Unité Mixte de Recherche 5104 CNRS - INPG - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

Integrating model-based design and preemptive scheduling in mixed time- and event-triggered systems

N. Scaife and P. Caspi

Report n^o TR-2004-12

June 1, 2004

Integrating model-based design and preemptive scheduling in mixed time- and event-triggered systems

N. Scaife and P. Caspi

June 1, 2004

Abstract

Model-based design is advocated as the method of choice when dealing with critical systems as well as high quality systems. However, it often abstracts implementation details such as execution times. This can be a problem when dealing with urgent events whose implementation requires preemptive scheduling. In this paper, we propose an inter-task communication mechanism on top of a fixed-priority deadline monotonic preemptive execution scheme, which preserves the ordering of computations validated in a "zero-time" synchronous framework. Then, we formally prove the correctness of the approach.

Keywords: model-based design, synchrony, fixed priority, preemptive scheduling, wait-free communication, double buffers.

Reviewers: Pascal Raymond

Notes: This work has been partially supported by the European Community through IST projects Next-TTA and Rise. This report presents an extended version of a paper presented at the Euromicro Conference on Real-Time Systems ECRTS04

How to cite this report:

@techreport { ,
title = { Integrating model-based design and preemptive scheduling in mixed time- and event-triggered systems},
authors = { N. Scaife and P. Caspi},
institution = { Verimag Technical Report },
number = {TR-2004-12},
year = { 2004},
note = { }
}

Contents

1 Introduction

Embedded and real-time systems are often safety-critical and require high quality design and guaranteed properties. For systems such as this, model-based design has been advocated as the method of choice for dealing with them. The design process consists of building models on which the required system properties are carefully checked and assessed and then deriving implementations such that these properties are preserved. This allows high quality to be achieved at, hopefully, a low cost.

The modelling formalisms and tools used for that purpose are often based on the so-called "synchronous" paradigm [4] which assumes, for the sake of simplicity and abstraction, "zero-time" executions. This is the case for the very popular and widely used "Simulink/Stateflow"[1] [24] tools which are considered as *de facto* standards in many embedded and real-time domains such as the avionics and automotive industries. This is also the case for synchronous languages and for most event-based simulation systems such as the "SCADE" [12] simulator. Furthermore, this paradigm proves to be quite versatile, allowing the handling of traditional, periodically sampled control systems (also called *time-triggered* (TT) *systems* [18]) and discrete event ones (called *event-triggered* (ET) *systems*) as well as mixed time- and event-triggered systems.

In mixed systems an easy situation is often considered, where hard real-time tasks are time-triggered and events only drive soft real-time tasks that can be postponed to the idle time left by the time-triggered tasks. As we shall show in the current paper this restriction can be relaxed to address the mixed situation with greater generality. In particular, if the system requires the handling of urgent events then this implies a preemptive scheduling which can be difficult to reconcile with the synchronous paradigm. We propose methods for coping with such situations.

In practice execution times are not null and this may result in a distortion between the models with zero-time assumptions and their implementations. If these distortions are important, the confidence gained from the modelling can be lost. This question is an important one but it is difficult to answer since it is very domain-dependent:

- When time-triggered systems are derived from continuous processes, the solution is usually based on numerical analysis, stability and jitter considerations [17].
- In [6], the Esterel [5] programming language is coupled with the Kronos timed-automaton model-checker [31] 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 [27], 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.

[1] Trademarks of the Mathworks company.

In this paper, we consider a quite frequent situation where the zero-time model accounts for the functional specification while complex timing issues are abstracted into deadlines associated with tasks, *i.e.*, selected pieces of the functional specification.

In the presence of event-triggered tasks with short deadlines (urgent tasks) the only possible implementation has to be based on preemptive scheduling and this raises the problem of data integrity in inter-task communication. This problem has been intensively studied and solutions are either based on locking mechanisms or on lock-free ones. The former make the scheduling problem more complex and can raise the weel-known "priority inversion" problem [29]. This is why lock-free and wait-free methods have been proposed. Though the terminology has not been standardized, lock-free methods seem to refer to methods where a reader may loop attempting to get uncorrupted data [20, 19, 2], while, in wait-free or loop-free methods, no loop is required but more spare is needed to store the shared data [11, 16].

However, there has actually been little previous work in which both the properties of the scheduling, the inter-task communication mechanism and the semantical accuracy of the modelling are jointly taken into account and we address issues arising from such considerations.

This paper concentrates on how ET and TT tasks can be modelled in SIMULINK and SCADE and then scheduled for a mixed implementation. An inter-task communication framework is then presented within which the fidelity of the modelling process with respect to the running application can be proven given some reasonable assumptions about the nature of semantic equality.

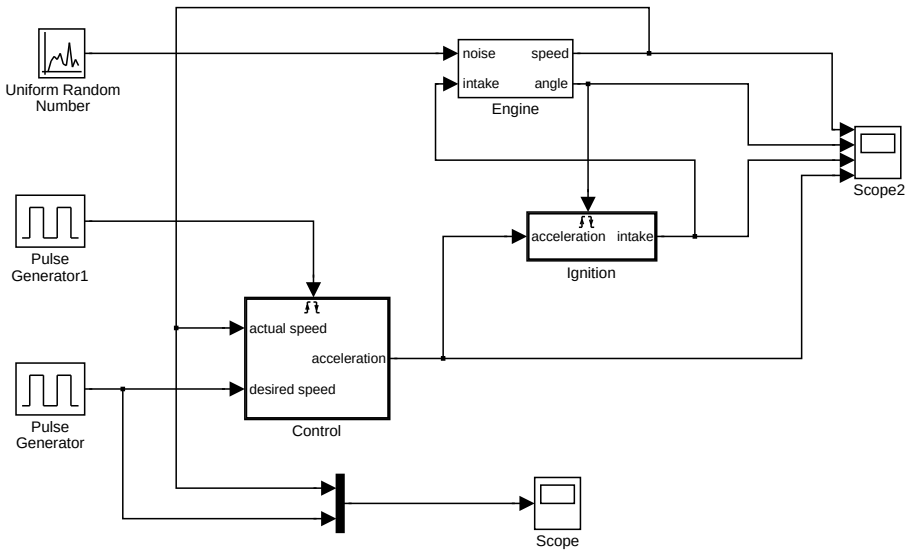

Figure 1: SIMULINK modelling of ET/TT tasks

2 Simulink and Scade modelling

It is a common opinion that the synchronous and time-triggered paradigms are closely related while event-triggered systems are associated with asynchrony. However, this point of view can be somewhat misleading. As a matter of fact, synchrony and asynchrony refer to whether concurrent activities share common time scales or not and this appears to be orthogonal to the different ways activities are triggered, either on time basis or on an event basis. This is why synchronous formalisms are equally able to model event-driven activities as well as time triggered ones and can also cope with mixed designs where the both are present. Fig. 1 shows a SIMULINK model which can be viewed conceptually as requiring both ET and TT tasks. The system being simulated is a cruise-control system for a car where the setpoint is the `desired_speed` input. There are three major components; the engine (`Engine`) which is basically an oscillator to which an impulse (ignition) must be given at the correct time, an ignition control unit (`Ignition`) which takes the required acceleration and generates this sequence of pulses, and a simple PID controller (`Control`). The `Ignition` task is triggered by the crank angle and is thus an ET process. The `Control` task, on the other hand, performs its computations at regular time intervals and is thus a TT process. Note that we envisage a system in which ET and TT tasks are mutually interdependent.

The input to our system is a correct SIMULINK model for which we assume that type and clock[2] inference have succeeded and that it has undergone translation into either SCADE by SIMULINK GATEWAY or into LUSTRE by S2L [9]. Figure 2 shows a typical SCADE translation.

To prove the feasibility of this system and to maintain properties of correctness and safeness we have to provide, firstly a characterisation of the scheduling which is possible for the models

[2]"Clock" refers to the timing information required for a synchronous implementation

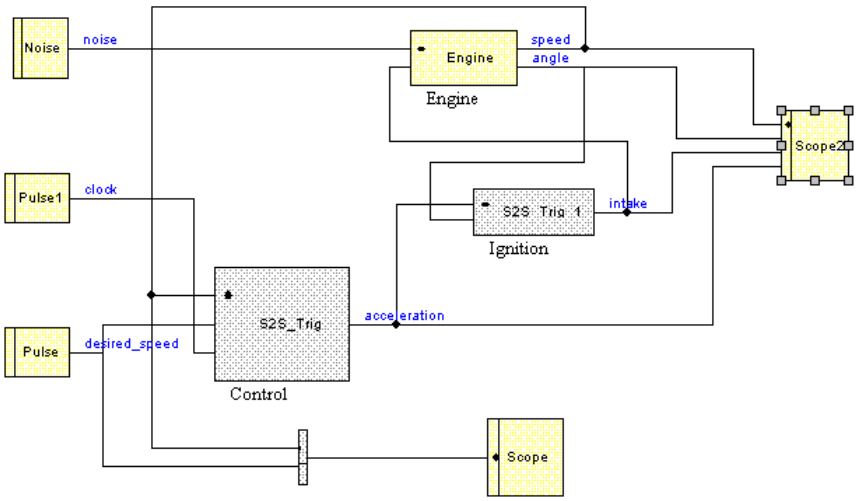

Figure 2: SIMULINK modelling of ET/TT tasks

we have specified, and secondly, that the semantics of the model is preserved under translation, i.e. that the simulation is a faithful representation of the execution.

3 Schedulability

3.1 Problem statement

As usual, time-triggered tasks are given a period and a deadline and we assume that only one instance of a given task can be active at the same time, which amounts to saying that deadlines are smaller than periods.

We assume that event-triggered tasks are sporadic with a predictable minimal interarrival time, which therefore plays the part of a minimum period and a deadline and, here also, deadlines are smaller than minimum periods. Furthermore, tasks are assumed to have known worst-case execution times. Thus, from an operational point of view, each task $\mathcal{T}_i$, be it time or event-triggered, has three associated parameters, its period or minimum period T_i, its deadline D_i and its WCET C_i.

3.2 Exploring the design space

Starting from the pionneering work of Liu and Layland [23], many approaches have been proposed for scheduling these kinds of tasks and for assessing their schedulability, among which we can cite [22, 21, 30, 3] and it is not easy to find one's way in this intricate landscape. Moreover, given our goal of semantic preservation, choices for the scheduling and for the communications are closely related. This is why exloring the design space is not easy. Some of the choices are:

- When only time-triggered tasks are considered, the static "table scheduling" approach of TTA [28] is appealing as it solves both the scheduling and the communication problem in an efficient way, by statically partionning long period tasks into subtasks that can fit within the gcd of the periods.
- When there are event-triggered tasks whose deadline is shorter than the execution time of another task, preemptive scheduling seems compulsory.
- In this case, fixed priority can be seen as the most "static" approach. It can be adopted without losing the benefits of a fully static approach. Furthermore, as we shall see in Section 4, fixed priority makes communication easier and less space-consuming.
- As said above, lock- and loop-based methods are likely to complicate the schedulability analysis. We therefore assume that we shall define wait-free communication methods and that our scheduling problem deals with independent tasks.

3.3 Fixed-priority preemptive scheduling

This leads us to adopt the deadline monotonic preemptive scheduling of [3]. Tasks are given fixed priorities $Prio_i$ in the reverse order of their deadlines:

$$D_i < D_j \Rightarrow Prio_j < Prio_i$$

In the setting of critical systems, schedulability tests are an important issue: deadlines must not be missed. Several schedulability tests have been proposed for this scheduling strategy. A recent and less pessimistic approach to fixed priority scheduling is to recursively compute worst-case response times without assuming any deadlines. These response times can then be compared with the deadlines [8]:

$$R_i \;=\; C_i + \sum_{j>i} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

where $i < j$ implies $Prio_i > Prio_j$. In this expression, $\left\lceil \frac{R_i}{T_j} \right\rceil$ computes the maximum number of times the higher-priority task $\mathcal{T}_j$ can preempt task $\mathcal{T}_i$ while it is running.

If $\forall_i : R_i \leq D_i$ then the system is schedulable. This equation implies an iterative solution:

$$r_i^{n+1} \;=\; C_i + \sum_{j>i} \left\lceil \frac{r_i^n}{T_j} \right\rceil C_j$$

which converges since the sum is a monotonically increasing function of n.

4 Functional semantics

Design in SIMULINK and SCADE can be seen as parallel designs where processes operate concurrently. In many cases, designers conceptualise the communication between these parallel processes under the "freshest value" semantic; this is often the case for control systems. In [11] a three buffer protocol is presented which ensures a wait-free communication protocol providing for this freshest value semantic. Furthermore, this protocol could be easily optimised using the techniques presented in [16] by taking advantage of the the fixed-priority scheme we have chosen.

However, one should be aware that this does not preserve the equivalence that the modelled behaviour equals the implemented behaviour, under some notion of semantic equality. The principal point of difference is that the ideal semantics assumes zero-time execution whereas the real-time semantics has to include a (bounded) computation time. This raises the problem of ensuring that in both cases computation proceeds with the same values. Preemption, however, can cause values in the real-time case to be delayed relative to the ideal case. For cases where this equivalence matters, we propose, in this section, a means of preserving it, on the basis of simple syntactic checks on the model structure and careful implementation of communications between processes.

4.1 Syntactic restrictions

Let us consider two tasks $\mathcal{T}_i$ and $\mathcal{T}_j$ such that $Prio_i > Prio_j$ and $\mathcal{T}_i$ computes a value $v_i = f_i(v_j)$, i.e. a function of a value produced by $\mathcal{T}_j$. Let e_i and e_j be the triggering events[3] associated with the tasks. Let us consider the situation, in the ideal "zero-time" model, where the nth occurrence

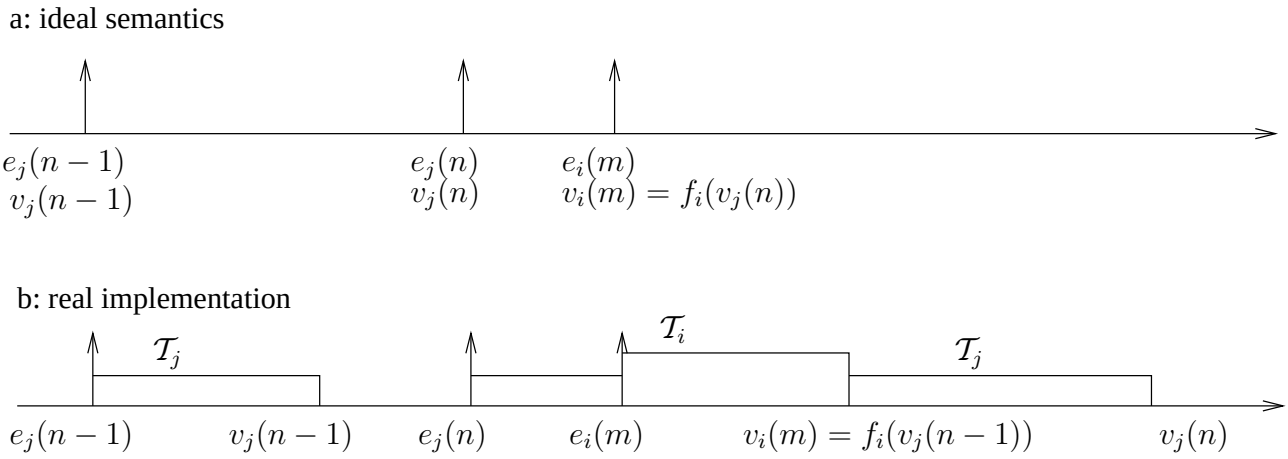

Figure 3: Example illustrating the need for syntactic restrictions

[3] We do not distinguish here between time- and event-triggered tasks. Thus any of these events can be produced by a clock.

of e_j takes place just before the mth occurrence of e_i. $\mathcal{T}_j$ instantly executes and produces $v_j(n)$. Then, $e_i(m)$ occurs and $\mathcal{T}_i$ executes and produces $v_i(m) = f_i(v_j(n))$. This situation is illustrated at Fig. 3-a.

In the real world, even if we provide communications which ensure the data integrity, it may be the case that $\mathcal{T}_i$ interrupts $\mathcal{T}_j$ before completion. As a result the $\mathcal{T}_j$ outputs may not be available for $\mathcal{T}_i$ computations. In this case, the result will be $v_i(m) = f_i(v_j(n-1))$ and there will be a difference between the ideal and real-time behaviours (cf. Fig. 3-b). A way to overcome this difficulty is to require that:

> *A higher priority task should not use values computed by an immediately preceding lower priority task.*

This is a syntactic restriction which amounts to checking that each communication from a low-priority task to an higher one be mediated by a unit delay triggered by the low-priority trigger. A SIMULINK trigger is called a clock in SCADE. Henceforth, we refer to these triggers as clocks.

Note that, conversely, this availability problem does not arise for communications from a high-priority task to a low-priority one because lower-priority tasks can always be sure that the higher-priority tasks have completed at the instant of starting computation. We thus do not require the same constraint for communications from higher to lower priority tasks.

4.2 Signalling and communication

Unfortunately, these syntactic restrictions are not sufficient for ensuring a sensible equivalence between ideal and real-time behaviours.

For instance, in a communication from high to low, it may be the case that the low-priority task starts reading, but is interrupted by the high-priority task which changes its results. When the lower priority task resumes reading, it will get incoherent data.

Even when communications are buffered as in the case of low to high communications, undesirable effects can take place, as illustrated by the following situation:

Consider three tasks, $\mathcal{T}_1$, $\mathcal{T}_2$ and $\mathcal{T}_3$ and corresponding triggering events e_1, e_2, e_3 such that $Prio_1 > Prio_2 > Prio_3$ and $\mathcal{T}_2$ uses some data v_3 from $\mathcal{T}_3$. Suppose that e_3 occurs followed by e_3 followed by e_1 followed by e_3 followed by e_2.

Let $v_3(n-2)$, $v_3(n-1)$ and $v_3(n)$ be the successive values produced by $\mathcal{T}_3$. Then, under the zero-time assumption, owing to our unit-delay restriction, $\mathcal{T}_2$ will use $v_3(n-1)$ for its computation. This situation is depicted in Fig. 4-a.

However, in the real-time framework, it can be the case that the preemption results in the execution order $\mathcal{T}_3$ followed by $\mathcal{T}_3$ followed by $\mathcal{T}_1$ followed by $\mathcal{T}_2$ followed by $\mathcal{T}_3$ which could happen under our fixed-priority preemption scheme. In this case, (cf. Fig. 4-b) $\mathcal{T}_2$ would use $v_3(n-2)$ for its computations because it starts executing before $\mathcal{T}_3$. This impairs the equivalence between ideal and real-time behaviours.

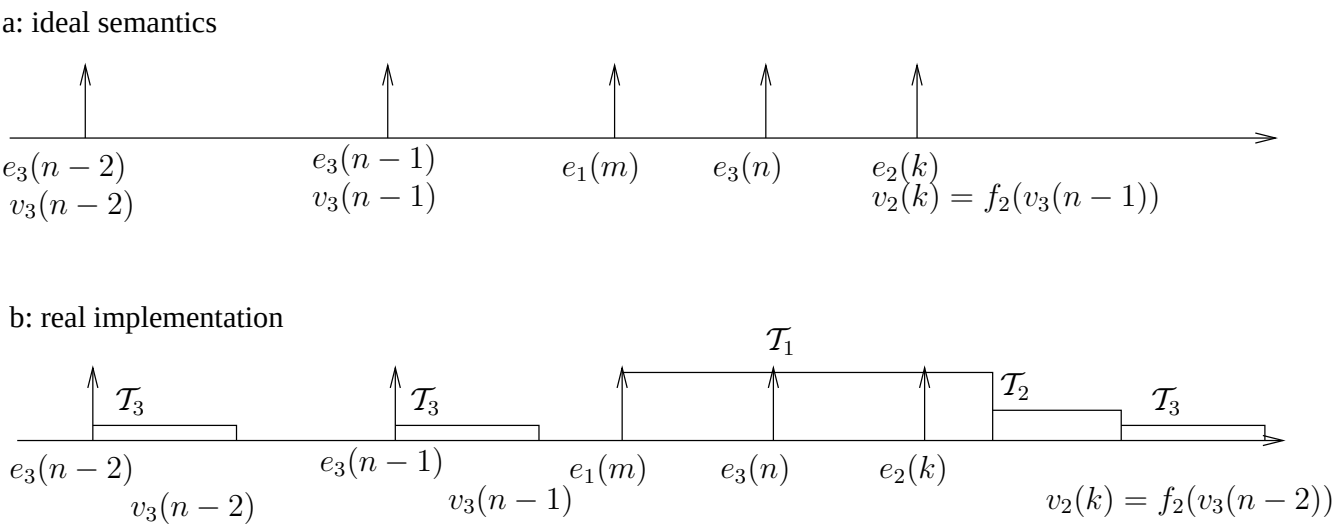

Figure 4: Example illustrating the need of distinguishing between event occurrence and task execution.

> *This shows that, if we want to keep in the implementation the same order of communication as in the model, the buffering mechanism should be controlled by the triggering event occurrences and not by the task executions.*

4.3 A communications scheme

We need to ensure that preemption by higher level tasks does not affect the relative clock timing for lower priority tasks. We assume here that $\mathcal{T}_i$ has a higher priority than $\mathcal{T}_j$.

Communications from $\mathcal{T}_j$ to $\mathcal{T}_i$

We institute a double buffering mechanism defined in Fig. 5. It is worth emphasising that it is the *occurrence* of e_j which toggles the buffers not the start of execution and that since these operations consist solely of toggling flags, it can be assumed they take no time. Note also that since we may need to disambiguate in the event of simultaneous occurrences of e_i and e_j the buffer toggling would have to be implemented by the scheduler rather than the tasks themselves.

Communications from i to j

Conversely, $\mathcal{T}_j$ cannot read from $\mathcal{T}_i$, because $\mathcal{T}_i$ can execute several times between the occurrence of $\mathcal{T}_j$ and its subsequent execution. This would require a lot of buffering. What we propose instead is that $\mathcal{T}_j$ informs $\mathcal{T}_i$ where to write its results. The communications mechanism is described in Fig. 6. Here also, the signalling mechanism is assumed to take no time.

Fig. 7 illustrates a typical low-to-high scenario. To make the figure more readable, each task is given its own time-line. Note, on the low-to-high part, how the e_j occurrences toggles the

Task $\mathcal{T}_j$ maintains a double buffer b_j:

- an "actual" buffer into which $\mathcal{T}_j$ writes results, and
- a "previous" buffer from which $\mathcal{T}_i$ reads results.

Algorithm:

- When e_j occurs, it instantaneously toggles the buffers. The "actual" buffer which has just been filled becomes the "previous" buffer and *vice versa*.
- When $\mathcal{T}_j$ executes it writes results to the "actual" buffer.
- When e_i occurs, it instantaneously stores the address of b_j's "previous" buffer at that instant.
- When $\mathcal{T}_i$ executes it reads values from the buffer whose address it stored when it occurred.

Figure 5: Communications scheme from low priority to high priority

buffers where $\mathcal{T}_j$ writes. Note also that the buffer where $\mathcal{T}_i(n)$ reads becomes the buffer where $\mathcal{T}_j(m)$ writes but, because of fixed priorities, no harm can result from this apparent conflict.

Fig. 8 illustrates a typical high-to-low scenario. Note that $\mathcal{T}_i(n+2)$ overwrites what was written by $\mathcal{T}_i(n+1)$. This is because no e_j has occurred between $e_i(n+1)$ and $e_i(n+2)$. Note also it may be the case that, when $e_j(m)$ occurs, a pending $\mathcal{T}_i(n)$ is about to execute. This pending $\mathcal{T}_i(n)$ will write in the "current" buffer, the address of which it stored (as a "next" buffer) when $e_i(n)$ occurred. This is not harmful as we know, because of the fixed priorities, that $\mathcal{T}_i$ is pending and has to complete execution before $\mathcal{T}_j$ can execute and read the "current" buffer.

Task $\mathcal{T}_j$ maintains a double buffer b_{ji} and a flag for each of the higher priority tasks $\mathcal{T}_i$ it needs values from:

- a "current" buffer from which $\mathcal{T}_j$ reads $\mathcal{T}_i$'s results, and
- a "next" buffer, where $\mathcal{T}_i$ writes its results, and

Algorithm:

- When e_i occurs it:
  - gets from b_{ji} the address of the "next" buffer at that instant, and
  - sets the flag.
- When $\mathcal{T}_i$ executes, it fills the "next" buffer whose address it stored when it occurred.
- When e_j occurs:
  - if the flag is set,
    - it toggles these buffers, and
    - clears the flag,
  - otherwise the buffers remain untoggled.
- When $\mathcal{T}_j$ executes, it get values from its "current" buffers.

Figure 6: Communications scheme from high priority to low priority

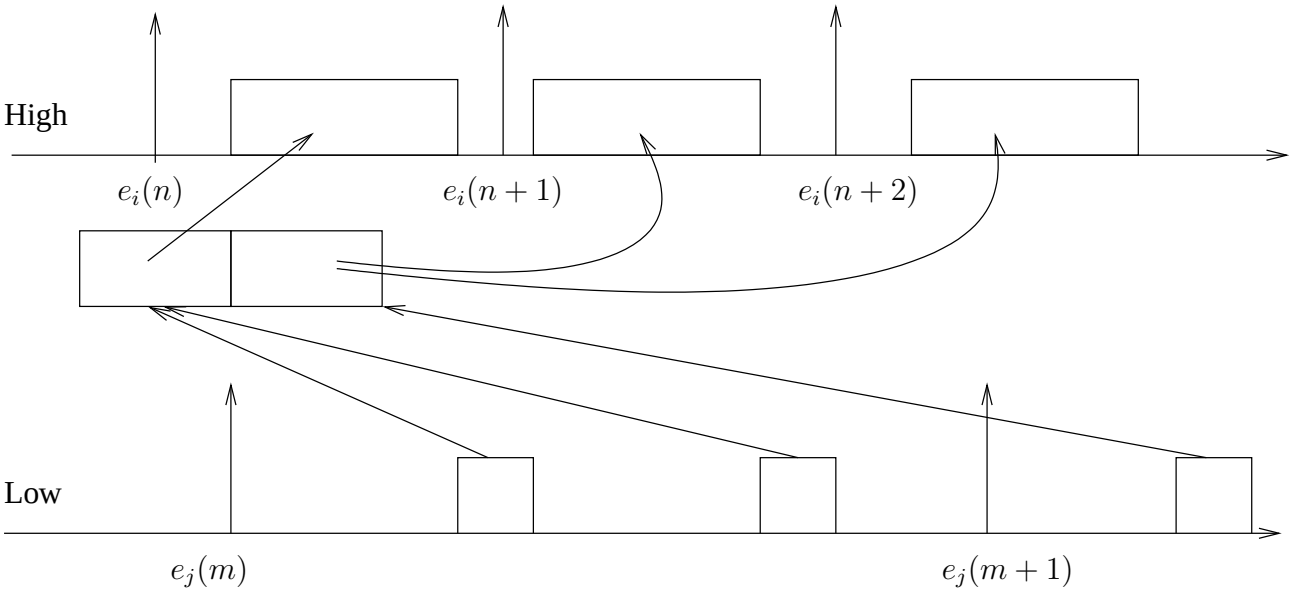

Figure 7: A typical low-to-high communication scenario

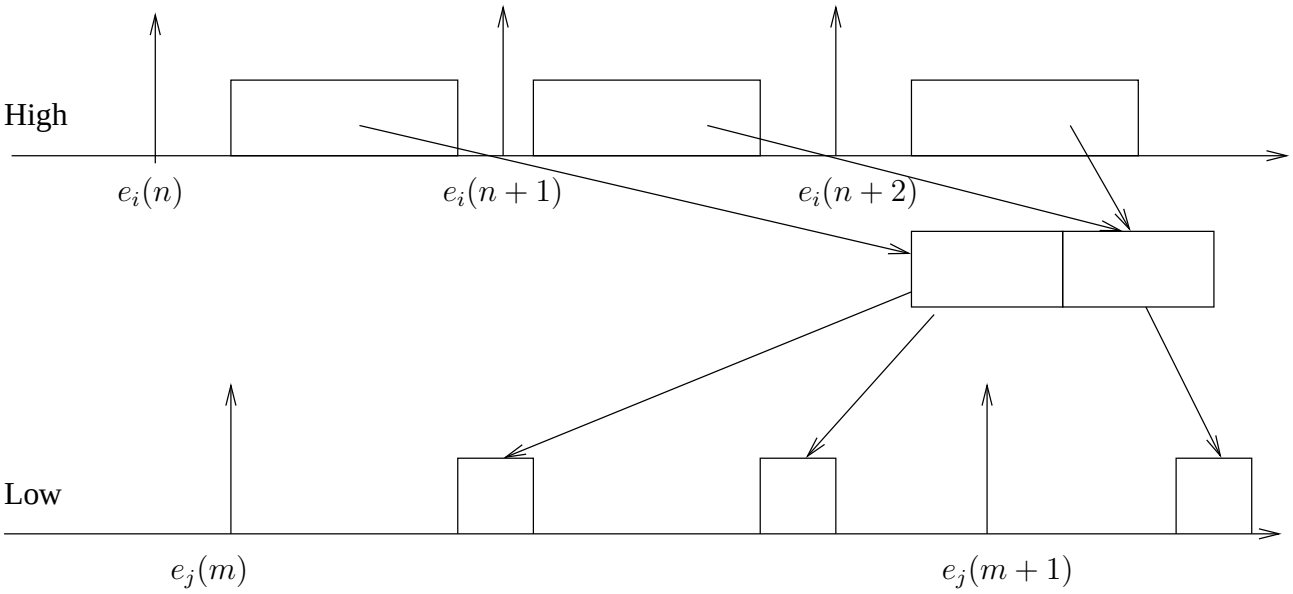

Figure 8: A typical high-to-low communication scenario

5 Formalisation and proof

This communication scheme has been formalised using the Caspi-Halbwachs framework [10] since this is the theoretical underpinning for SCADE and LUSTRE. It also matches a discrete time subset of SIMULINK. Moreover, a complete description of the scheme has been developed in LUSTRE [13] and model-checked using Lesar [14], the model-checker associated with Lustre.

5.1 Formalisation

5.1.1 Framework

There are several ways we could model these semantics, for instance, timed automata [1] or real-time temporal logics [15]. However, we propose modelling the system using the Caspi-Halbwachs framework [10] since this is the theoretical underpinning for SCADE and LUSTRE. It also matches quite well a discrete time subset of SIMULINK. In this framework, a flow is a pair of sequences (x, d) where $x : \mathbb{N} \rightarrow \mathbb{V}$ is a sequence of values and $d : \mathbb{N} \rightarrow \mathbb{T}$ is an increasing sequence of times (dates), with the interpretation: $x(n)$ takes its value at time $d(n)$.

The counter function

We can also associate with t the counter function $c : \mathbb{T} \rightarrow \mathbb{N}$, giving, at any time t, the index of the last $d(n)$ preceding t:

$$c(t) = sup\,\{n | d(n) \leq t\}$$

We then have the Galois connection properties:

$$\begin{array}{rcl} d \circ c & \leq & I \\ c \circ d & \geq & I \end{array}$$

where I is the identity function. Moreover, if we restrict ourselves to events which cannot occur more than once at a time, the latter gives:

$$c \circ d = I$$

which reads: the number of event occurrences at the time of the nth occurrence is exactly n.

The preceding value

We also have the preceding value which can be implemented in SIMULINK using the $1/z$ block (with inherited sample time) and in SCADE using the `fby` operator:

$$p(x) = x \circ (I - 1)$$

where $(I - 1) = \lambda n \cdot max(0, n - 1)$ acts as a unit delay. Thus,

$$\begin{array}{lcl} p(x)(0) & = & x(0) \\ p(x)(n+1) & = & x(n) \end{array}$$

Sampling and holding

Sampling and holding take place in SIMULINK's "triggered subsystems" and in SCADE's "activation conditions".

Holding takes place at the outputs of a subsystem: given an output flow (xs, ds) which runs at the pace ds of the subsystem, the external world sees a trajectory v such that:

$$v = x \circ c$$

which reads: the value of a flow at time t is the value at the index of the last occurrence of the corresponding event preceding t.

Sampling takes place at the inputs of a subsystem: given an input flow (x, d) and the sampling event ds, the result of the sampling is the flow (xs, ds) such that:

$$xs = x \circ c \circ ds$$

which is simply the value of the trajectory picked at the sampling instants. This can be rephrased as: the nth value of xs is the value taken by x at the last occurrence of x preceding the nth occurrence of ds.

These very simple primitives, together with usual mathematical functions allow us to provide a simple and accurate semantics of SIMULINK and SCADE diagrams.

5.1.2 Ideal semantics

We define the ideal semantics in terms of n tasks, each computing a flow. The computation of the flow in each task can use values from the other tasks. Priorities are in reverse order such that task 1 has the highest priority.

This arises naturally from viewing a SIMULINK computation as a set of parallel tasks which compute values and communicate results instantaneously. The prioritisation scheme corresponds roughly to that of the statically-ordered interactions which ensure the uniqueness of SIMULINK simulations.

STATEFLOW is more problematical since its behaviour is defined solely by the interpretation algorithm. However, we can choose to impose a set of syntactical checks upon models which allow us to view the simulation as having a synchronous semantics with a suitable prioritisation scheme. Models which cannot be made to conform have to be rejected by our system. For both SIMULINK and STATEFLOW we are also constrained to a particular set of simulation parameters

which allow us to view the simulation in this way. However, for a model intended to be implemented directly in an application there will be a sensible set of parameters, for example only discrete-time simulation need be considered.

Thus, under this synchronous semantics and according to the syntactic restrictions stated in Section 4.1 each task computes its value as a function of:

- the *current* value of the *preceding* value of tasks with lower priorities, and
- the *current* value of tasks with higher priorities.

Thus the system can be specified in terms of the following system of equations:

$$\begin{array}{lcl}
x_1 & = & f_1(p(x_1), \quad \ldots \quad p(x_j) \circ c_j \circ d_1, \quad \ldots \quad p(x_n) \circ c_n \circ d_1) \\
\vdots & & \\
x_i & = & f_i(x_1 \circ c_1 \circ d_i, \quad \ldots \quad p(x_i), \quad \ldots \quad p(x_n) \circ c_n \circ d_i) \\
\vdots & & \\
x_n & = & f_n(x_1 \circ c_1 \circ d_n, \quad \ldots \quad x_j \circ c_j \circ d_n, \quad \ldots \quad p(x_n))
\end{array}$$

5.1.3 Real-time semantics

The real-time, preemptive semantics is defined in similar terms except that each task is characterised by three sequences of times, d, db and de, the times at which computation is requested, begins and ends, respectively. It is assumed that each task samples its environment at the beginning of computations. This gives a similar set of equations as for the ideal case:

$$\begin{array}{lcl}
x_1 & = & f_1(p(x_1), \quad \ldots \quad p(x_{j1}), \quad \ldots \quad p(x_{n1})) \\
\vdots & & \\
x_i & = & f_i(x_{1i}, \quad \ldots \quad p(x_i), \quad \ldots \quad p(x_{ni})) \\
\vdots & & \\
x_n & = & f_n(x_{1n}, \quad \ldots \quad x_{jn}, \quad \ldots \quad p(x_n))
\end{array}$$

where, for $i < j$ (i.e. i has higher priority):

$$x_{ij}(n) \; = \; (x_i \circ (I+1) \circ ce_i \circ d_i \circ c_i \circ d_j \circ c_j \circ db_j)(n)$$

This is the value sent at the next execution of i following the last arrival of i preceding the last arrival of j preceding the time when j executes. Also:

$$x_{ji}(n) = (x_j \circ ce_j \circ d_j \circ c_j \circ d_i)(n)$$

This is the value stored at the last execution of j preceding the last arrival of j preceding the time when i occurs.

These equations embody the syntactic restrictions of Section 4.1 and the buffering mechanism proposed in Section 4.2.

5.1.4 Semantic equivalence

At this point, we can make more precise our notion of equivalence. What we require is:

> *Given the same sequences of external inputs, the ideal and real-time behaviours exhibit the same sequences of outputs.*

Clearly, inputs may not be acquired at the same instants and the value at an arbitrary instant may also differ. Neither will outputs be emitted at the same instants, but we know the limits of the temporal distortion between the ideal and real timings which are embodied in the deadlines used in the schedulability analysis. The question of whether the overall system can tolerate the resulting value and time distortions is domain and design dependent and will be left out of the scope of the present paper.

5.1.5 Real-time conditions

Before being able to prove the equivalence defined above, we need to formalise the operating conditions of the real-time system and in particular, formalise our fixed-priority scheduling policy.

The priority conditions assuming $i < j$ are shown in Fig. 9. These conditions can be derived from either the basic properties of flows or from the fixed priority scheduling properties. For instance, condition 1 is a direct result of our definition of the real-time flow and the resultant interpretation of the associated three sequences of times. Condition 2 is a result of the scheduling assumptions. Since we can assume we are working with a schedulable system then the property of deadline preservation ($\forall_i : R_i \leq D_i$) leads directly to this deadline condition. The fixed priority scheduling assumption has several consequences. Mostly, we are interested in the case where a higher priority task is pending but not executing when a lower priority task occurs. In this case preemption gives condition 3.(c). Note that 3.(c) is equivalent to the combination of 3.(a) and 3.(b) (i.e. that task i preempts task j) but from the point of view of the lower priority task rather than the higher priority one.

5.1.6 Proof of equivalence

For the purposes of proving the equivalence of the ideal and real semantics the communications mechanism described above would make this proof extremely complex. However, assuming that this mechanism works and is correct we can simply reduce it to the assumption on the real-time clocks that they are equivalent to instantaneous computation with respect to the temporal ordering of value emission events. Given this assumption the rest of the proof is relatively simple.

Communications from j to i

This is easy because as soon as i arrives, it knows that j will not execute before i completes execution itself. Thus, as a first approximation, it is sufficient to show that i reads the current value contained in the "previous" buffer where j stores its values. However, as mentioned previously,

1. Ends should follow beginnings:

$$d_i(n) \leq db_i(n) < de_i(n)$$

2. The scheduling conditions are such that deadlines are not missed:

$$(de_i \circ (I - 1))(n) < d_i(n)$$

3. Fixed-priority scheduling implies that:

   (a) If d_i takes place between the beginning and the end of a j computation, then so also do db_i and de_i take place in this interval:

$$\forall n,\ (ce_j \circ d_i)(n) < (cb_j \circ d_i)(n) \Rightarrow (ce_j \circ d_i)(n) = (ce_j \circ db_i)(n) = (ce_j \circ de_i)(n)$$

   (b) If d_i takes place between the request and the beginning of a j computation, then db_i and de_i also take place in this interval:

$$\forall n,\ (cb_j \circ d_i)(n) < (c_j \circ d_i)(n) \Rightarrow (cb_j \circ d_i)(n) = (cb_j \circ db_i)(n) = (cb_j \circ de_i)(n)$$

   (c) If d_j takes place between the request and the end of a i computation, then db_j does not take place before de_i:

$$\forall n,\ (ce_i \circ d_j)(n) < (c_i \circ d_j)(n) \Rightarrow (c_i \circ db_j)(n) = (ce_i \circ db_j)(n)$$

Figure 9: Priority conditions (i has higher priority than j)

we need to assume that this buffer is instantaneously toggled when j occurs and that the address of this buffer is instantaneously stored by i as soon as i occurs. Thus, given:

$$x_{ji}(n) = (x_j \circ ce_j \circ d_j \circ c_j \circ d_i)(n)$$

It is easy to prove that:

$$(ce_j \circ d_j \circ c_j \circ d_i)(n) = ((I-1) \circ c_j \circ d_i)(n)$$

or that:

$$(ce_j \circ d_j)(n) = (I-1)(n)$$

Actually, the number of j ends at the time of the nth j occurrence is obviously $n-1$.

Communications from i to j

Again, as stated previously, j cannot read from i, because i can execute several times between the occurrence of j and its subsequent execution, so we arrange for i to send values to j so that j can select the appropriate one.

As a first approximation, it is sufficient to ensure that i sends its value to a buffer from which j can read it. However, we need to assume that this buffer is instantaneously toggled when j occurs and that the address of this buffer is instantaneously stored by i as soon as i occurs. As before:

$$x_{ij}(n) = (x_i \circ (I+1) \circ ce_i \circ d_i \circ c_i \circ d_j \circ c_j \circ db_j)(n)$$

We need to prove:

$$((I+1) \circ ce_i \circ d_i \circ c_i \circ d_j \circ c_j \circ db_j)(n) = (c_i \circ d_j)(n)$$

This is easy since the number of j occurrences at the time of the nth j beginning is obviously n:

$$(c_j \circ db_j)(n) = I(n)$$

Also, the number of i ends at the time of the nth i occurrence is obviously $n-1$:

$$((I+1) \circ ce_i \circ d_i)(n) = I(n)$$

Availability from i to j

We also need to prove that this value is available, and this should come from the real-time conditions. We need to prove:

$$(de_i \circ (I+1) \circ ce_i \circ d_i \circ c_i \circ d_j \circ c_j \circ db_j)(n) \leq db_j(n)$$

According to condition 3.(c), we have to consider two cases:

1) No preemption of j by i occurs, i.e. when j occurs, i has executed:

$$(ce_i \circ d_j)(n) = (c_i \circ d_j)(n)$$

We then have:

$$\begin{aligned}
&de_i \circ (I+1) \circ ce_i \circ d_i \circ c_i \circ d_j \circ c_j \circ db_j(n) \\
&= de_i \circ (I+1) \circ ce_i \circ d_i \circ c_i \circ d_j(n) \\
&= de_i \circ (I+1) \circ ce_i \circ d_i \circ ce_i \circ d_j(n) \\
&= de_i \circ ce_i \circ d_j(n) \\
&\leq d_j(n) \\
&< db_j(n)
\end{aligned}$$

2) Preemption of j by i occurs, possibly multiple times. However, we can assume that when j begins execution no i computations are pending:

$$(c_i \circ db_j)(n) = (ce_i \circ db_j)(n)$$

We then have:

$$\begin{aligned}
&de_i \circ (I+1) \circ ce_i \circ d_i \circ c_i \circ d_j \circ c_j \circ db_j(n) \\
&= de_i \circ (I+1) \circ ce_i \circ d_i \circ c_i \circ d_j(n) \\
&= de_i \circ c_i \circ d_j(n) \\
&\leq de_i \circ c_i \circ db_j(n) \\
&= de_i \circ ce_i \circ db_j(n) \\
&\leq db_j(n)
\end{aligned}$$

5.2 Model-checking

Surprisingly, this communication scheme can be model-checked, *i.e.*, automatically proved, using LUSTRE and LESAR . This is achieved by:

1. Describing the communication scheme in LUSTRE.
2. Describing the driving events in LUSTRE.
3. Comparing the ideal and real-time behaviours in LUSTRE.
4. Running the LESAR proof.

In doing this, several difficulties have to be overcome:

1. LESAR can only reason about boolean data-types. But our communication scheme is assumed to be able to convey any data-type and, if we replace these data-types by booleans, it may be the case that our scheme looks correct because the values it conveys cannot take more than two values. Yet, using the technique of uninterpreted functions [7], we

```
node lowtohighbuf(fromev, toev,  fromact: bool; fromval: bool^n)
returns (toval: bool^n);
var even,  odd: bool^n;
    bitfrom, bitto: bool;
let
  bitfrom = false -> if fromev then not pre bitfrom
                     else pre bitfrom;
  bitto = false -> if toev then not bitfrom
                   else pre bitto;
  even = if fromact and bitfrom then fromval
         else (init -> pre even);
  odd = if fromact and not bitfrom then fromval
        else (init -> pre odd);
  toval = if bitto then even
          else odd;
tel
```

Figure 10: A low-to-high double buffer

can replace the unknown data-type with a fixed number of distinct values and run model-checking. This number is the maximum number m of distinct values that can be present in the systems at the same time. To implement this idea, we replace the data-type with a n-vector of booleans, such that 2^n exceeds m.

2. LUSTRE is a synchronous untimed formalism and we need to express timing relations and schedulability. Yet, we can do it by running a logical abstraction which over approximate these timing relations. Provided this over-approximation is not too large, proofs can succeed.

5.2.1 The communication scheme in LUSTRE

Figure 10 shows the LUSTRE modelling of a low-to-high buffer.

In this figure,the booleans `fromev, toev` tell when the triggering events of the writing and reading tasks occur. The boolean `fromact` tells when the input task writes the buffer. The boolean vector verb-fromval- is the value that is written when `fromact` occurs. Finally, `toval` is the value the buffer makes available to the reading task.[4]

`even` and `odd` are the two buffers and `bitfrom` is the bit telling which one is the current one. `bitto` is the address of the previous buffer. As a matter of fact, this bit should be encapsulated in the reading tasks, as there should be different instances of this bit for each reading task. Since we consider here only one reading task, we encapsulated it in the buffer itself.

[4]In the LUSTRE data-flow framework, it suffices that the buffer displays the value. The reading operation will be performed by the receiving task.

```
node hightolowbuf(fromev, toev,  fromact: bool; fromval: bool^n)
returns (toval: bool^n);
var even,  odd: bool^n;
    bitfrom, bitto: bool;
let
  bitfrom = false -> if fromev and ((pre bitfrom) = pre bitto)
                     then not pre bitfrom
                     else pre bitfrom;
  bitto = false -> if toev then bitfrom
                   else pre bitto;
  even = if fromact and bitfrom then fromval
         else (init -> pre even);
  odd = if fromact and not bitfrom then fromval
        else (init -> pre odd);
  toval = if bitto then even
          else odd;
tel
```

Figure 11: A high-to-low double buffer

The equation defining `bitfrom` shows the buffer toggling when `fromev` occurs. The equation defining `bitto` shows how the `toev` occurrence samples the previous buffer. The equations defining `even` and `odd` show how these buffers are updated when the `fromact` takes place, according to which one is the current buffer. Finally, `toval` shows how these buffers are displayed according to which one is the previous buffer.

Figure 11 shows a high-to-low buffer. The interface and local variables are the same as in the low-to-high case. Not yet that now, the `bitto` is a private bit of the buffer as there is one such buffer for each communicating couple of tasks.

Note in the equation defining `bitfrom` how the test `(pre bitfrom) = pre bitto` stands for the flag allowing the buffer toggling. Note also that, when `toev` occurs, `bitto` is set to the next buffer (thus current and next become the same).

5.2.2 Describing the events in LUSTRE

Figure 12 shows how we model fixed-priority, preemptive scheduling in LUSTRE. It also shows how we model hypotheses in LUSTRE by defining a boolean expression (in general called `prop`) that should stay true as long as the hypothesis holds.

Then, node `cyclic` says that the triggering event, the beginning and the end of the corresponding task occur cyclically. This expresses that the schedulability conditions are met and that dead-lines are satisfied. Here, `after(s1, s2)` says that `s2` always occurs after `s1` and `forgetfirst(s1)` eliminates the first occurrence of `s1`.

Finally, `priority` says that task1 has higher priority than task 2, in that both are cyclic and that task 2 can neither begin nor end between the occurrence of task 1 triggering event and the

```
include "utils.lus";

-- s event occurrence
-- sb begin execution
-- se end of execution

node cyclic(s, sb, se: bool) returns (prop: bool);
let
  prop = after(s,  sb) and
    after(sb,  se) and
      after(se,  forgetfirst(s));
tel

-- s1 has higher priority than s2

node priority (s1, sb1, se1, s2, sb2, se2: bool)
returns (prop: bool);
let
  prop = cyclic(s1, sb1, se1) and
    cyclic(s2, sb2, se2) and
      neverbetween(s1, se1, sb2) and
        neverbetween(s1, se1, se2);
tel
```

Figure 12: Fixed priority preemptive scheduling in LUSTRE

```
node lthverif(val: bool^n; s1, sb1, se1, s2, sb2, se2: bool)
returns(prop: bool);
var ideal1, ideal2: bool^n;
let
  assert priority(s1, sb1, se1, s2, sb2, se2);
  ideal2 = if s2 then val
           else (init -> pre ideal2);

  ideal1 = if s1 then current ((init when s2) -> pre (ideal2 when s2))
           else (init -> pre ideal1);

  prop = if sb1 then vecteq(ideal1, lowtohighbuf(s2, s1, se2, ideal2))
          else true;

tel
```

Figure 13: The low-to-high proof in LUSTRE

end of its execution.

Undefined nodes are drawn from the included `utils.lus` library. For the sake of completeness, this library is given in appendix A.

5.2.3 The LESAR proofs

The low-to-high case Figure 13 shows the proof node of the low-to-high communication scheme. Here `val` is an arbitrary flow of values, `s1, sb1, se1, s2, sb2, se2` are arbitrary flows of events.

`ideal2` is the `val` flow sampled by the triggering event `s2` and figures out the value that would be transmitted in the ideal semantic world. Similarly, `ideal1` is the values that would be read by task 1 in the ideal world. Note here the presence of the unit delay `pre` on the clock of the writing task.

Finally `prop` computes the truth value of the property according to which, when task 1 reads the buffer, it gets exactly the value which it would have received in the ideal world.

In the proof results given in figure 14, LESAR automatically show that, under the assumptions given in the `assert` clause, the `prop` defined in this node stays always true.

The high-to-low case Figures 15 and 16 display similarly the high-to-low case. Note that, here, task 1 is the writing task and task 2 the reading one and that there is no unit delay in the ideal communication pattern.

```
arve# lesar verif.lus lthverif -v -states 100000
--Pollux Version 1.6
start normalisation ... done
start minimal network generation ..... done (226 -> 165 nodes)
building bdds ... 64 (on 64)

computing relevant statevars ... done (39 on 39)
DONE => 99529 states 367161 transitions

=>total bdd memory : 614431 nodes (~ 62403.15 K)
TRUE PROPERTY
```

Figure 14: The low-to-high proof in LESAR

```
node htlverif(val: bool^n; s1, sb1, se1, s2, sb2, se2: bool)
returns(prop: bool);
var ideal1, ideal2: bool^n;
let
  assert priority(s1, sb1, se1, s2, sb2, se2);
  ideal1 = if s1 then val
           else (init -> pre ideal1);

  ideal2 = if s2 then ideal1
           else (init -> pre ideal2);

  prop = if sb2 then vecteq(ideal2, hightolowbuf(s1, s2, se1, ideal1))
         else true;

tel
```

Figure 15: The high-to-low proof in LUSTRE

```
arve# lesar verif.lus htlverif -v -diag -states 100000
--Pollux Version 1.6
start normalisation ... done
start minimal network generation .... done (201 -> 148 nodes)
building bdds ... 57 (on 57)

computing relevant statevars ... done (32 on 32)
DONE => 22489 states 88105 transitions

=>total bdd memory : 106880 nodes (~ 10855.00 K)
TRUE PROPERTY
root@arve:/home/caspi/RISE/ETTT/LUSTRE#
```

Figure 16: The high-to-low proof in LESAR

6 Conclusions

We have outlined a mixed event-triggered and time-triggered system. Firstly, a method of modelling such systems using SIMULINK and translating through SIMULINK GATEWAY to SCADE was described. This would allow the verification of important properties by the SCADE verification tool before implementation on the target architecture.

The two most important properties of this system are schedulability and correctness of the modelling with respect to the implementation. For schedulability, we applied a simple deadline monotonic scheme which has a degree of optimality, is easily computed and matches our communications mechanisms. For functional correctness we proposed the use of unit delays to overcome problems of temporal ordering in preemptive systems and a suitable double buffering communication scheme. The originality of this double buffering technique lies in the dissociation between the instants when the buffers are chosen and toggled and the instants when they are used for proper communication.

With suitable generalisation and making some minimal assumptions about the timing of communications we were able to prove the correctness of event orderings. There has been little work in the past on systems which address both of these issues together. However, using the properties of the scheduling algorithm we were able to derive a set of semantical constraints within our framework which we were then able to use in the correctness proof. This is the novel aspect to our work. It is difficult to say how general the technique is, or how feasible it would be to automate some of the reasoning we have used, but for the system we have described here it works very well.

We are still obliged, however, to find adequate RTOS concepts and tools supporting both our scheduling policy and signalling and communication scheme. Depending upon the application domains we investigate, compliance with standards may be required. It is unlikely that a standard exists which can directly support our methods, for instance the combination of the OSEK standard [26] and its time-triggered equivalent OSEKTIME [25] for the automotive industry assumes non-preemption of time-triggered tasks by event-triggered ones, so it may be necessary to propose the evolution of such standards to meet our requirements.

Future work in this direction could also include studying whether the functional semantics can be preserved under different scheduling schemes such as incorporating inter-task dependencies, period less than deadline or tasks with variable run-times.

References

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994. 5.1.1

[2] J Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. In *Proceedings of the 16th Real-Time Systems Symposium*, pages 28–37. IEEE Computer Society, 1995. 1

[3] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atalanta, 1991. 3.3

[4] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceeding of the IEEE*, 79(9):1270–1282, September 1991. 1

[5] G. Berry and G. Gonthier. The ESTEREL synchronous programming language, design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992. 1

[6] V. Bertin, E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine. Taxys = Esterel + Kronos. A tool for verifying real-time properties of embedded. In *Conference on Decision and Control, CDC'01*. IEEE Control Systems Society, December 2001. 1

[7] Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In David L. Dill, editor, *Conference on Computer-Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 68–80. Springer-Verlag, 1994. Stanford, California, June 21–23, 1994. 1

[8] A. Burns, G. Bernat, and I. Broster. A probabilistic framework for schedulability analysis. In R Alur and I Lee, editors, *Proceedings of the Third International Embedded Software Conference, EMSOFT*, number LNCS 2855 in Lecture Notes in Computer Science, pages 1–15, 2003. 3.3

[9] P. Caspi, A. Curic, A. Maignan, C. Sofronis, and S. Tripakis. Translating discrete-time simulink to lustre. In R. Alur and I. Lee, editors, *EMSOFT'03*, Lecture Notes in Computer Science. Springer Verlag, 2003. 2

[10] P. Caspi and N. Halbwachs. A Functional Model for Describing and Reasoning About Time Behaviour of Computing Systems. *Acta Informatica*, 22:595–627, 1986. 5, 5.1.1

[11] J. Chen and A. Burns. Loop-free asynchronous data sharing in multiprocessor real-time systems based on timing properties. In *Proceedings of the Real-Time Computing Systems and Applications Conference*, pages 236–246, 1999. 1, 4

[12] Esterel Technologies, Inc. *SCADE Language - Reference Manual 2.1*. 1

[13] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991. 5

[14] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag. 5

[15] T. Henzinger. It's about time: Real-time logics reviewed. In *CONCUR '98: Ninth International Conference on Concurrency Theory*, number 1466 in LNCS, pages 439–454, 1998. 5.1.1

[16] H. Huang, P. Pillai, and K. G. Shin. Improving wait-free algorithms for interprocess communication in embedded real-time systems. In *USENIX 2002 Annual Technical Conference*, pages 303–316. http://www.usenix.org/publications/library/proceedings/, 2002. 1, 4

[17] K.J.Åström and B.Wittenmark. *Computer Controlled Systems*. Prentice-Hall, 1984. 1

[18] H. Kopetz. *Real-time Systems - Design Principles for Distributed Embedded Applications*. Kluwer Academic, 1997. 1

[19] H. Kopetz and J. Reisinger. Nbw: A non- blocking write protocol for task communication in real-time systems. In *Proceedings of the 14th Real-Time System Symposium*, pages 131–137. IEEE Computer Society, 1993. 1

[20] L. Lamport. Concurrent reading and writing. *Communications of ACM*, 20(11):806–811, 1977. 1

[21] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced Aperiodic Responsiveness in Hard Real-Time Environments. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 261–270, San Jose, CA, December 1987. 3.2

[22] J.Y.T. Leung and J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks. *Performance Evaluation*, 2(4):237–250, December 1982. 3.2

[23] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *JACM*, 20(1):46–61, 1973. 3.2

[24] The Mathworks Inc. *Simulink 5 Reference Manual*. 1

[25] OSEK Group, `http://www.osek-vdx.org`. *OSEK/VDX Time-Triggered Operating System Specification 1.0*, version 1.0 edition, July 2001. 6

[26] OSEK Group, `http://www.osek-vdx.org`. *OSEK/VDX Operating System Specification 1.0*, version 2.2.1 edition, January 2003. 6

[27] M. A.A. Sanvido, A. Ghosal, and T. A. Henzinger. xgiotto Language Report. Technical Report UCB/CSD-3-1261, Computer Science Division (EECS) University of California, July 2003. 1

[28] C. Scheidler, G. Heiner, R. Sasse, E. Fuchs, H. Kopetz, and C. Temple. Time-Triggered Architecture (TTA). In *Proceedings EMMSEC'97*, Florence, Italy, Nov 1997. 3.2

[29] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990. 1

[30] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems*, 1(1):27–60, 1989. 3.2

[31] S. Yovine. Kronos: A verification tool for real-time systems. *International Journal of Software Tools for Technology Transfer*, 1(1/2):123–133, October 1997. 1

A The `utils.lus` library

```
-- event library
-- event are assumed to be separated

node separated(s: bool) returns (prop:  bool);
let
  prop = (true -> pre s) => not s;
tel

-- after

node after(s1,  s2 : bool) returns (prop :  bool);
var todo : bool;
let
  assert separated(s1);
  assert separated(s2);
  todo = if s1 then true
         else if s2 then false
              else (false -> pre todo);
  prop = s2 => (false -> pre todo);
tel

-- forget the first occurrence of s1

node forgetfirst(s1: bool) returns (s2: bool);
var done: bool;
let
  assert separated(s1);
  done = if s1 then true
         else (false-> pre done);
  s2 = s1 and (false-> pre done);
tel

-- never between

node neverbetween (s1, s2, s3: bool) returns (prop: bool);
var now: bool;
let now = if s1 then false
              else if s2 then true
                   else (true -> pre now);
  prop = s3 => now;
tel
```