# Multitask Implementation of Synchronous Reactive Models with Earliest Deadline First Scheduling

Zaid Al-bayati and Haibo Zeng
McGill University, Canada
zaid.al-bayati@mail.mcgill.ca, haibo.zeng@mcgill.ca

Marco Di Natale
Scuola Superiore Sant'Anna, Italy
marco@sssup.it

Zonghua Gu
Zhejiang University, China
zgu@zju.edu.cn

***Abstract*—Model-based development of embedded systems enables early verification of functionality. Software implementations can be automatically generated from models, reducing the possibility of injected errors upon condition that the generated code is correct, efficient, and schedulable on the selected platform. When multitask implementations are generated for multirate systems, a time-feasible (schedulable) implementation may be possible only with additional functional delays on selected communication links. These delays may degrade the performance of feedback controls and require additional memory for buffering the channel state. In this paper, we present a branch-and-bound procedure and a heuristic algorithm to minimize the use of functional delays in the synthesis of model implementations on platforms scheduled by EDF (Earliest Deadline First). The proposed heuristic is shown to find close-to-optimal solutions, while executing much faster than (possibly exhaustive) branch-and-bound.**

## I. Introduction

Model-based design is emerging as a solution for handling the increasing complexity of embedded systems thanks to its capability for advanced verification and validation. In model-based design, the functionality is specified according to a language based on a formal model of computation. Synchronous Reactive (SR) models are widely used for control-dominated applications. They have been developed since the early 80s, including Esterel [4], Lustre [16], SIGNAL [17], and the Simulink graphical language [1]. They are popular today in many application domains, such as the avionics and automotive industries.

After the functional model is validated/verified, a software implementation on the selected platform is automatically generated using methods and tools that guarantee the preservation of semantic properties of interest. This step is of high practical relevance, since the best selection of the platform and model implementation may bring significant cost savings.

Control systems are multi-rate (composed of cooperating functions executing at different periods). In such cases, multitask implementations are preferable, because they allow higher resource utilizations and improve schedulability, compared to their single-task counterparts. However, most commercial tools offer limited options and very little user control for code generation. The development of functional SR models and languages has not been matched by a similar effort on languages, methods, and tools for the definition of the platform and the selection of the best mapping and implementation as a set of software tasks.

### A. Related Work

Esterel or Lustre models are typically implemented as a single executable that runs according to an event server model. Reactions to events are decomposed into atomic actions that are partially ordered by the causality analysis of the program. The scheduling is generated at compile time trying to exploit the partial causality order of functions and the generated code executes without the need of an operating system [4], [14], [25]. The main concern is to check that the synchronous assumption holds. That is, ensuring the longest chain of reactions to any event is completed within the system base period (the greatest common divisor of all the periods in the system). In the generation of code from Simulink models two options are available: a single-task (executing at the base period) or a fixed-priority multitask implementation, in which one task is generated for each period in the model, and tasks are scheduled by Rate Monotonic.

The general conditions for a semantics-preserving implementation of communications among synchronous nodes on a single-core platform are discussed in [8], where a communication mechanism (a customized wait-free protocol) is proposed. The optimization of the buffer size for flow-preserving communications is discussed in [13], [21], [24]. The extension of the communication mechanisms to model implementations on multicore platforms is discussed in [26]. The implementation of Esterel/Lustre on asynchronous and distributed architectures has been discussed in a number of papers. Techniques for generating semantic-preserving implementations of synchronous models on TTA are presented in [7]. Methods for desynchronization in distributed implementations are discussed in [9], [20]. A more general approach consists of an intermediate mapping of synchronous models into Kahn Process Networks [6], for which a correct implementation in an unsynchronized architecture platform can be found more easily [22] (very likely at the price of additional overhead and pessimism in the time performance).

Recently, a new synchronous language, Prelude [15] [19], has been proposed. Prelude is not only a functional modeling language, but includes rules and operators for the selection of a mapping onto platforms with Earliest Deadline First (EDF) scheduling, including symmetric multicore architectures [11]. The enforcement of the partial order of execution that is required by the SR model semantics is obtained in Prelude

by a deadline modification algorithm. Communication among nodes executing at different rates is realized using the protocol and data structures defined in [21] (an extension of [8] which is optimal with respect to the number of required buffers for a system containing only periodic nodes).

In general, in a correct task implementation, the execution order of the code implementing the node/block reactions must be consistent with their partial order in the model. In addition, the signal values communicated among nodes in the logical time execution must be preserved in the real-time task execution, regardless of possible preemptions or variable execution times.

In some cases, *a correct task implementation is not schedulable unless the designer modifies the model by adding communication delays on selected links with performance and memory costs*. The issue can be formalized as a *design optimization problem*: finding the schedulable implementation that requires the addition of the minimum number of functional delays, or, as better stated, the delays with minimum performance and memory penalty.

The problem of finding such an optimal definition of a multitask implementation for multirate Simulink models (scheduled with fixed priority) is discussed in [12] where a branch-and-bound algorithm is presented. Later in [13], an MILP (Mixed Integer Linear Programming) formulation is provided. A formulation for an MILP solution of an extended version of the problem (including multiple delays) considering both fixed priority and EDF mappings is presented in [18], but the paper does not present any application for the proposed formulation, which is not practical for problems of realistic size. Besides, it imposes the same relative deadlines for all instances of the same task, which can lead to sub-optimal solutions.

### B. Our Contributions

In this paper, we consider the problem of mapping a multirate synchronous system onto a single-core platform scheduled by EDF, showing how the definition of deadlines tighter than periods and appropriate wait-free communication mechanisms can be alternatively selected to preserve communications flows. Deadline restrictions come at no memory price, but may endanger system schedulability. Buffering using wait-free communication mechanisms may allow looser deadlines at the price of additional memory requirements and degraded control performance. We propose an algorithm that searches for efficient (and possibly optimal) feasible solutions, where an efficient solution is one that minimize the weighted sum of the functional delays. To evaluate our approach, we apply it to randomly generated task sets as well as an industrial case study.

The rest of the paper is organized as follows. Section II provides an introduction to synchronous reactive models. Section III discusses the options of implementing synchronous reactive models and highlights the tradeoff between the system schedulability and control algorithm performance. The algorithms to minimize the functional delays in the implementation are detailed in Section IV. In Section V, the experimental results on random task sets and the industrial case study are presented. The paper is concluded in Section VI.

## II. Description of the Functional Model

A Synchronous Reactive model of computation is represented by a *Directed Acyclic Graph* (DAG) $\mathcal{G} = \{\mathcal{N}, \mathcal{E}\}$, where $\mathcal{N}$ is the set of nodes (in the terminology of Lustre and Prelude, or blocks in Simulink), and $\mathcal{E}$ is the set of edges or links between the nodes. From the architecture point of view, we assume implementation on a single processor where concurrent tasks are scheduled by dynamic priority (EDF).

$\mathcal{N} = \{N_1, \ldots, N_{|\mathcal{N}|}\}$ is the set of *functional nodes*. Each node can be a Moore or Mealy-type behavior. Nodes can be arranged in a hierarchy in which a node consists of a network of children nodes until a level of atomic or leaf nodes. In this work, we restrict ourselves to *nodes triggered by periodic events*, therefore each leaf (or atomic) node $N_i$ is characterized by a period $T_i$. Node $N_i$ has one or more *input ports* and *output ports*. Input ports carry a set of signals with the uniform *sampling period* $T_i$. The signals are processed by the node and the result of the computation is a set of signal with the same rate, produced on the output ports.

$\mathcal{E} = \{E_1, \ldots, E_{|\mathcal{E}|}\}$ is the set of *links*. A link $E_i = (N_h, N_k)$ connects the output port of functional node $N_h$ (the source node) to an input port of node $N_k$ (the sink).

In Prelude, an *offset* can be associated with the execution of periodic nodes. Composition of atomic nodes can be obtained by passing signals (signal values). The communication may be associated with rate transition operators, such as the oversampling, undersampling, and phase offset operators (the details can be found in [15]). The definition of the communication and the partial order of execution apply to the instances of the node executed in the hyperperiod.

A fundamental part of the model semantics is the rules dictating the evaluation order of the nodes. Any Mealy type node for which the output update function is dependent on the inputs (i.e., any node with direct *feedthrough*) cannot execute until the node driving its input has executed. Other nodes (of Moore type) set their outputs based on the node state only (input values acquired in previous time steps and/or initial conditions specified as a node parameter), and are not of type feedthrough. The set of topological dependencies implied by the direct feedthrough defines a partial order of execution among nodes. The partial order must be accounted for in the simulation and in the run-time execution of the model.

If two nodes $N_i$ and $N_j$ are in an input-output relationship (the output of $N_j$ depends on its input coming from one of the outputs of $N_i$, and $N_j$ is of type feedthrough), then the communication link is associated with a precedence constraint between them, denoted by $N_i \rightarrow N_j$. In case $N_j$ is not of type feedthrough, then the link has a *unit delay*, as indicated by $N_i \stackrel{-1}{\rightarrow} N_j$.

Let $N_i(k)$ represent the $k$-th occurrence of node $N_i$, then a sequence of activation times $r_i(k)$ is associated to $N_i$. Given

time $t \geq 0$, we define $n_i(t)$ to be the number of times that $N_i$ has been activated before or at $t$.

In case of a link $N_i \rightarrow N_j$, if $i_j(k)$ denotes the input of the $k$-th occurrence of $b_j$, then the SR semantics specify that this input is equal to the output of the last occurrence of $b_i$ that is no later than the $k$-th occurrence of $b_j$, i.e.,

$$i_j(k) = o_i(m), \text{ where } m = n_i(r_j(k)). \quad (1)$$

The timeline on the bottom of Figure 1 illustrates the execution of a pair of nodes with SR semantics. The horizontal axis represents time. The vertical arrows capture the time instants when the nodes are activated and compute their outputs from the input values. In the figure, it is $i_j(k) = o_i(m)$ and $i_j(k+1) = o_i(m+2)$.

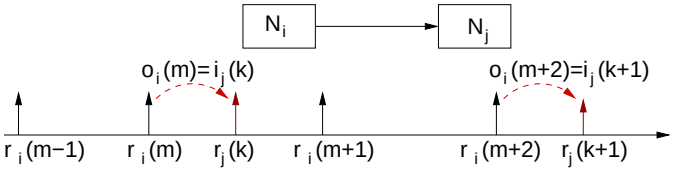


Fig. 1. Input/output relation with no delay on the communication link.

On the other hand, if $N_i \overset{-1}{\rightarrow} N_j$, then the previous output value is read, that is,

$$i_j(k) = o_i(m-1), \text{ where } m = n_i(r_j(k)). \quad (2)$$

Figure 2 shows the effect of the added delay on the input/output relation.

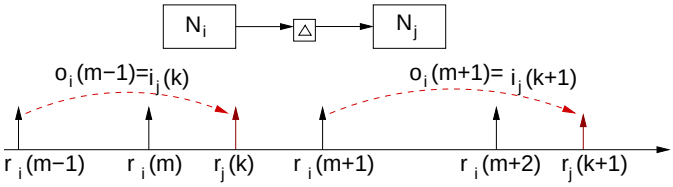


Fig. 2. Input/output relation with unit delay on the communication link.

A cyclic dependency among nodes where output values are instantaneously produced based on the inputs (all nodes in the cycle of type feedthrough) results in a fixed point problem and possibly inconsistency (in general, a violation of the SR functional behavior). In this research, we assume that *no cycles exist inside the structure of the graph*. This is a common restriction in SR models, since it largely reduces the complexity of the analysis and avoids algebraic loops, while still leaving enough expressive power to define the behavior of most embedded systems [4].

We use the following example to illustrate the different impact on the schedule of a feedthrough communication with or without delay. In the multi-rate system shown on the left side of Figure 3 (case **(a)**), characterized by communication with oversampling, a possible order of execution of the nodes at simulation time would be the one represented in the upper timeline (bottom part of the figure, labeled as **(a)**). Node $C$ is executed at the base rate and, because of the feedthrough dependencies, it must follow both $A$ and $B$. Clearly, the execution of $C$ *after* $A$ and $B$ makes its schedulability within its period (1 time unit) harder. The execution order in which $C$ executes first corresponds to the model on the right side, labeled as **(b)**, in which a delay of one time unit is added to the communications from $A$ and, respectively, $B$ to $C$.

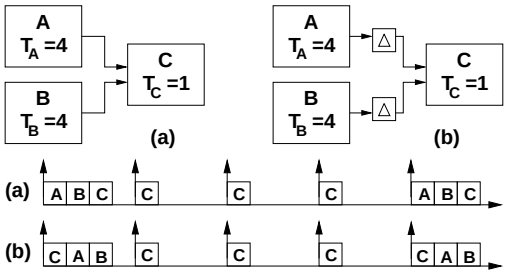


Fig. 3. An example of execution order.

Such added delays change the behavior of the model, and affect the performance of the control algorithm, but they can relax the precedence constraints imposed on the scheduling and improve schedulability. In such cases, if the control error is the performance parameter of interest, it is possible to associate to each delay value a performance cost.

We use a relatively complex example from the Simulink library to illustrate the impact of added delays on control performance. Figure 4 shows a Simulink model of a hydraulic servomechanism controlled by a pulse-width modulated (PWM) solenoid. It is a representative of feedback control loops in which there is a flow of data from the sensor to the control and from the control back to the actuator (represented by the red line over the communication links in the figure).

We change the model to add functional delays to the data communications between the sensors, the actuators and the control, to simulate the effect on the quality of control. Figure 5 shows the same hydraulic servo model, with additional functional delays on the sensor and actuator paths.

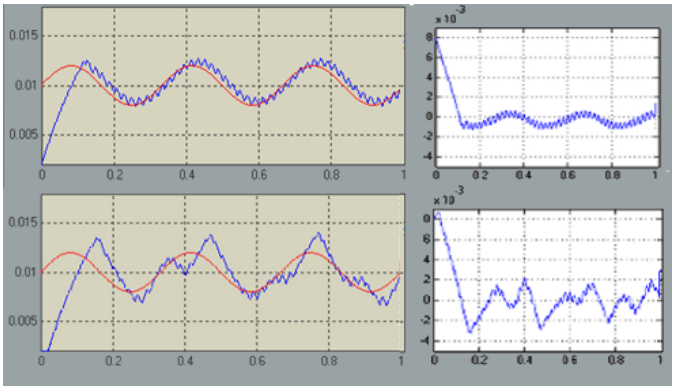

Fig. 6. Actuator position and error for the hydraulic servo without (top) and with (bottom) delays.

Figure 6 shows the results of the simulation of the example model, in the top row without added delays, and in the bottom

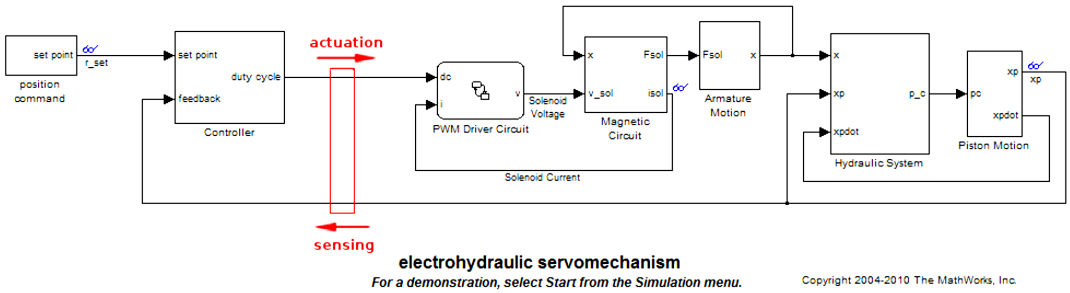


Fig. 4. A Simulink example of an hydraulic servomechanism (representative of a suspension control).

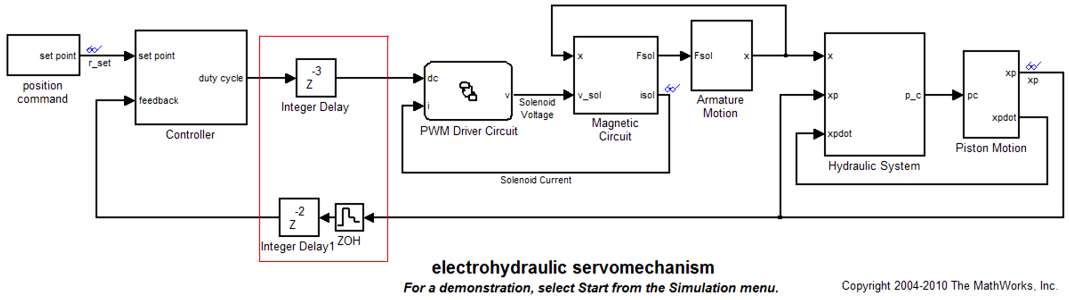


Fig. 5. Hydraulic servo with additional communication delays.

row when a unit delay is added on the sensor path. The results of the simulation show the displacement compared with the reference signal (red line of the left graph), the output (blue line of the left graph), and the error (right graph) in these two cases. The control quality with a unit delay is somewhat degraded, and the simulation results show that the error is now about four times larger. For this control model, it is possible to measure the control error (in millimeter, mm) on the given reference signal for different delay values on the actuator and sensor paths. The results are shown in Table I.

| sensor delay | actuator delay | max error (mm) |
|---|---|---|
| 0 | 0 | 0.75 |
| 1 | 0 | 1.1 |
| 0 | 1 | 1.6 |
| 1 | 1 | 1.6 |

TABLE I
MAX ERRORS FOR DELAYS ON THE SENSOR AND ACTUATOR PATHS

In this case, if the control error is the performance parameter of interest, it is possible to associate to each delay value a performance cost. If convex optimization is used to compute an optimal design configuration, we need to approximate the dependency of the performance from the number of delays with a convex function. For example, we can find a convex hull or linearize the function expressed by the table using a least square approximation, or use continuous piecewise linear function to approximate such curves (approximate formulations for use in mixed integer linear programming (MILP) can be found in [23]). In this paper, we assume that *a function is given which approximates the relationship between the delays on the communication links and the control error.*

## III. IMPLEMENTATION OF SR MODELS

When implementing a synchronous model onto an execution platform, a suitable set of mapping rules must be defined, associating a task to each node and the appropriate scheduling attributes to the tasks. The mapping of the nodes onto tasks (static scheduling of the code sections implementing the output and state update functions of the nodes inside the task) and the scheduling policies must be defined in accordance with the execution rate constraints and partial order of execution defined for the nodes. Also, a mapping should include the design of the communication mechanisms implementing links between nodes. The communication mechanisms must preserve the signal flows in the model for the selected scheduling policy. Formally, every source node must be executed without instance skips, and Equations (1) or (2) must hold for any signal exchanged between two nodes.

For simplicity of presentation, in the mapping of functional nodes onto EDF scheduled systems, we make the same assumption as Prelude, *that each node $N_i$ is implemented by a dedicated task $\tau_i$*. This is clearly impractical because of context switch overheads and in reality multiple nodes are implemented by the same task, but the essence of the problem between communicating nodes belonging to different tasks remains the same. In the following, we use the terms task and node interchangeably. In the implementation, the node executes (its state and output update functions) with a worst case execution time $C_i$, a period equal to the node period $T_i$, an initial phase $\Phi_i$, and a relative deadline $D_i \leq T_i$ which is *no larger than its period*. Instances of a task are referred to as jobs. The $n$-th job of a task $\tau_i$, denoted by $\tau_i[n]$, is released at time $r_i(n) = (n-1)T_i + \Phi_i$ with an absolute deadline

$d_i(n) = r_i(n) + D_i$.

A set of deadline adjustments can be used to enforce the partial order of execution under EDF scheduling. In Prelude, the *absolute deadline* of each job is adjusted as follows:

$$\forall i_j(k) = o_i(m), \quad d_i^*(m) = \min\{d_i(m), d_j^*(k) - C_j\} \quad (3)$$

As a consequence, the writer job is always given higher priority than the reader job, thereby guaranteeing the correct order of execution. The deadline assignment is done in reverse topological order, starting from tasks implementing only nodes without a successor (for which the deadlines are unchanged $d_i^*(n) = r_i(n) + D_i$).

When the communicating blocks are connected by a link with delays, the flow constraints and the corresponding scheduling conditions need to be updated. A general type of precedence constraint with single or multiple delays is defined in [18], where the schedulability analysis on EDF is discussed and the authors show how to improve schedulability by relaxing the precedence constraints (rather than enforcing the correct execution order by changing the deadlines). When a delay is added on the communication link $N_i \rightarrow N_j$, it is no longer required that $i_j(k) = o_i(m)$ where $m = n_i(r_j(k))$. The relaxed version requires that $i_j(k-d) = o_i(m+e)$ where $d$ and $e$ are non-negative integers (the communication delay expressed as the sum of a number $d$ of writer periods, plus an integer number $e$ of reader periods). This schedulability improvement comes with a cost. The relaxation of precedence constraints requires an increase in the size of communication buffers, since the output results of the delayed instance will have to be stored in memory. It also changes the behavior of the model, and affects the performance of the control algorithm, as discussed in the previous section.

### A. Trading functional delays for schedulability

The two competing methods (deadline modification and precedence relaxation by adding delays) can be used to find the feasible (schedulable) solution with minimum performance/memory cost. However, the task of finding optimal values for deadline assignments and the integers $d$ and $e$ while minimizing buffer sizes is a complex optimization problem.

As a special case of [18], we restrict the delay addition to $d = 1$ and $e = 0$. This is sufficient to ease the system schedulability in the case of deadlines lower than or equal to periods ($D \leq T$). For example, in Figure 2, the $(m-1)$-th instance of $N_i$ generates data for the $k$-th instance of $N_j$, and its absolute deadline is no larger than $r_i(m)$, and consequently, $r_j(k)$. Thus, the flow preservation requirements do not impose any additional constraints on the system schedulability, compared to independent tasks.

The strategy in Prelude is to adjust the deadline of each node, without additional communication delays. From the standpoint of model-based design, this adheres to a strict cascade flow: the functional model has been designed and validated and should not be modified. However, the systems with modified deadlines may simply be unschedulable, as shown by the following simple example.

**Example:** Consider a system that consists of two tasks $\tau_1 \rightarrow \tau_2$ with $\tau_1(T_1 = 3, C_1 = 1.5)$ and $\tau_2(T_2 = 2, C_2 = 0.75)$. Offsets are 0 for both tasks and deadlines are equal to their periods. If we try to schedule the tasks with EDF, using the deadline encoding technique of Prelude, then the system is not schedulable because $d_1(0) = \min(3, 2 - 0.75) = 1.25 < C_1$.

In this case, if a unit delay is added on the link between the two tasks, then the deadline constraint can be relaxed. The system becomes schedulable with EDF, as the two tasks are independent with a total utilization of $1.5/3 + 0.75/2 = 87.5\%$.

Once again, this example shows the possible tradeoff between functional delays and schedulability. In this work, we propose a method for the optimal addition of unit delays into dynamically (EDF) scheduled systems. Delays are added minimally to the system to enhance schedulability without incurring an excessive penalty in the overall response times. Our work targets EDF scheduled systems and applies to any pair of (reader/writer) task periods, as opposed to [13], in which tasks are scheduled by fixed-priority, and every pair of communicating blocks must have harmonic periods. .

As in (3), we allow the (independent) assignment of an absolute deadline to each individual job. The example below shows how a job-oriented deadline adjustment can avoid some unnecessary delays as compared to the constraint relaxation approach in [18] (which requires that the same relative deadlines is assigned to all instances of the same task).

**Example:** Consider a system that consists of 3 tasks: $\tau_A(T_A = 3, C_A = 0.5)$, $\tau_B(T_B = 3, C_B = 1)$ and $\tau_C(T_C = 2, C_C = 1)$ such that $\tau_B \rightarrow \tau_C$. Release offsets are assumed to be 0 for all tasks and relative deadlines are initially equal to periods. If we try to schedule the system using EDF, with a single relative deadline per task and using the constraint relaxation approach, the relative deadline of $\tau_B$ cannot exceed 1 as the latest start time of the first instance of $\tau_C$ is 1. However, this will assign an absolute deadline at $t = 4$ to the second instance of $\tau_B$, which is violated. The only way to schedule this system is to relax the constraint on the communication link $\tau_B \rightarrow \tau_C$ by introducing a unit delay buffer on it.

If we can assign different relative deadlines for different jobs of the same task, the second instance of $\tau_B$ will have an absolute deadline at $t = 5$, and the system is schedulable. A valid schedule is shown in Figure 7 (without additional delays).

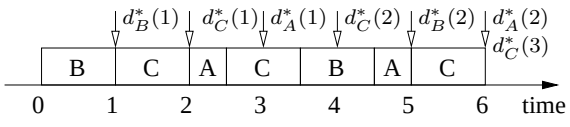


Fig. 7. Benefits of allowing different relative deadlines for different instances of the same task. Downwards arrows denote the absolute deadlines.

In practice, absolute deadline adjustment can be equivalently achieved using an array of relative deadlines for each task. Each entry in this array (referred to in Prelude as deadline word) is the relative deadline for a single job of the task. Because of the periodic pattern of task releases in the hyper-

period $H$ (the least common multiple of the task periods), it is sufficient to use an array $D_i^*$ of size $n_i = \frac{H}{T_i}$ to store the relative deadlines for each task $\tau_i$. The $k$-th instance of $\tau_i$ will take the relative deadline $D_i^*(k \mod n_i)$, for which the absolute deadline is $d_i^*(k) = D_i^*(k) + r_i(k)$. For example, the deadline array for task $\tau_B$ in Figure 7 is $\{1,2\}$. A rolling counter $c_i$ is used to indicate the deadline index. $c_i$ is incremented at each new activation of $\tau_i$, and when $c_i = n_i$, it is reset to zero.

## IV. Optimization Algorithm

In this section, we present our algorithm for finding the schedulable system configuration with minimum weighted sum of additional functional delays. The input is a graph representing a system of nodes with feedthrough dependencies. The weight of an edge represents the associated cost if a functional delay is added to the communication link. *These costs depend on the control functionality defined by the model and represent performance and/or memory penalties. We assume such costs can be estimated by the control designer*. Without loss of generality, we assume that each link $i$ is associated with a cost ($cost_i$) in the range $[0, 1]$, such that $\sum_{i=1}^{n} cost_i = 1$. Moreover, we assume that *the system is EDF schedulable when all feedthrough dependencies are removed*. This implies that the overall utilization $U \leq 1$.

We develop two algorithms to insert functional delays in the communication links to relax feedthrough dependencies. A branch-and-bound procedure for finding a schedulable system configuration is presented in Section IV-A. This procedure always finds the configuration with the minimal weighted sum of additional delays. However, it takes a long time to execute even for small systems and is unscalable to large systems. We present a faster and more scalable heuristic approach in Section IV-B.

### A. Branch-and-Bound Algorithm

The branch-and-bound algorithm guarantees to find a configuration of added delays that has the minimal cost of any feasible (schedulable) configuration. It sets two endpoints: a (possibly infeasible) solution with no added delays (the starting point). For our problem, we also identify an initial feasible solution (the initial bound) that adds delays to all the communication links. The algorithm begins at the starting point, exhaustively enumerates all candidate solutions in a tree-like structure and prunes non-promising branches, until a feasible solution of minimal cost is found.

Each vertex in the branch-and-bound tree presents a distinct solution (configuration of added delays). The vertex is characterized by: *delay configuration, cost, bound, and schedulability*. The cost of a vertex is the sum of the costs of its delays. The bound of a vertex is simply the best solution that can be obtained if this vertex is further expanded (its children are generated). The schedulability of a vertex is the state of the system (schedulable or not) when its delay configuration is enacted. An *active* vertex has a bound lower than the current best solution. A priority queue (`PQ` in Algorithm 1) is used to store active vertices that have not been expanded yet. The vertex with the lowest bound is given the highest priority. In this way, the most promising solutions are explored first and vertices can be pruned faster. An example of a branch-and-bound tree is shown in Figure 8. In this example it is assumed that the system has 3 delays and that their costs are 1, 2, and 3 respectively.

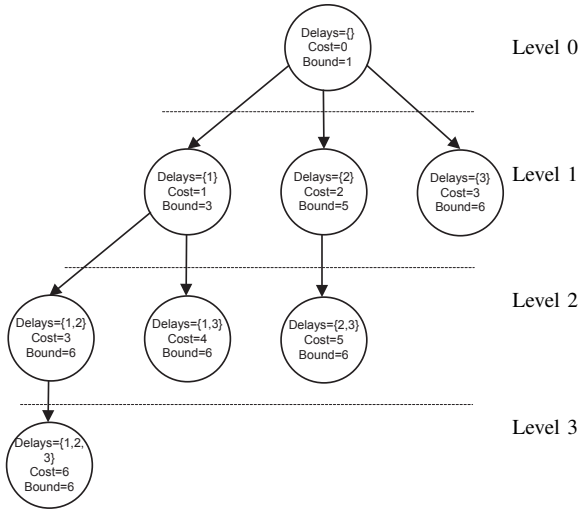


Fig. 8. Example of a branch-and-bound tree.

The proposed branch-and-bound is shown in Algorithm 1 below. A branch-and-bound algorithm typically consists of the following parts: *initialization*, *branching*, *evaluation*, *update*, and *pruning*. For the proposed algorithm, the *initialization* phase assigns values to the endpoints (line 1 of Algorithm 1). The `root` is the search tree starting point and represents the (possibly infeasible) solution with minimum cost (no delays added). The current best solution `optimalsol` is initialized to the other endpoint. This solution can simply be achieved by removing all feedthrough dependencies (adding delays to all links). This solution is schedulable as required by our assumptions and has a cost of 1.

After the endpoints are defined, iterative *branching* (the while loop in lines 4-21) is used to explore the possible delay configurations. Starting from the root which has no delays, functional delays are added incrementally. The *level of a vertex* in the branch-and-bound tree (the distance of the vertex to the root) is defined as the number of added delays. If a parent has $x$ delays (of level $x$), then its children will each have $x + 1$ delays. Each child inherits the same $x$ delays from the parent plus a single additional delay which is distinct for each child as shown in Figure 8.

In order to avoid redundant delay additions, an order is enforced on the links. The list of communication links in the system, `Links` is sorted by increasing cost (line 3 of Algorithm 1). To prevent the creation of redundant vertices, the distinct delay of each child must have a higher cost (higher index in `Links`) than all of the $x$ delays of the parent. If we denote the highest index (cost) link of the parent by

**Algorithm 1:** Branch-and-Bound Algorithm for Minimizing the Weighted Sum of Functional Delays

```
1: root,optimalsol= initialize()
2: PQ.push(root)
3: Links.sortLow2High()
4: while PQ not empty do
5:   parent = PQ.pop()
6:   startIndex = getLinksIndex(parent)+1
7:   for i = startIndex → Links.size() do
8:     createChild(i)
9:     checkSchedulability(child)
10:    if system is schedulable then
11:      if child.cost < optimalsol.cost then
12:        optimalsol=child
13:        prune(PQ,optimalsol.cost)
14:      end if
15:    else
16:      if child.bound < optimalsol.cost then
17:        PQ.push(child)
18:      end if
19:    end if
20:  end for
21: end while
```

$j$, then its first child will have the same $x$ delays of the parent plus a delay on the link identified by `Links[j+1]`. The second child will have the same $x$ delays and a delay on `Links[j+2]` and so on. This operation is in lines 6-8. The function `getLinksIndex()` returns the index $j$ of the parent delay with highest cost. `createChild(`$i$`)` is then called $n - j$ times, where $n$ is the number of links in the system. Each time, `createChild(`$i$`)` copies the delays of the parent into the child $c$ and then adds a delay on the link `Links[i]` where $i \in [j+1, n]$. The other attributes of the child $c$ are calculated by `createChild(`$i$`)`. The schedulability of the system is checked (with the improved necessary-and-sufficient test in [27]) assuming the vertex delays are assigned to the system links. The cost of $c$ is the sum of the costs of its delays, and its lower cost bound is `c.bound=c.cost+cost(Links[i+1])`.

Once a vertex is created, it is immediately visited and evaluated. If the system is schedulable, the cost of the vertex is compared against the cost of the current best solution. If it is lower, the best solution `optimalsol` is *updated*. The queue of active vertices is also updated (line 13), by pruning all the vertices with lower bounds higher than the new best solution. If the system is not schedulable, then we check if the vertex is active (line 16). This is done by checking whether its bound is lower than the current best solution. If so, the children of the vertex can still potentially provide a better solution, and the vertex is added to `PQ`.

While the branch-and-bound algorithm guarantees optimality, it is very time consuming. Experimental results (in Section V) show that this algorithm takes a long time to execute even for relatively small systems. To overcome the scalability issue, we present a scalable heuristic algorithm.

### *B. Heuristic algorithm*

The heuristic algorithm provides a scalable solution which is capable of handling complex systems with a large number of nodes/tasks. Unlike the branch-and-bound, it does not guarantee optimality. However, as shown in Section V, for most systems, it computes the optimal solution and is orders of magnitude faster than the branch-and-bound. The heuristic is summarized in Algorithm 2. It operates in two phases:

- **Phase 1:** Find an initial feasible solution;
- **Phase 2:** Improve on the solution by removing unnecessary delays.

In Phase 1, an initial schedulable solution is found. The deadline modification algorithm is first applied to the system (possibly making it not schedulable). Then, we create a list `AffectedNodes` of tasks that have modified deadlines because of feedthrough communications. In the next step, delays are selectively added to the links between those nodes and their successors to relax feedthrough constraints. The function `critSucc(`$\tau_i$`)` returns the critical successor of a task $\tau_i$ (the successor with the earliest deadline). If $\tau_i$ has no critical successor ($\tau_i$ does not have a successor with feedthrough constraints and its deadline is its original deadline), then the function returns -1. For each task $\tau_i$ in `AffectedNodes`, a delays is added to the link $(\tau_i, \texttt{critSucc}(\tau_i))$. The list `AffectedNodes` is updated after the addition of functional delays because some tasks may have more than one receivers that may affect their deadlines. In this case, adding a delay on the link $(\tau_i, \texttt{critSucc}(\tau_i))$ will simply change, not eliminate, the critical successor of $\tau_i$. The function `updateAffectedNodes()` identifies the new set of critical successors based on the new delay assignment. This operation continues until the list `AffectedNodes` becomes empty after calling `updateAffectedNodes()`, which means that there are no modified deadlines due to feedthrough dependencies and the system is schedulable.

**Algorithm 2:** Heuristic Algorithm for Minimizing the Weighted Sum of Functional Delays

```
1: Phase 1:
2: AffectedNodes = List of tasks τ_i with critSucc(τ_i) != -1
3: while AffectedNodes not empty do
4:   τ_i ←AffectedNodes.head
5:   addDelay(τ_i, critSucc(τ_i))
6:   updateAffectedNodes()
7: end while
8:
9: Phase 2:
10: Δ = List of added delays δ_i = (s_i, r_i)
11: Δ.sort()
12: for all delays δ_i = (s_i, r_i) in Δ do
13:   removeDelay(s_i, r_i)
14:   updataDeadlines()
15:   schedAnalysis()
16:   if system is not schedulable then
17:     addDelay(s_i, r_i)
18:   end if
19: end for
```

The solution obtained at the end of Phase 1 is schedulable, but typically has a very high cost. Phase 2 is an optimization procedure that takes the solution obtained in the previous phase and improves it by removing redundant delays. The delays added in Phase 1 are placed in a list $\Delta$ and sorted by decreasing cost. Then, they are tentatively removed one-by-one, starting with those with highest cost. After the removal of each delay, the schedulability of the system is analyzed (`schedAnalysis()`) using the procedure in [27]. After each removal of a delay, the function `updateDeadlines()` is called to recalculate the deadlines of all the tasks affected by the change.

If the system becomes unschedulable after removing a delay, the delay is restored. Sorting and trying to remove delays by cost plays an important role in reducing the overall cost of the final solution. Several alternatives exist for adding a delay to relax the deadline of a given task. If the low cost alternatives are removed first, the system may become unschedulable when the high cost alternatives are removed.

In some applications, delays on several communication links may have the same impact on the system performance (in terms of the control quality or memory overhead). To break these ties, we add a second sorting criterion to the function `sort()`. Added delays are sorted (in order) by

- decreasing cost of their link;
- increasing worst case execution time of the sender, if the costs of the links are the same.

We tried other policies for selecting the task link from the list $\Delta$. In our experiments they were less effective than the simple heuristic based on the worst case execution time values, and are discussed as alternative approaches in Section V-A1.

## V. Experimental Results

In this section, we used randomly generated systems and an automotive case study to evaluate the performance of the proposed algorithms. The experiments are performed on a system with 2.8 GHz CPU and 8 GB of memory.

### A. Random Systems

In the first set of experiments, we use TGFF [3] to generate random task graphs. Each task has a maximum fan-out of 3 and a maximum fan-in of 2. For comparison purposes, the average number of tasks generated for each system configuration in this experiment is first kept at 15, as the runtime for branch-and-bound becomes very high for larger systems. After the generation of the system, the system parameters (utilizations, periods, execution times, deadlines) are generated as follows. The total utilization (U) of the system is between 0.5 and 0.99. We use the UUniFast algorithm [5] to assign to each tasks a uniformly distributed utilization $u_i$ such that $\sum_{i=1}^{n} u_i$ = U. The period of each task is randomly selected from the set {5, 10, 20, 40, 50, 100, 200, 400, 500, 1000}ms. The execution time $C_i$ for each task was derived from its period $T_i$ and utilization $u_i$. The deadline of each task is initially set equal to its period. For each total utilization $U$, we generate 1000 random systems.

In order to show the improvement obtained using job-level deadlines as opposed to a single deadline for each task (as proposed in [18]), another version of the heuristic algorithm using a single deadline per task is implemented. Experimental results for the three algorithms (branch-and-bound as in Algorithm 1, heuristic with job-level deadline as in Algorithm 2, heuristic with task-level deadline) are shown in Figures 9 and 10.

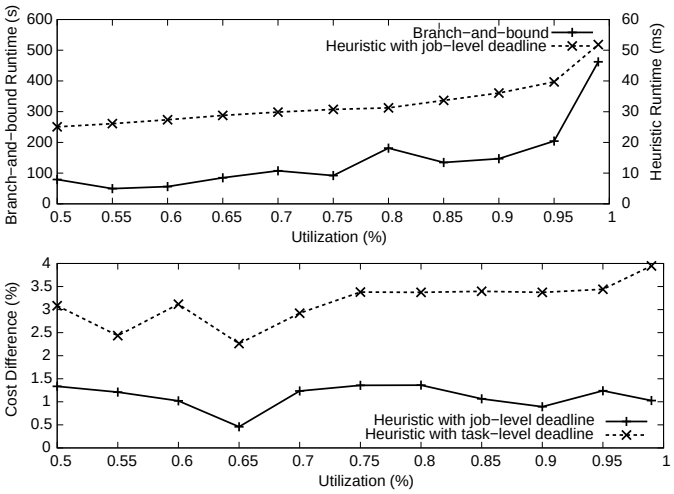


Fig. 9. Comparison among algorithms for random weights.

Figure 9 shows the experimental results obtained when random costs are assigned to delays. The bottom graph shows the additional cost of the solution found by the two heuristics, compared to the branch-and-bound solution. The top graph shows the runtime of the branch-and-bound algorithm (using left y-axis) and the heuristic with job-level deadline (using right y-axis). The heuristic with job-level deadline provides a reasonable solution both in terms of runtime and accuracy. Its cost is in average only 1.1% higher than the branch-and-bound algorithm (the optimum solution). However, its average runtime is 32.8ms, or about 4400 time faster. The figure also shows the performance cost of imposing a single relative deadline for each task as compared to having one deadline assigned to each job. The additional cost is 3.25% compared to the optimum.

In the case in which weights are all identical (the optimization objective is to minimize the number of added delays), the heuristic algorithm depends on the secondary sorting criterion (to sort the list $\Delta$ in Phase 2 of Algorithm 2). As shown in Figure 10, the results are similar to the case of random weights. The heuristic with job-level deadline adds 4.36% more delays on average compared to the optimal solution provided by the branch-and-bound algorithm, and the runtime is 2 to 3 orders of magnitude smaller. When all jobs have the same relative deadline was imposed, the results are 8.14% worse than the optimum.

To study the scalability of the heuristic algorithm, we generated systems containing up to 150 tasks. Figure 11 shows the average runtime for the heuristic algorithm as a function

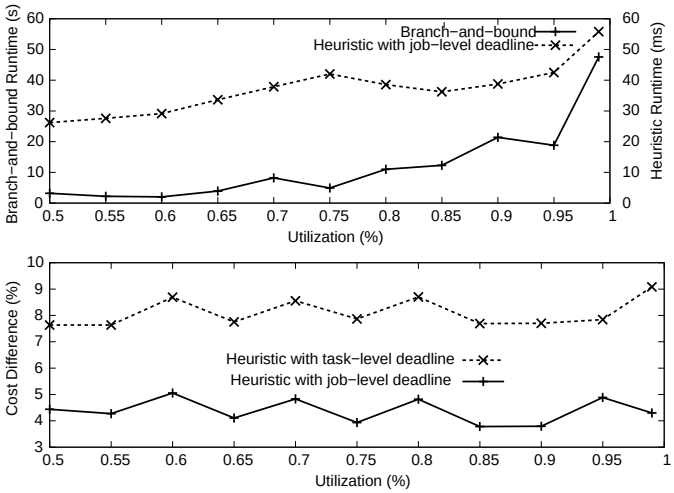


Fig. 10. Comparison among algorithms for minimizing number of added delays.

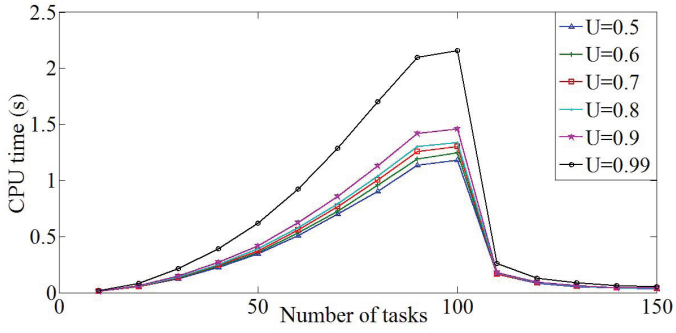


Fig. 11. Runtime of heuristic vs. system size at different utilizations.

of the number of tasks in the system for different utilizations. As shown in the figure, the runtime increases with the system utilization. However, the runtime increases with the number of tasks in the range $[10, 100]$ and decreases afterwards. This is largely due to the fact that when the task number increases, individual task utilizations become smaller and a schedulable solution can be easier to find. The longest runtime for the heuristic is for systems with 100 tasks and a total utilization of 99%, where it takes about 2.15 seconds per system on average.

*1) Algorithm Design Alternatives:* The heuristic algorithm presented in this paper is the result of a selection process in which several alternative options have been tried and evaluated, before the final algorithm presented in Section IV is selected as the best one.

When trying to improve the schedulability of a system, the most direct way is to detect the tasks that cause schedulability failures and add a functional delays to its outgoing links. As mentioned earlier, we use the schedulability analysis method in [27], where a number of endpoints $t$ of time intervals belonging to a finite set and upper bounded by $t_f$ are checked for a possible violation of a schedulability condition. The algorithm starts from the maximum value $t_f$ and goes backwards in time to 0 to check whether $\forall t < t_f, h(t) \leq t$ ($h(t)$ is the requested load with deadline less than t). In case this condition fails, we try to detect the violating link (the one that causes an increase in the function $h(t)$) and add a delay onto it. Our experiments with this approach showed that in many cases there are many links that contribute to the failure. In general, it is very difficult to find the best link on which the delay should be added. The approach of starting with a feasible solution and then removing unnecessary delays, starting with the ones with higher cost performs better.

| | $C$ | $\overline{D}$ | $U$ | $\overline{D} - C$ | $\overline{\rho}$ |
|---|---|---|---|---|---|
| # Delays | 24108 | 24226 | 24681 | 24297 | 24674 |
| Runtime (s) | 408.34 | 630.51 | 366.26 | 631.48 | 480.51 |

TABLE II
VARIATIONS OF THE SECOND SORTING CRITERION IN ALGORITHM 2

The sorting criterion used to select the order in which delays are tentatively removed (Phase 2 of the heuristic) is critical for obtaining good solutions. Experiments showed that if the sorting step (line 11 of Algorithm 2) is removed, the cost of the solution increases by about 35% for the case of random weights. For applications where the weights of the links are all equal (thus the objective is to minimize the number of delays), the second sorting criterion is critical, and sorting by execution time ($C$) proves to be the best option. We have tried several other metrics, as shown in Table II. The first alternative was sorting by relative deadlines ($D$). Since each job has its deadline, to rank tasks we used the average of all the relative deadlines ($\overline{D}$) of their jobs in the time interval covered by the schedulability analysis. Other variants we tried are: utilization ($U$), laxity ($\overline{D} - C$), and the average density which is given by:

$$\overline{\rho}_i = \frac{\sum_{k=1}^{n_i} \frac{C_i}{D_i^*(k)}}{n_i} \tag{4}$$

where $D_i^*(k)$ is the relative deadline of the $k$-th instance of task $\tau_i$. Table II reports the total number of added delays and the runtime for the 11000 random systems. All the alternative sorting metrics computed solutions with higher cost (than the proposed algorithm) in approximately the same time.

### B. An Automotive Case Study

We apply the optimization algorithm to an automotive case study (available in [13]). The experiments are performed on an industrial case study consisting of a fuel injection embedded controller. The case study is a simplified version of the full control system with 90 function blocks (out of 200 in the real system), executed with 7 different periods (in ms): 4, 5, 8, 12, 50, 100, and 1000. The execution times of some functions are provided as part of the case study. The others are assigned to achieve a system utilization of 94.1%, which is close to the values found in real systems of this type. The function blocks are communicating through 106 links. Among them, 37 are from high-rate to low-rate nodes, and 31 are for low to high communications. The details of the communication graph (including the size of the communication links) and the time

parameters of the function blocks (including their periods and WCETs) can be found in [13].

The case study is unschedulable with Prelude because of the imposed feedthrough dependencies. However, our heuristic algorithm was able to find a feasible solution while minimizing the memory requirements of the added functional delays to 1050 bytes, which is about 16.0% of the memory cost (6578 bytes) if a delay is added to all the communication links. The solution was reached in 0.28 seconds.

## VI. Conclusions

In this paper, we consider the optimization of the multitask implementation for synchronous reactive models scheduled with dynamic priority (EDF). We show that there is opportunity for improving the conservative behavior of previous work by reducing the need for additional latency and memory buffers when implementing the functional model. We present an algorithm to minimize the weighted sum of the additional functional delays and demonstrate that the proposed heuristic yields a solution with close-to-optimal solution compared to exhaustive search.

## References


[1] MathWorks. The Mathworks Simulink and StateFlow User's Manuals. web page: http://www.mathworks.com.

[2] Esterel Technologies. SCADE Suite. web page: www.esterel-technologies.com.

[3] Task Graphs For Free. [Online] *Available at http://ziyang.eecs.umich.edu/ dickrp/tgff*

[4] G. Berry and G. Gonthier. "The Esterel synchronous programming language: Design, semantics, implementation." *Sci. Comput. Program*, 19(2):87–152, Nov. 1992.

[5] E. Bini and G. Buttazzo. "Measuring the Performance of Schedulability Tests." *Real-Time Systems*, 30(1-2):129–154, May 2005.

[6] P. Caspi and M. Pouzet. "Synchronous kahn networks." In *Proc. ACM SIGPLAN Conference on Functional Programming*, 1996.

[7] P. Caspi, A. Curic, A.Maignan, C. Sofronis, S. Tripakis, and P. Niebert. "From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications." In Proc. ACM SIGPLAN conference on Language, compiler, and tool for embedded systems, 2003.

[8] P. Caspi, N. Scaife, C. Sofronis, and S. Tripakis. "Semantics-preserving multitask implementation of synchronous programs." *ACM Trans. Embed. Comput. Syst.*, 7(2):1–40, Jan. 2008.

[9] P. Caspi and A. Benveniste. "Time-robust discrete control over networked loosely time-triggered architectures." In *Proc. IEEE Control and Decision Conference*, 2008.

[10] H. Chetto, M. Silly, and T. Bouchentouf. "Dynamic scheduling of real-time tasks under precedence constraints." *Real-Time Systems*, 2(3):181–194, Sept. 1990.

[11] M. Cordovilla, F., J. Forget, E. Noulard, and C. Pagetti. "Developing critical embedded systems on multicore architectures: the Prelude-SchedMCore toolset." In *Proc. 19th International Conference on Real-Time and Network Systems*, 2011.

[12] M. Di Natale and V. Pappalardo. Buffer optimization in multitask implementations of simulink models. *ACM Trans. Embed. Comput. Syst.*, 7(3):1–32, 2008.

[13] M. Di Natale, L. Guo, H. Zeng, and A. Sangiovanni-Vincentelli. "Synthesis of Multi-task Implementations of Simulink Models with Minimum Delays." *IEEE Transactions on Industrial Informatics*, 6(4):637–651, Nov. 2010.

[14] S. A. Edwards. "An Esterel compiler for large control-dominated systems." *IEEE Trans. Computer-Aided Design*, 21(2):169–183, Feb. 2002.

[15] J. Forget, F. Boniol, D. Lesens, and C. Pagetti. "A multi-periodic synchronous data-flow language." In *Proc. 11th IEEE High Assurance Systems Engineering Symposium*, 2008.

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

[17] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. "Programming real-time applications with SIGNAL." *Proceedings of the IEEE*, 79(9):1321-1336, Sept. 1991.

[18] L. Mangeruca, M. Baleani, A. Ferrari, and A. Sangiovanni-Vincentelli. "Uniprocessor scheduling under precedence constraints for embedded systems design." *ACM Transactions on Embedded Computing Systems*, 7(1), Dec. 2007.

[19] C. Pagetti, J. Forget, F. Boniol, M. Cordovilla, and D. Lesens. "Multi-task Implementation of Multi-periodic Synchronous Programs." *Discrete Event Dynamic Systems*, 21(3):307–338, Sept. 2011.

[20] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. "Concurrency in synchronous systems." *Formal Methods in System Design*, 28(2):111-130, March 2006.

[21] C. Sofronis, S. Tripakis, and P. Caspi. "A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling." In *Proc. 6th ACM International Conference on Embedded Software*, 2006.

[22] S. Tripakis, C. Pinello, A. Benveniste, A. Sangiovanni-Vincentelli, P. Caspi and M. Di Natale. "Implementing Synchronous models on Loosely Time-Triggered Architectures." *IEEE Transactions on Computers*, 57(10):1300–1314, Oct. 2008.

[23] J. P. Vielma, S. Ahmed and G. Nemhauser. "Mixed-integer models for non-separable piecewise linear optimization: unifying framework and extensions." *Operations Research*, 58:303-315, 2010.

[24] G. Wang, M. Di Natale, and A. Sangiovanni-Vincentelli. "Improving the size of communication buffers in synchronous models with time constraints." *IEEE Transactions on Industrial Informatics*, 5(3):229–240, Aug. 2009.

[25] D. Weil, V. Berlin, E. Closse, M. Poize, P. Venier, and J. Pulou. "Efficient compilation of Esterel for real-time embedded systems." In *Proc. Int. Conf. Compilers, Architecture, and Synthesis for Embedded Syst.*, 2000

[26] H. Zeng and M. Di Natale. "Mechanisms for Guaranteeing Data Consistency and Time Determinism in AUTOSAR Software on Multi-core Platforms." In *Proc. 6th IEEE Symposium on Industrial Embedded Systems*, 2011.

[27] F. Zhang and A. Burns. "Schedulability Analysis for Real-Time Systems with EDF Scheduling." *IEEE Transactions on Computers*, 58(9):1250–1258, Sept. 2009.