of  $\tau_j$ 's period,  $\tau_j$  will never get to execute. This contradicts our assumption that the system is schedulable.

Now, we assume that tasks sample their data at the start of execution so if we can arrange for reading of the data



**Figure 10: Multi-periodic events with consecutive periods in powers of 2, atomic reads**

to be completed prior to the period mid-point then the related buffer becomes free before the next emissions of higher-priority tasks. If we have a fixed bound on communications we can simply add this time to the execution time for each higher-priority task and the scheduler can then guarantee atomicity in the sense described here. Alternatively, we could rely upon the operating system to provide atomic data transfer. In any case, we can make further savings in the number of required buffers.

Consider Figure 10 which illustrates the situation for three tasks and atomic reads. We can now implement communications from  $\tau_1$  to tasks  $\tau_2$  and  $\tau_3$  using only a single buffer because we can use the same buffer for  $r1a$  to  $r2a$  and for  $r1c$  to  $r2b$ . In fact, for writing task  $\tau_i$ , we need  $\lfloor (n - i)/2 \rfloor$  buffers. The reason for the divide by two is that since only every second emission from task  $\tau_i$  is needed we do not need a new buffer each time we add a new higher-priority task. To see this consider the two processor case in Figure 10 ( $\tau_2$  and  $\tau_3$ ) which needs a single buffer (from  $r2a$  to  $r3a$ ) and compare with the three processor case in the same figure. Since  $r2b$  is ignored, no further buffers are required. For  $n$  tasks, with  $n$  even we need  $(n/2)^2$  buffers, and for  $n$  odd we need  $(n/2)^2 + (n/2)$  buffers in total.

The buffer indexing function for this situation is simply  $B(i)/2$  and the data residence time  $R(i)/2$ .

Non-consecutive powers of two do not pose any serious problems since these are simply subsets of the current analysis. The only complication is that the buffer indexing may require support in the form of index tables rather than analytical formulae as above. More general harmonic cases can also be treated using the methods here, for example periods of  $T_1 = 1$ ,  $T_2 = 2$  and  $T_3 = 6$  are almost the same as for the  $1 - 2 - 4$  case apart from the duplicate emissions from  $\tau_1$  to  $\tau_2$ .

## 5.3 The general multi-periodic case

We finally consider the general multi-periodic case, where each period  $T_i$  is a positive integer (as previously, we assume static-priority, rate-monotonic scheduling). Our first result is that, for one writer task communicating with  $n$  reader tasks,  $n + 1$  single buffers are both necessary and sufficient, in the worst case. Second, we provide an algorithm which, given a set of multi-periodic tasks, computes the minimal number of buffers required by this set (this number is often less than  $n + 1$ ). The algorithm also computes the indexing

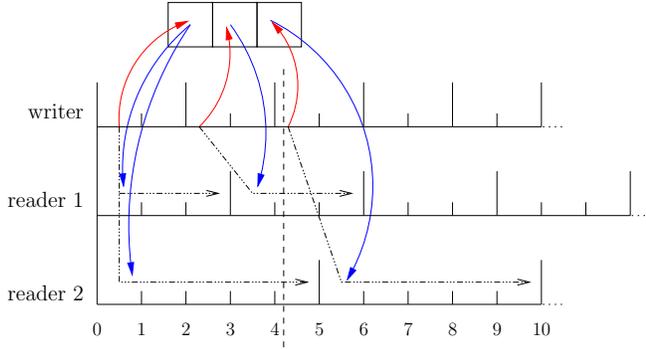


Figure 11: An example of the general multi-periodic case

scheme, that is, which buffer a task should read from or write to at any given time.

The algorithm simulates the execution of all tasks up to the hyper-period  $T_h = lcm\{T_1, \dots, T_n\}$ . It allocates new buffers as necessary, re-using buffers when they are not needed anymore by any reader. As a first step towards explaining the algorithm, observe that if  $\tau_w \rightarrow \tau_j$  then at cycle  $k \in \{0, 1, \dots, T_h - 1\}$ , the reader task  $\tau_j$  needs the value written by the writer task  $\tau_w$  at cycle

$$l(j, k) = \lfloor \frac{\lfloor k/T_j \rfloor T_j}{T_w} \rfloor T_w. \quad (2)$$

$\lfloor k/T_j \rfloor T_j$  gives the last activation of  $\tau_j$  before  $k$  and  $l(j, k)$  is the last activation of  $\tau_w$  before that. In Figure 11 we see an example where the writer has period  $T_w = 2$  and the readers  $T_1 = 3$  and  $T_2 = 5$ . For instance,  $l(1, 7) = 6$  and  $l(2, 7) = 4$ .

The same example shows that  $n + 1$  buffers are necessary, in the worst case. Here we have  $n = 2$  readers and we need 3 buffers. The three buffers are used to store the outputs of the first, second and third occurrences of  $\tau_w$ , respectively. The first output is needed by the first occurrence of both  $\tau_1$  and  $\tau_2$ . The second output is needed by the second occurrence of  $\tau_1$ . The third buffer is necessary in order because when  $\tau_w$  starts writing, after time 4,  $\tau_2$  may not have released the first buffer yet.

The algorithm is shown in Figure 12. The algorithm is run for each writer task, with the corresponding set of reader tasks. Constant  $Th$  is the hyper-period and  $n$  is the number of readers. Variable `buff_no` counts the number of buffers that are needed.  $W[0..Th-1]$  is the indexing array of the writer:  $W[k]$  is the buffer where the writer must write to at cycle  $k$ . No such array is needed for the readers: at cycle  $k$ , reader  $j$  reads from buffer  $W[l(j, k)]$ .  $LW[1..buff\_no]$  is a temporary array:  $LW[b]$  is equal to the last time that the writer wrote in buffer  $b$ . The predicate `needed(k)` “filters” the cycles where the output of the writer is not used by any reader: it is defined formally as  $k \bmod T_w = 0 \wedge \exists j \in \{1, \dots, n\}, \exists k' < T_h, k = l(j, k')$ . The worst-case time complexity of the algorithm is  $O(T_h^2 \cdot n)$ . The amount of memory needed to store the indexing arrays is  $O(T_h \cdot (n+1))$ . However, this can be optimized, since the value  $W[k]$  only changes at the beginnings of the periods.

In order to show that not more than  $n + 1$  single buffers are needed in the general multi-periodic case, we will prove

```

buff_no := 0 ;
for k := 0..Th-1 such that needed(k) do
  if ( exists b in [1..buff_no] such that
    for all j in [1..n]: LW[b] <> l(j,k) ) then
    W[k] := b ;      /* re-use buffer b */
    LW[b] := k ;
  else
    buff_no++ ;     /* add new buffer */
    W[k] := buff_no ;
    LW[buff_no] := k ;
  end if ;
end for ;

```

Figure 12: Buffer optimization algorithm for the general multi-periodic case

that  $buff\_no \leq n + 1$  is an invariant of the loop of the program of Figure 12. The invariant holds after initialization. Since `buff_no` is incremented only in the `else` part, it suffices to prove that, when  $buff\_no = n + 1$ , the `else` part will not be executed, that is, the condition of the `if` part holds. To prove this, observe that for any  $b \neq b'$ , we have  $LW[b] \neq LW[b']$ : this is because each  $k$  is only assigned once in some element of  $LW$ . Thus, when  $buff\_no = n + 1$ ,  $LW$  holds  $n + 1$  distinct elements. On the other hand, there are at most  $n$  distinct elements in the set  $L = \{l(j, k) \mid j = 1, \dots, n\}$ . Thus, by the pigeon-hole principle, there must exist a  $b$  such that  $LW[b] \notin L$ .

## 6. PROOF OF CORRECTNESS BY MODEL-CHECKING

### 6.1 Proof of the general buffering schemes

We can use model-checking to prove the correctness of the (non-optimized) buffering schemes presented in Section 4. A formalization and detailed discussion of the proof for the static-priority case can be found in the technical-report version of [12], available from Verimag’s web site<sup>7</sup>. Here, we summarize the main principles of the proof and show how it can be extended to the EDF case as well.

In order for model-checking to be applicable, the model to be checked must be finite-state. On the other hand, the scheme must be able to deal with an arbitrary number of tasks, which gives rise to an *a priori* infinite model. The solution is based on the following claim: *if the buffering scheme is correct for any pair of writer-reader tasks, then it is correct for any set of tasks*. This claim is correct provided the execution semantics of the writer-reader pair is abstracted in an appropriate way, in order to take into account the effects of the other tasks, which are not modeled.

In particular, the idea is to model each task  $\tau_i$  by three events,  $r_i$ ,  $b_i$  and  $e_i$ , corresponding to the release, beginning and end of an instance of a task, respectively. Then, the execution semantics are captured by placing restrictions on the possible orders of the above events. One restriction is the *cyclic* order of the above events for each task, namely,  $r_i \rightarrow b_i \rightarrow e_i \rightarrow r_i \rightarrow \dots$ . Other restrictions are placed in order to model the scheduling algorithm.

Let us first show how to model static-priority scheduling. Let  $\tau_1$  be the high-priority task and  $\tau_2$  be the low-priority

<sup>7</sup> <http://www-verimag.imag.fr>

task. Then, we know that neither  $b_2$  nor  $e_2$  can occur between  $r_1$  and  $e_1$ . Indeed,  $\tau_2$  cannot start before  $\tau_1$  finishes. Also, if  $\tau_2$  has already started when  $r_1$  occurs then it is preempted, thus, will not finish before  $\tau_1$  finishes. These ordering restrictions can be modeled using a finite-state automaton, or some other finite-state model. In our case, where we use the model-checker Lesar developed for the synchronous language Lustre, they are modeled as Lustre assertions [11].

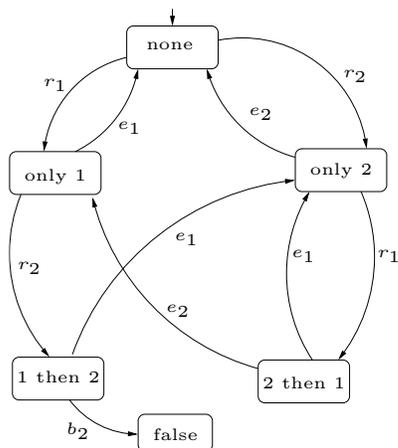


Figure 13: Assumptions modeling EDF scheduling

EDF scheduling can be modeled in a similar way. Let  $\tau_1$  be the task with the smaller deadline and  $\tau_2$  be the task with the larger deadline. Figure 13 shows the restrictions modeled as an automaton: if the state “false” is reached then the restrictions are violated. This happens when  $r_1$  occurs before  $r_2$ , yet  $b_2$  occurs before  $e_1$ : this means that  $\tau_2$  begins before  $\tau_1$  finishes, which cannot happen under EDF. Note that these restrictions are weaker than those imposed for the static-priority case discussed above.

To prove correctness using model-checking, we modeled the high-to-low and low-to-high buffering schemes in Lustre. A *data-independence* property [13] permits to abstract data values using booleans. The above ordering restrictions are used as assertions to model execution semantics under static-priority or EDF schedulers. Finally, we also modeled the ideal semantics and used Lesar to prove that the ideal semantics are preserved by the implementation.

## 6.2 Proof of correctness of the optimized buffering schemes, harmonic case

Model-checking can also be used to prove correctness of the optimized versions, however, this can be done *a priori* only for a given, rather than arbitrary, set of tasks. We have followed this approach and modeled the buffering schemes for the harmonic multi-periodic case (Sections 5.1 and 5.2). In both cases, we have managed to model-check completely only systems of  $n = 3$  tasks. The model gets too large for Lesar to handle for  $n = 4$ . We did manage, however, to partially verify the  $n = 4$  case, selecting various subsets of the model and proving them correct. Since there are no irregularities in these schemes (each case is built by extending the previous one in a regular way) we can expect the scheme to be correct for all  $n$ .

## 7. CONCLUSIONS AND PERSPECTIVES

We have studied the problem of semantics-preserving implementations of inter-task communication. We have extended our previous work which proposed a semantics-preserving buffering scheme for static-priority scheduling to EDF scheduling. We also showed that buffer requirements can be optimized in the multi-periodic case. Finally, we discussed how model-checking can be used to prove the correctness of the buffering scheme.

Our current objective is to extend this work further to multi-processor execution platforms. This has been partly done in [6] for synchronous distributed architectures and multi-periodic tasks, where static, non-preemptive scheduling was assumed. We still need to cover loosely synchronous [3] or asynchronous architectures with preemptive scheduling for more general task arrival patterns.

## 8. REFERENCES

- [1] ASTRÖM, K., AND WITTENMARK, B. *Computer Controlled Systems*. Prentice-Hall, 1984.
- [2] BENVENISTE, A., AND BERRY, G. The synchronous approach to reactive and real-time systems. *Proceeding of the IEEE* 79, 9 (September 1991), 1270–1282.
- [3] BENVENISTE, A., CASPI, P., GUERNIC, P. L., MARCHAND, H., TALPIN, J., AND TRIPAKIS, S. A protocol for loosely time-triggered architectures. In *EMSOFT’02 (2002)*, vol. 2491 of *LNCS*, Springer.
- [4] BERRY, G. *The foundations of Esterel*. MIT Press, 2000, pp. 425–454.
- [5] BERTIN, V., CLOSSE, E., POIZE, M., PULOU, J., SIFAKIS, J., VENIER, P., WEIL, D., AND YOVINE, S. Taxys = Esterel + Kronos. A tool for verifying real-time properties of embedded systems. In *CDC’01 (2001)*, IEEE.
- [6] CASPI, P., CURIC, A., MAIGNAN, A., SOFRONIS, C., TRIPAKIS, S., AND NIEBERT, P. From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. In *LCTES’03 (2003)*, ACM.
- [7] DAWS, C., OLIVERO, A., TRIPAKIS, S., AND YOVINE, S. The tool Kronos. In *Hybrid Systems III (1996)*, vol. 1066 of *LNCS*, Springer.
- [8] HENZINGER, T., KIRSCH, C., SANVIDO, M., AND PREE, W. From control models to real-time code using Giotto. *IEEE Contr. Sys. Mag.* 23, 1 (2003).
- [9] LIU, C., AND LAYLAND, J. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM* 20, 1 (Jan. 1973), 46–61.
- [10] THE MATHWORKS INC. *Developing Embedded Targets for Real-Time Workshop Embedded Coder (R13)*.
- [11] RATEL, C., HALBWACHS, N., AND RAYMOND, P. Programming and verifying critical systems by means of the synchronous data-flow programming language Lustre. In *ACM-SIGSOFT Conference on Software for Critical Systems (1991)*.
- [12] SCAIFE, N., AND CASPI, P. Integrating model-based design and preemptive scheduling in mixed time- and event-triggered systems. In *ECRTS’04 (2004)*, IEEE.
- [13] WOLPER, P. Expressing interesting properties of programs in propositional temporal logic. In *Proc. 13th ACM Symp. POPL (1986)*, pp. 184–192.