# From Dataflow Specification to Multiprocessor Partitioned Time-triggered Real-time Implementation[*]

**Thomas Carle[1], Dumitru Potop-Butucaru[2], Yves Sorel[3], and David Lesens[4]**

1 **Brown University**
**247 Browen Street, Providence, Rhode Island 02906, USA**
`http://orcid.org/0000-0002-1411-1030`
`thomas_carle@brown.edu`

2 **INRIA Paris-Rocquencourt**
**Domaine de Voluceau, Rocquencourt – B.P. 105, 78153 Le Chesnay, France**
`http://orcid.org/0000-0003-3672-6156`
`dumitru.potop@inria.fr`

3 **INRIA Paris-Rocquencourt**
**Domaine de Voluceau, Rocquencourt – B.P. 105, 78153 Le Chesnay, France**
`yves.sorel@inria.fr`

4 **Airbus Defence & Space**
**Route de Verneuil, 78133 Les Mureaux, France**
`http://orcid.org/0000-0002-3452-9960`
`david.lesens@astrium.eads.net`

## Abstract

Our objective is to facilitate the development of complex time-triggered systems by automating the allocation and scheduling steps. We show that full automation is possible while taking into account the elements of complexity needed by a complex embedded control system. More precisely, we consider deterministic functional specifications provided (as often in an industrial setting) by means of synchronous data-flow models with multiple modes and multiple relative periods. We first extend this functional model with an original real-time characterization that takes advantage of our time-triggered framework to provide a simpler representation of complex end-to-end flow requirements. We also extend our specifications with additional non-functional properties specifying partitioning, allocation, and preemptability constraints. Then, we provide novel algorithms for the off-line scheduling of these extended specifications onto partitioned time-triggered architectures *à la* ARINC 653. The main originality of our work is that it takes into account at the same time multiple complexity elements: various types of non-functional properties (real-time, partitioning, allocation, preemptability) and functional specifications with conditional execution and multiple modes. Allocation of time slots/windows to partitions can be fully or partially provided, or synthesized by our tool. Our algorithms allow the automatic allocation and scheduling onto multi-processor (distributed) systems with a global time base, taking into account communication costs. We demonstrate our technique on a model of space flight software system with strong real-time determinism requirements.

**2012 ACM Subject Classification** Computer systems organization Real-time systems
**Keywords and phrases** Time-triggered, off-line real-time scheduling, temporal partitioning
**Digital Object Identifier** 10.4230/LITES-v002-i002-a001
**Received** 2013-10-01 **Accepted** 2015-07-22 **Published** 2015-10-01

[*] Partially supported by the FUI8 PARSEC project.

## 1 Introduction

This paper addresses the implementation of embedded control systems with strong functional and temporal determinism requirements. The development of these systems is usually based on *model-driven* approaches using high-level formalisms for the specification of functionality (Simulink, Scade[11]) and/or real-time system architecture and non-functional requirements (AADL [18], UML/Marte [34]).

The temporal determinism requirement also means that the implementation is likely to use *time-triggered* architectures and execution mechanisms defined in well-established standards such as TTA, FlexRay [44], ARINC 653 [3], or AUTOSAR [5].

The time-triggered paradigm describes sampling-based systems (as opposed to event-driven ones) [28] where sampling and execution are performed at predefined points in time. The offline computation of these points under non-functional constraints of various types (real-time, temporal isolation of different criticality sub-systems, resource allocation) often complicates system development, when compared to classical event-driven systems. In return for the increased design cost, system validation and qualification are largely simplified, which explains the early adoption of time-triggered techniques in the development of safety- and mission-critical real-time systems.

### 1.1 Contribution

The objective and contribution of this paper is to facilitate the development of time-triggered systems by automating the allocation and scheduling steps for significant classes of functional specifications, target time-triggered architectures, and non-functional requirements. On the application side, we consider general dataflow synchronous specifications with *conditional execution*, *multiple execution modes*, and *multiple relative periods*. Explicitly taking into account conditional execution and execution modes during scheduling is a key point of our approach, because the offline computation of triggering dates limits flexibility at runtime. For instance, taking into account conditional execution and modes allows for better use of system resources (efficiency) and a simple modeling of reconfigurations.

On the architecture side, we consider multiprocessor *distributed architectures*, taking into account *communication costs* during automatic allocation and scheduling.

In the non-functional domain, we consider *real-time*, *partitioning*, *preemptability*, and *allocation constraints*. By *partitioning* we mean here the *temporal partitioning* specific to TTA, FlexRay (the static segment), and ARINC 653, which allows the static allocation of CPU or bus time slots, on a periodic basis, to various parts (known as partitions) of the application. Also known as static time division multiplexing (TDM) scheduling, partitioning further enhances the temporal determinism of a system.

The main originality of our paper is to consider all these aspects together, in an integrated fashion, thus allowing the automatic implementation for complex real-life applications. Other originality points concern the specification of real-time properties, which we adapted to our time-triggered framework, and the handling of partitioning information. In the specification of real-time properties, the use of deadlines that are longer than the periods naturally arises. It allows a more natural real-time specification, improved schedulability, and less context changes between partitions (which are notoriously expensive). Regarding the partitioning information, allocation of time slots/windows to partitions can be fully or partially provided, or synthesized by our tool.

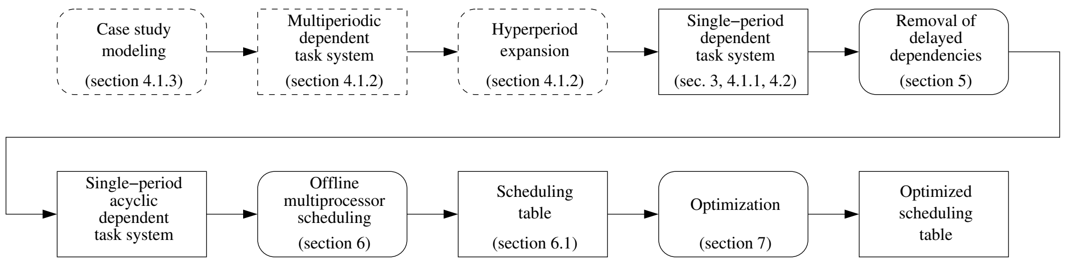
Case study modeling (section 4.1.3)
Multiperiodic dependent task system (section 4.1.2)
Hyperperiod expansion (section 4.1.2)
Single–period dependent task system (sec. 3, 4.1.1, 4.2)
Removal of delayed dependencies (section 5)
Single–period acyclic dependent task system
Offline multiprocessor scheduling (section 6)
Scheduling table (section 6.1)
Optimization (section 7)
Optimized scheduling table

**Figure 1** The proposed implementation flow.

## 1.2 The Application

We apply our technique on a model of spacecraft embedded control system. Spacecrafts are subject to very strict real-time requirements. The unavailability of the avionics system (and thus of the software) of a space launcher during a few milliseconds during the atmospheric phase may indeed lead to the destruction of the launcher. From a spacecraft system point of view, the latencies are defined between acquisitions of measurement by a sensor to sending of commands by an actuator. Meanwhile, the commands are established by a set of control algorithms (Guidance, Navigation and Control or GNC).

Traditionally, the GNC algorithms are implemented on a dedicated processor using a classical multi-tasking approach. But today, the increase of computational power provided by space processors allows suppressing this dedicated processor and distributing the GNC algorithms on the processors controlling either the sensors or the actuators. For future spacecraft (space launchers or space transportation vehicles), the navigation algorithm could for instance run on the processor controlling the gyroscope, while the control algorithm could run on the processor controlling the thruster. The suppression of the dedicated processor allows power and mass saving.

But the sharing of one processor by several pieces of software of different Design Assurance Levels (*e.g.* gyroscope control and navigaton) requires the use of an operating system ensuring the Time and Space Partitioning (or TSP) between these pieces of software. Such operating systems, like ARINC 653 [3], feature a hierarchic 2-level scheduler where the top-level one is of static time-triggered (TDM) type. Furthermore, for predictability issues, the processors of the distributed implementation platform share a common time base. This means that the execution platform offers the possibility of a globally time-triggered implementation. We therefore aim for distributed implementations that are time-triggered at all levels, which improves system predictability and allows the computation of tighter worst-case bounds on end-to-end latencies.

The scheduling problem we consider is therefore as follows: End-to-end latencies are defined at spacecraft system level, along with the offsets of sensing and actuation operations. Also provided are safe worst case execution time (WCET) estimations associated to each task on each processor on which they can be executed. What must be computed is the time-triggered schedule of the system, including the activation times of each partition and each task, and the bus frame.

## 1.3 Proposed Implementation Flow and Paper Outline.

The remainder of the paper is organized as follows. Section 2 reviews related work. Section 3 presents our target execution platforms. It spends significant space on the careful definition of time-triggered and partitioned systems. Sections 4 to 7 define the proposed implementation flow for time-triggered systems, whose global structure is depicted in Figure 1. These sections also present experimental results. Section 8 concludes.

The two types of boxes in Figure 1 (dashed and solid) reflect the dual nature of our paper, which contains not only work on the definition of new scheduling algorithms, but also on integrating these algorithms into a larger implementation flow. First of all, our paper provides the full formal specification of an off-line scheduling problem for single-period dependent task systems with complex non-functional requirements. We also provide optimized scheduling algorithms for solving this problem. The task model taken as input by these algorithms is that of Definition 2, augmented with the formal definitions of Section 3 (architecture, partitions, windows, MTF) and the non-functional property definitions of Section 4.2 (release dates, deadlines). The output of the algorithms is a scheduling table, formally defined in Section 6.1. These formalisms and algorithms, whose presentation takes most of the paper, are represented with solid boxes in Figure 1.

The remaining (dashed) boxes represent implementation steps that are needed to ensure the practical applicability of our work, but which either bring no scientific contribution (*e.g.* hyper-period expansion) or which concern the initial modeling steps, which are not fully automatable. This is why we only dedicate them two subsections and use an example-driven presentation. We also use examples to explain how the formal constructs of the task model can be used to model various non-functional requirements arising in practical system design. For instance, in Section 4.2.1.4 we use our case study to show how release dates and deadlines can be used to model one type of architecture-related constraints (related to the size of the input buffers).

## 2 Related Work

The main originality of our work is to define a complex task model allowing the specification of all the functional and non-functional aspects needed for the efficient implementation of our case study. Of course, *prior work already considers all these functional and non-functional aspects, but either in isolation (one aspect at a time), or through combinations that do not cover our modeling needs.* Our contributions are the non-trivial combination of these aspects in a coherent formal model and the definition of synthesis algorithms able to build a running real-time implementation.

Previous work by Henzinger *et al.* [26, 25], Forget *et al.* [37], Marouf *et al.* [33], and Alras *et al.* [2] on the implementation of multi-periodic synchronous programs and the work by Blazewicz [7] and Chetto *et al.* [12] on the scheduling of dependent task systems have been important sources of inspiration. By comparison, our paper provides a general treatment of ARINC 653-like partitioning and of conditional execution, and a novel use of deadlines longer than periods to allow faithful real-time specification.

The work of Caspi *et al.* [11] addresses the multiprocessor scheduling of synchronous programs under bus partitioning constraints. By comparison, our work takes into account conditional execution and execution modes, allows preemptive scheduling, and allows automatic allocation of computations and communications. Taking advantage of the time-triggered execution context, our approach also relies on fixed deadlines (as opposed to relative ones), which facilitates the definition of fast mapping heuristics.

Another line of work on the scheduling of dependent tasks is represented by the works of Pop *et al.* [38] and Wei Zheng *et al.* [47]. In both cases, the input of the technique is a DAG, whereas our functional specifications allow the use of delayed dependencies between successive iterations of the DAG. Other differences are that the technique [47] does not take into account ARINC 653-like partitioning or conditional execution, and the technique of [38] does not allow the specification of complex end-to-end latency constraints. Fohler [20] does consider conditional control, but does so in a mono-processor, non-partitioned, non-preemptive context.

The off-line (pipelined) scheduling of tasks with deadlines longer than the periods has been previously considered (among others) by Fohler and Ramamritham [21], but this work does not consider, as we do, partitioning constraints and the use of execution conditions to improve resource allocation. This is also our originality with respect to other classical work on static scheduling of periodic systems [42].

Compared to previous work by Isovic and Fohler [27] on real-time scheduling for predictable, yet flexible real-time systems, our approach does not directly cover the issue of sporadic tasks, but allows a more flexible treatment of periodic (time-triggered) tasks. Based on a different representation of real-time characteristics and on a very general handling of execution conditions, we allow for better flexibility *inside* the fully predictable domain.

From an implementation-oriented perspective, Giotto [25, 26], ΨC [32], and Prelude [37, 41] make the choice of mixing a globally time-triggered execution mechanism with on-line priority-driven scheduling algorithms such as RM or EDF. By comparison, we made the choice of taking *all* scheduling decisions off-line. Doing this complicates the implementation process, but imposes a form of temporal isolation between the tasks which reduces the number of possible execution traces and increases timing precision (as the scheduling of one task no longer depends on the run-time duration of the others). In turn, this facilitates verification and validation. Furthermore, a fully off-line scheduling approach such as ours has the potential of improving worst-case performance guarantees by taking better decisions than a RM/EDF scheduler which follows an as-soon-as-possible (ASAP) scheduling paradigm. For instance, the transformations of Section 7 reduce the number of notoriously expensive partition changes by using a scheduling technique that is not ASAP. These partition changes are not taken into account in the optimality results concerning the EDF scheduling of Prelude [37].

Compared to classical work on the on-line real-time scheduling of tasks with execution modes (*cf.* [6]), our off-line scheduling approach comes with precise control of timing, causalities, and the correlation (exclusion relations) between multiple test conditions of an application. It is therefore more interesting for us to use a task model that explicitly represents execution conditions. We can then use *table-based scheduling* algorithms that precisely determine when the same resource can be allocated at the same time to two tasks because they are never both executed in a given execution cycle.

The use of execution conditions to allow efficient resource allocation is also the main difference between our work and the classical results of Xu [46]. Indeed, the exclusion relation defined by Xu does not model conditional execution, but resource access conflicts, thus being fundamentally different from the exclusion relation we define in Section 4. Our technique also allows the use of execution platforms with non-negligible communication costs and multiple processor types, as well as the use of preemptive tasks (unlike in Xu's paper).

The off-line scheduling on partitioned ARINC 653 platforms has been previously considered, for instance by Al Sheikh *et al.* [45] and by Brocal *et al.* in Xoncrete [8]. The first approach only considers systems with one task per partition, whereas our work considers the general case of multiple tasks per partition. The second approach (Xoncrete) allows multiple tasks per partition, but does not seem interested in having a functionally deterministic specification and preserving its semantics during scheduling (as we do). For instance, its input formalism does not specify periods, but ranges of acceptable periods, and the first implementation step adjusts these periods to reduce their lowest common multiple (thus changing the semantics[1]). Other differences are that our approach can take into account conditional execution and execution modes, and that we allow scheduling onto multi-processors, whereas Xoncrete does not.

More generally, our work is related to work on the scheduling for precision-timed architectures (*e.g.* [17]). Our originality is to consider complex non-functional constraints. The work on the

[1] Changing the relative periods directly impacts the communication scheme between the tasks, and thus modifies the semantics of the application

PharOS technology [32] also targets dependable time-triggered system implementation, but with two main differences. First, we follow a classical ARINC 653-like approach to temporal partitioning. Second, we take all scheduling decisions off-line. This constrains the system but reduces the scheduling effort needed from the OS, and improves predictability.

References on time-triggered and partitioned systems, as well as scheduling of synchronous specifications will be provided in the following sections.

## 3 Architecture Model

In this paper, we consider both single-processor architectures and bus-based multi-processor architectures with a globally time-triggered execution model and with strong temporal partitioning mechanisms. This class of architectures covers the needs of the considered case study, but also covers platforms based on the ARINC 653, TTA, and FlexRay (the static segment) standards.

Formally, for the scope of this paper, an architecture is a pair $\mathit{Arch} = (B(\mathit{Arch}), \mathit{Procs}(\mathit{Arch}))$ formed of a broadcast message-passing bus $B(\mathit{Arch})$ connecting a set of processors $\mathit{Procs}(\mathit{Arch}) = \{P_1, \ldots, P_n\}$ for some $n \geq 1$. We assume that the bus does not lose, create, corrupt, duplicate or change the order of messages it transmits. Previous work by Girault *et al.* [22] (among others) can be used to extend this simple model (and the algorithms of this paper) to deal with fault-tolerant architectures with multiple communication lines and more complex interconnect topologies. The architecture supports a time-triggered, partitioned execution model detailed below.

## 3.1 Time-triggered Systems

In this section we define the notion of time-triggered system used in this paper. It roughly corresponds to the definition given by Kopetz [28], and is a sub-case of the definition given by Henzinger and Kirsch [26]. We shall introduce its elements progressively, explaining what the consequences are in practice.

### 3.1.1 General Definition

By *time-triggered systems* we understand systems satisfying the following 3 properties:

**TT1** A system-wide time reference exists. We shall refer to this time reference as the *global clock*.[2] All timers in the system use the global clock as a time base.

**TT2** The execution duration of code driven by interrupts other than the timers (*e.g.* interrupt-driven driver code) is negligible. In other words, for timing analysis purposes, code execution is only triggered by timers synchronized on the global clock.

**TT3** System inputs are only read/sampled at timer triggering points.

This definition places no constraints on the sequential code triggered by timers. In particular:

- Classical sequential control flow structures such as *sequence* or *conditional execution* are permitted, allowing the representation of modes and mode changes.
- Timers are allowed to preempt the execution of previously-started code.

This definition of time-triggered systems is fairly general. It covers single-processor systems that can be represented with *time-triggered e-code programs*, as they are defined by Henzinger and

[2] For single-processor systems the global clock can be the CPU clock itself. For distributed multiprocessor systems, we assume it is provided by a platform such as TTA [29] or by a clock synchronization technique such as the one of Potop *et al.* [39].

Kirsch [26]. It also covers multiprocessor extensions of this model, as defined by Fischmeister et al. [19] and used by Potop et al. [39]. In particular, our model covers time-triggered communication infrastructures such as TTA and FlexRay (static and dynamic segments) [29, 44], the periodic schedule tables of AUTOSAR OS [5], as well as systems following a multi-processor periodic scheduling model without jitter and drift.[3] It also covers the execution mechanisms of the avionics ARINC 653 standard [3] provided that interrupt-driven data acquisitions, which are confined to the ARINC 653 kernel, are presented to the application software in a time-triggered fashion satisfying property *TT3*. One way of ensuring that *TT3* holds is presented in [35], and to our knowledge, this constraint is satisfied in all industrial settings.

### 3.1.2 Model Restriction

The major advantage of time-triggered systems, as defined above, is that they have the property of *repeatable timing* [16]. Repeatable timing means that for any two input sequences that are identical in the large-grain timing scale determined by the timers of a program, the behaviors of the program, including timing aspects, are identical. Of course, this ideal property must be amended to take into account the fact that the global clock may not be very accurate or that interrupt-driven driver code does take time and influences the execution of time-triggered code. In practice, however, repeatability can be ensured with good precision. Repeatability is extremely valuable in practice because it largely simplifies debugging and testing of real-time programs. A time-triggered platform also insulates the developer from most problems stemming from interrupt asynchrony and low-level timing aspects.

However, the applications we consider have even stronger timing requirements, and must satisfy a property known as *timing predictability* [16]. Timing predictability means that formal timing guarantees covering *all* possible executions of the system should be computed off-line by means of (static) analysis. The general time-triggered model defined above remains too complex to allow the analysis of real-life systems. To facilitate analysis, this model is usually restricted and used in conjunction with WCET analysis of the sequential code fragments.

In this paper we consider a restriction of the general definition provided above. In this restriction, timers are triggered following a fixed pattern which is repeated periodically in time. Following the convention of ARINC 653, we call this period the *major time frame (MTF)*. The timer triggering pattern is provided under the form of a set of fixed offsets $0 \leq t_1 < t_2 < \ldots < t_m < MTF$ defined with respect to the start of each $MTF$ period. Note that the code triggered at each offset may still involve complex control, such as conditional execution or preemption.

This restriction corresponds to the classical definition of time-triggered systems by Kopetz [28, 29]. It covers our target platform, TTA, FlexRay (the static segment), and AUTOSAR OS (the periodic schedule tables). At the same time, it does not fully cover ARINC 653. As defined by this standard, partition scheduling is time-triggered in the sense of Kopetz. However, the scheduling of tasks inside partitions is not, because periodic processes can be started (in normal mode) with a release date equal to the current time (not a predefined date). To fit inside Kopetz's model, an ARINC 653 system should not allow the start of periodic processes *after* system initialization, *i.e.* in normal mode.

[3] But these two notions must be accounted for in the construction of the global clock [39].

## 3.2 Temporal Partitioning

Our target architectures follow a strong temporal partitioning paradigm similar to that of ARINC 653.[4] In this paradigm, both system software and platform resources are *statically* divided among a finite set of *partitions* $Part = \{part_1, \dots, part_k\}$. Intuitively, a partition comprises both a software application of the system and the execution and communication resources allocated to it. The aim of this static partitioning is to limit the functional and temporal influence of one partition on another. Partitions can communicate and synchronize only through a set of explictly-specified inter-partition channels.

We are mainly concerned in this paper with the execution resource represented by the processors. To eliminate timing interference between partitions running on a processor, the static partitioning of the processor time is done using a static time division multiplexing (TDM) mechanism. In our case, the static TDM mechanism is built on top of the time-triggered model of the previous section. It is implemented by partitioning, separately for each processor $P_i$, the *MTF* defined above into a finite set of non-overlapping *windows* $W_i = \{w_i^1, \dots, w_i^{k_i}\}$. Each window $w_i^j$ has a fixed start offset $tw_i^j$, a duration $dw_i^j$, and it is either allocated to a single partition $partw_i^j$, or left unused. Unused windows are called spares and identified by $partw_i^j = spare$.

Software belonging to partition $part_i$ can only be executed during windows belonging to $part_i$. Unfinished partition code will be preempted at window end, to be resumed at the next window of the partition. There is an implicit assumption that the scheduling of operations inside the *MTF* will ensure that non-preemptive operations will not cross window end points.

For the scheduling algorithms of Section 6, the partitioning of the *MTF* into windows can be either an input or an output. More precisely, all, none, or part of the windows can be provided as input to the scheduling algorithms.

## 3.3 Example

To present the type of partitioned time-triggered system we synthesize and the underlying execution mechanisms, we rely on the example of Figure 2. This example will gradually introduce conditional execution and deterministic mode changes, pre-computed preemptions, and the pipelined execution of operations (operations that start in one *MTF* and complete in the next), which poses significant problems, especially when combined with conditional execution. The formal definitions supporting the intuitive presentation of the example will be provided in the following sections.

Our system has two processors, denoted P1 and P2, which are connected *via* a bus. As explained above, execution is periodic. Our figure provides the execution pattern for one period, under the form of a scheduling table. A full execution of the system is obtained by indefinitely repeating this pattern. In our case, the length of the scheduling table, which gives the MTF (global period) of the system is 40 milliseconds (ms).

The scheduling table defines the start dates of all the operations that can be executed during one MTF, be them computations or communications. For operations that can be preempted, it also defines all the dates where they are preempted and resumed (all these dates are pre-computed off-line). Finally, our table also defines the worst-case end dates of the operations. All dates are defined in the time reference of the global clock provided by the time-triggered platform. In our example, this clock (pictured at the bottom of the figure) has a precision of 1 ms. For instance, operation $F_1$ starts in each MTF at date 0 and ends (in the worst case) at date 8. Synchronization of operations with respect to the global clock also ensures the correct synchronization between operations. For

[4] Spatial partitioning aspects are not covered in this paper.

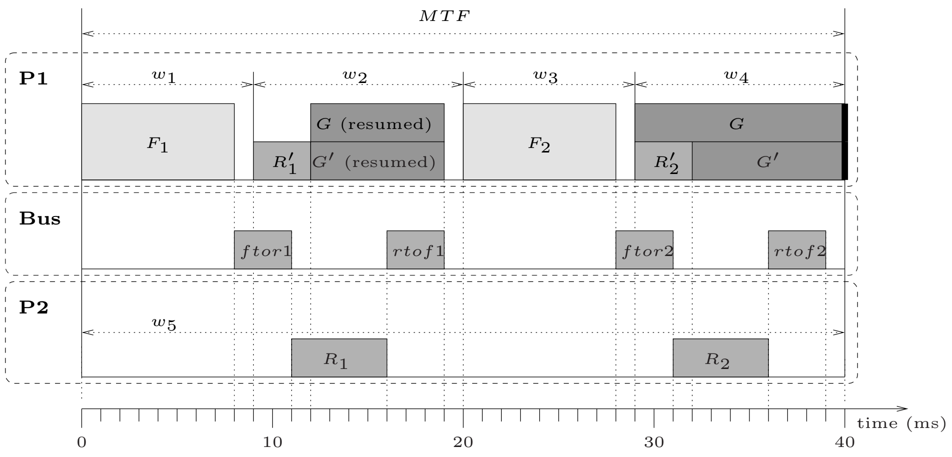

**Figure 2** An example of time-triggered partitioned system. In this example only processor P1 requires partitioning.

instance, $F_1$ ending before date 8 allows the synchronization with the communication *ftor1* which transmits a value from $F_1$ to $R_1$.

To comply with Kopetz' time-triggered model, the start/preempt/resume dates of all operations are computed off-line. For instance, on processor P1 operations can only start at dates 0, 9, 20, 29, and 32. However, code triggered at each offset may involve complex control such as conditional execution or preemption. Our example features both. Conditional execution is used here to represent execution modes. There are 2 modes. In *mode1* are executed $F_1$, $F_2$, $G$, $R_1$, $R_2$ and the bus communications. The other mode, named *mode2*, is meant to be activated when processor $P2$ is removed from the system due to a hardware reconfiguration. In this mode, operations $F_1$ and $F_2$ remain unchanged, but the other are replaced with the lower-duration counterparts $G'$, $R'_1$, and $R'_2$ (and no communication operations).

In our example, mode changes occur at the beginning of the MTF. Only operations belonging to the currently-active mode can be started during the MTF. Operations of different modes can be scheduled on the same processor at the same dates, as is the case for $G$ and $G'$ and $G$ and $R'_2$. This form of *double reservation* is forbidden if the operations belong to the same mode because our resources (processors and buses) are sequential.

Preemption is needed in the scheduling of long tasks and in dealing with partitioning. In our example, we assumed that the computation operations are divided in 2 partitions: *part1* contains $F_1$ and $F_2$, and *part2* all the other operations. During each MTF, the processor time of P1 is statically divided in 4 windows. Windows $w_1$ and $w_3$ are allocated to *part1*, and windows $w_2$ and $w_4$ are allocated to *part2*. To simplify the presentation of the algorithms we assume in this paper that bus time is not partitioned. Processor P2 only executes operations of *part2*, so that its MTF contains one window $w_5$ spanning over the entire MTF.

Window $w_4$ starts at date 29 and ends at date 40. Operation $G$ starts at date 29, but its worst-case duration is 18 ms, longer than the 11 ms of $w_4$. Since window $w_1$ belongs to another partition, operation $G$ must be pre-empted at the end of $w_4$, and it can be resumed at date 12 (within $w_2$) in the next MTF. Our scheduling table therefore contains a pre-computed preemption (represented with a thick black bar at the end of $G$ in $w_4$), and the execution time of $G$ is divided between 2 windows. Similarly, the execution of $G'$ is divided in two by a precomputed preemption.

An important hypothesis in our work is that the scheduling table specifies not only the start date of an operation, but also the dates where an operation can resume after a pre-computed preemption. For instance, operations $G$ and $G'$ resume at date 12, not at date 9. This hypothesis facilitates scheduling. In time-triggered operating systems such as ARINC 653, where only start dates are specified, this hypothesis can be easily implemented by using simple manipulations of task priorities.[5]

Another important hypothesis we make is that once an operation is started, it must be completed. In other words, it can be preempted and resumed, but it cannot be aborted. This hypothesis is key in ensuring the predictability and determinism of our systems. Intuitively, an operation can change the state of sensors, actuators, or internal variables, and it is difficult to determine the system state after an abortion operation.

But the absence of abortion requires much care in dealing with conditional execution and execution mode changes. Assume, for instance, that operation $G$ is started during one MTF, and that the mode changes from *mode1* to *mode2* at the end of that MTF. Then, the scheduling of the next MTF must allow $G$ to resume and complete while ensuring that the operations of *mode2* can be started. This explains why $R_1'$ and the resumed part of $G$ cannot be scheduled at the same date, even though $G$ and $R_1'$ belong to different modes.

## 4 Task Model

Following classical industrial design practices, the specification of our scheduling problem is formed of a *functional specification* and a set of *non-functional properties*. Represented in an abstract fashion, these two components form what is classically known as a *task model* which allows the definition of our scheduling algorithms.

### 4.1 Functional Specification

Our scheduling technique works on *deterministic* functional specifications of dataflow synchronous type,[6] such as those written in SCADE/Lustre [11]. These formalisms, which are common in the design of safety-critical embedded control systems, allow the representation of *dependent task systems* featuring *multiple execution modes*, *conditional execution*, and *multiple relative periods*.

However, a full presentation of all the details of our formalism would only complicate the presentation of this paper. Instead, we define here a task model containing just the formal elements needed to define our scheduling algorithms, and then explain using intuitive examples how an abstract task system is obtained from a data-flow synchronous specification. The full formal definition of the synchronous formalism used by our tool and of its relation with other formalisms used in embedded systems design and real-time scheduling is provided in [9].

#### 4.1.1 Single-period Dependent Task Systems

We define our task model in two steps. The first one covers systems with a single execution mode:

▶ **Definition 1** (Non-conditioned dependent task system). A non-conditioned dependent task system is a directed graph defined as a triple $D = (T_D, A_D, \Delta_D)$. Here, $T_D$ is the finite set of tasks. The finite set $A_D$ contains dependencies of the form $a = (src_a, dst_a, type_a)$, where $src_a, dst_a \in T_D$

[5] By using helper tasks that are executed at the time triggering points, and which raise or lower the priority of the other tasks to ensure that the greatest priority belongs to the one that must be executed (started or resumed) in the following time slot.

[6] Dataflow synchronous formalisms should not be confused with Lee and Messerschmitt's Synchronous Data Flow (SDF) [30] and derived models. For instance, data-dependent conditional execution cannot be faithfully represented in SDF.

are the source, respectively the destination task of $a$, and $type_a$ the type of the data messages transmitted from the source to the destination (identified by a name).[7] The directed graph determined by $A_D$ must be acyclic. The finite set $\Delta_D$ contains *delayed* dependencies of the form $\delta = (src_\delta, dst_\delta, type_\delta, depth_\delta)$, where $src_\delta$, $dst_\delta$, $type_\delta$ have the same meaning as for regular dependencies and $depth_\delta$ is a strictly positive integer called the *depth* of the dependency[8].

Non-conditioned dependent task systems have a cyclic execution model. At each execution cycle of the task system, each of the tasks is executed exactly once. We denote with $t^n$ the instance of task $t \in T_D$ for cycle $n$. The execution of the tasks inside a cycle is partially ordered by the dependencies of $A_D$. If $a \in A_D$ then the execution of ${src_a}^n$ must be finished before the start of ${dst_a}^n$, for all $n$. Note that dependency types are explicitly defined, allowing us to manipulate communication mapping.

The dependencies of $\Delta_D$ impose an order between tasks of successive execution cycles. If $\delta \in \Delta_D$ then the execution of ${src_\delta}^n$ must complete before the start of ${dst_\delta}^{n+depth_\delta}$, for all $n$.

We make the assumption that a task has no state unless it is explicitly modeled through a delayed arc. This assumption is a semantically sound way of providing more flexibility to the scheduler. Indeed, assuming by default that all tasks have an internal state (as classical task models do) implies that two instances of a task can never be executed in parallel. Our assumption does not imply restrictions on the way systems are modeled. Indeed, past and current practice in synchronous language compilation already relies on separating state from computations for each task, the latter being represented under the form of the so-called *step function* [4]. Thus, existing algorithms of classical synchronous compilers can be used to put high-level synchronous specifications into the form required by our scheduling algorithms.[9]

Definition 1 is similar to classical definitions of dependent task systems in the real-time scheduling field [12], and to definitions of data dependency graphs used in software pipelining [1, 13].

But we need to extend this definition to allow the efficient manipulation of specifications with multiple execution modes. The extension is based on the introduction of a new *exclusion relation* between tasks, as follows:

▶ **Definition 2** (Dependent task system). A dependent task system is a tuple $D = (T_D, A_D, \Delta_D, EX_D)$ where $\{T_D, A_D, \Delta_D\}$ is a non-conditioned dependent task system and $EX_D$ is an exclusion relation $EX_D \subseteq T_D \times T_D \times \mathbb{N}$.

The introduction of the exclusion relation modifies the execution model defined above as follows: if $(\tau_1, \tau_2, k) \in EX_D$ then ${\tau_1}^n$ and ${\tau_2}^{n+k}$ are never both executed, for any execution of the modeled system and any cycle index $n$. For instance, if the activations of $\tau_1$ and $\tau_2$ are on the two branches of a test we will have $(\tau_1, \tau_2, 0) \in EX_D$. An example of how the exclusion relation works is given in Section 4.1.3.2.

The relation $EX_D$ is obtained by analysis of the execution conditions in the data-flow synchronous specification. A full-fledged definition and analysis of the data expressions used as execution conditions has been presented elsewhere [39] and would take precious space in this paper. The relation $EX_D$ needs not be computed exactly. Any sub-set of the exact exclusion relation between tasks can safely be used during scheduling (even the void sub-set). However, the more exclusions

[7] The data type is needed to determine the duration of data transmissions.

[8] Regular dependencies can be seen as delayed dependencies with depth 0. Nevertheless the two types of arcs correspond to different semantic constructs in the synchronous model we use, and their treatment in the scheduling flow is very different. We therefore use different definitions.

[9] This has already been done for a Lustre/Scade dialect [15] and for SynDEx specifications [9].

we take into account, the better results the scheduling algorithms will give because tasks in an exclusion relation can be allocated the same resources at the same dates.

### 4.1.2 Reduction of Multi-period Problems to Single-period Task Systems

Many scheduling problems involve multi-period task systems. For instance, the scheduling table of Figure 2 was obtained from a specification involving 5 tasks $F$, $R$, $G$, $R'$ and $G'$ whose periods are respectively 20ms, 20ms, 40ms, 20ms, and 40 ms. This information is usually classified as non-functional. However, the functional specification also depends on it. More precisely, without knowing the *ratio* between the periods of the tasks it is impossible to precisely define how the tasks exchange data, and therefore it is impossible to ensure the determinism of the functional specification.

Our formal model of dependent task system does not directly represent the relative periods of the tasks.[10] Instead, following the example of [47], we rely on the *hyperperiod expansion* detailed below to transform multi-period task systems into equivalent single period ones. By using this method, our approach does not lose generality while allowing us to focus on the treatment of the exclusion relation and the non-functional properties.

The *hyperperiod* of a task system is the least common multiple of the periods of its tasks/operations. For the task system defined above $\{F, R, G, R', G'\}$ the hyperperiod is 40 ms. For simplicity, we assume in this paper that the MTF of the implementation is equal to the hyperperiod of the task system.

Hyperperiod expansion is a classical operation of the scheduling theory [36, 43], which consists in replacing a multi-period task system with an equivalent one in which tasks have all the same period, equal to the hyperperiod of the initial task system. Hyperperiod expansion works by determining how many instances of the tasks in the initial system are needed to cover the hyperperiod. In our example, $F$ has period 20ms, so it must be replaced by 2 tasks $F_1$ and $F_2$, both of period 40 ms. Similarly, $R$ is replaced by $R_1$ and $R_2$ and $R'$ is replaced by $R'_1$ and $R'_2$. Tasks $G$ and $G'$ do not require replication because their periods are already equal to the hyperperiod.

Replication of tasks is accompanied by the replication of dependency arcs [36], which follows the same rules, but is more complicated due to the fact that an arc can connect tasks of different periods, and thus may involve under- or over-sampling (as it can be specified in languages such as Prelude or Giotto). The period-driven replication must infer which instances of the source and destination task must be connected through arcs.

The full hyperperiod expansion of our example, which allows the synthesis of the scheduling table of Figure 2, is pictured in Figure 3. Regular dependencies are represented here using solid arcs. Delayed dependencies are represented using dashed arcs. The label of a delayed dependency gives its depth. For instance, a regular dependency connects $F_1$ and $F_2$ to signify that inside a hyperperiod (MTF) $F_1$ must be executed before $F_2$. We did not graphically represent here the type information specified by our formal model, which determines the kind (and amount) of data that is passed from $F_1$ to $F_2$. The delayed dependency of depth 2 between $G$ and $F_1$ means that the instance of $G$ started in $MTF$ of index $n$ must be completed before $F_1$ is started in the $MTF$ of index $n + 2$, for all $n$. Note that task $F$ has a state, which is expanded into arcs connecting its instances $F_1$ and $F_2$, whereas the other tasks have no state.

[10] The *compact* representation of multi-period specifications, as well as its *efficient* manipulation for scheduling purposes is elegantly covered by Forget *et al.* [37], drawing influences from Chetto *et al.* [12] and Cohen *et al.* [14], among others.

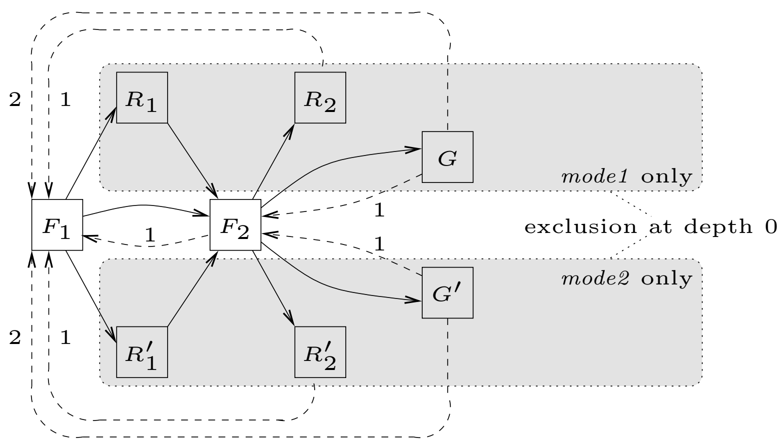

**Figure 3** Example of dependent task system.

```
process MTFfunction()
(| mode := Fmode2 $1 init true
 | (Fs1,Fmode1,ftor1,ftog1) := F(Fs2 $1 init K0, rtof1 $1 init K1, gtof $2 init K2) %F1%
 | (Fs2,Fmode2,ftor2,ftog2) := F(Fs1, rtof2, gtof $1 init K2)                     %F2%
 | rtof1 := R(ftor1 when mode) default Rprime(ftor1 when not mode)       %R1 and R1prime%
 | rtof2 := R(ftor2 when mode) default Rprime(ftor2 when not mode)       %R2 and R2prime%
 | gtof  := G(ftog when mode) default Gprime(ftog when not mode)           %G and Gprime%
 |)
 where mode,Fmode1,Fmode2:boolean ; F1s,F2s:FStatetype ;                %state variables%
       ftor1,ftor2:FtoRtype ; rtof2,rtof1:RtoFtype ;
       ftog1,ftog2:FtoGtype ; gtof:GtoFtype ;
 end;
```

**Figure 4** Synchronous program corresponding to the task system of Figure 3.

Figure 3 also emphasizes the use of exclusion relations. In our example, an exclusion relation of depth 0 relates $\tau$ and $\tau'$ for all $\tau \in \{R_1, R_2, G\}$ and $\tau' \in \{R'_1, R'_2, G'\}$. We have graphically represented these relations with a relation between the two systems of tasks that are activated in only one of the two modes. Tasks $F_1$ and $F_2$ belong to both modes.

As explained above, dependent task systems are the abstract model containing just the formal elements needed to define our scheduling algorithms. But our tools work on full-fledged dataflow synchronous programs defining all details needed to allow executable code generation. Figure 4 provides a data-flow synchronous program corresponding to the dependent task system in Figure 3. This program is written in the Signal/Polychrony language [24], which our tool can take as input. In our simple case, the correspondence between elements of the synchronous program and the dependent graph formalism is straightforward: calls to $R$, $R'$, $F$, and $G$ become the tasks of the data-flow graph, data-flow dependencies become the arcs, and the delays of depth 1 and 2, identified with `$1` and `$2`, become the delayed dependencies (with the same depth). The main difference between the two description levels is that the synchronous program defines the exact execution conditions (not just the exclusions) and all the initial values defining the initial state of the system.

### 4.1.3 Modeling of the Aerospace Case Study

The specification of the space flight application was provided under the form of a set of AADL [18] diagrams, plus textual information defining specific inter-task communication patterns,

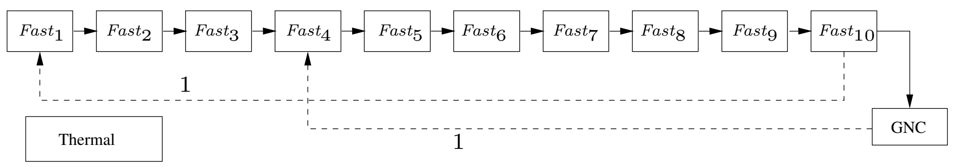

**Figure 5** The *Simple* example.

determinism requirements, and a description of the target hardware architecture. In this section we use the simpler version of the specification (with fewer tasks), the results on the full example being provided in Section 7.

Our first step was to derive a task model in our formalism. This modeling phase showed that the initial system was over-specified, in the sense that real-time constraints were imposed in order to ensure causal ordering of tasks instances using AADL constructs. Removing these constraints and replacing them with (less constraining) data dependencies gave us more freedom for scheduling, allowing for a reduction in the number of partition changes. The resulting specification is presented in Figure 5.

Our model, named *Simple* represents a system with 3 tasks *Fast*, *GNC*, and *Thermal*. The periods of the 3 tasks are 10ms, 100ms, and 100ms, respectively, meaning that *Fast* is executed 10 times for each execution of *GNC* and *Thermal*. The hyperperiod expansion described in Section 4.1.2 replicates task *Fast* 10 times, the resulting tasks being $Fast_i$, $1 \leq i \leq 10$. Tasks *GNC* and *Thermal* are left unchanged because their period equals the hyperperiod. The direct arcs connecting the tasks $Fast_i$ and *GNC* represent regular (intra-cycle) data dependencies of $A_{Simple}$. Delayed data dependencies of depth 1 represent the transmission of information from one MTF to the next. In this simple model, task *Thermal* has no dependencies.

#### 4.1.3.1 From Non-determinism to Determinism

The design of complex embedded systems usually starts with *sets of requirements* allowing multiple implementations. Space launchers are no exception to this rule. For instance, the requirements for example *Simple* do not impose the presence of a delayed arc connecting *GNC* to $Fast_4$. Instead, they require that there exists an $i$ such that the feedback from *GNC* to $Fast_i$ is performed under a given latency constraint.

But determinism has its advantages: As explained in Section 3.1.2, it largely simplifies debugging and testing. Moreover, the determinism of the functional specification allows for a significant decoupling of software development, including verification and validation steps, from allocation and scheduling choices.

This is why a deterministic functional specification is built early in the development process of space launchers. The first step in this direction is made when Giotto-like rules [25] are used to build a fixed set of data dependencies, thus creating a deterministic functional specification. These rules only depend on the number of tasks before hyperperiod expansion, their relative periods, and a coarse view on the flow of data from task to task. This information can be easily recovered from the requirements and from early implementation choices.

The Giotto-like rules allow the fast construction of a deterministic specification and are easy to understand and implement. However, they do not take into account latency constraints, nor task durations, and thus the initial functional specification may not allow real-time implementation. In our example, the initial dependency pattern did not allow the respect of the latency constraint on the feedback from *GNC* to *Fast*. When this happens, the dependencies are modified manually within the limits fixed by the requirements to allow real-time implementation.

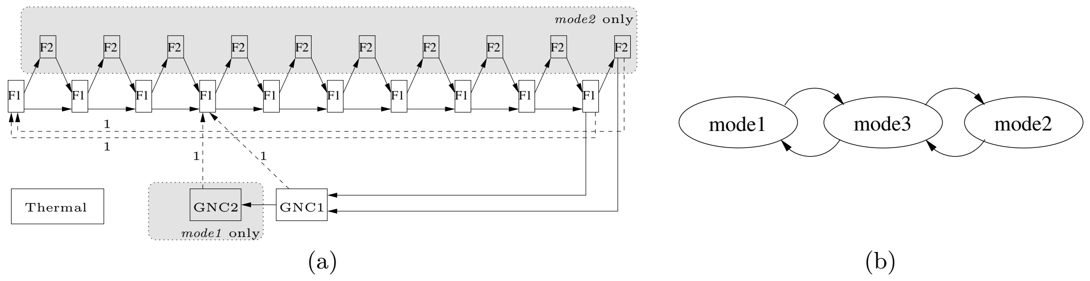

**Figure 6** Example 3: Dataflow specification with conditional execution (a) and possible mode transitions (b).

But these manual modifications come at a cost, especially when they must be done late in the design flow, requiring the re-validation of the whole design. This cost can be acceptable when the system is first built, which means once every 20 years or so. But once a deterministic implementation is built, it is strongly desired that subsequent *modifications* of the system preserve unchanged the functionality of system parts that are not modified (and in particular their determinism). If this is possible during the system modification, then the confidence in the modified system is improved and less effort is needed for the re-validation of the system.

In other words, design choices made during the initial implementation become desired properties for subsequent modifications, where they are considered as part of the functional specification. In our example, we assume that the feedback from *GNC* to $Fast_4$ is such an implementation choice made in the initial implementation, and which we include in the functional specification.

Our algorithms and tool allow the scheduling of such deterministic specifications. For the cases where the requirements are non-deterministic, our algorithms and tool can be used to speed up an otherwise manual exploration of the possible design choices (but this exploration process is not covered in the current paper).

#### 4.1.3.2 Representing Execution Modes

The dependent task system of Figure 5 does not represent execution modes, implicitly assuming that for each task the scheduling will always use its worst-case execution time.

But a space launcher application does make use of conditional execution and execution modes, and the scheduling can be optimized by taking them into account. The difficulty is to allow scheduling in a way that takes into account modes and is also compatible with the execution mechanisms of the launcher. Recall that the number of tasks in the launcher is fixed – 3. Mode changes do not trigger here the start or stop of tasks, as in our example of Figure 2. Instead, they are encoded with changes of state variables that enable or disable the execution of various code fragments inside the 3 tasks.[11]

In this approach, the only macroscopic property of a task that changes depending on the mode and can therefore be exploited during scheduling is its duration. Representing mode-dependent durations using our formalism requires a non-trivial transformation, detailed through the example in Figure 6.

We assume that our system has 3 modes (1, 2 and 3), the mode 3 being a transition mode between 1 and 2, as shown in Figure 6(b). We assume that the duration of tasks *Fast* and *GNC*

[11] The state change code is also part of the tasks. It can include arbitrarily complex code, such as the step function of an explicit state machine.

depends on the mode. We denote with $WCET(\tau, P)_m$ the duration of task $\tau \in \{Fast, GNC\}$ on processor $P$ in mode $m \in \{1, 2, 3\}$. We assume that $WCET(Fast, P)_3 = WCET(Fast, P)_1 < WCET(Fast, P)_2$ and that $WCET(GNC, P)_3 = WCET(GNC, P)_2 < WCET(GNC, P)_1$ for all $P$. Then, our modeling is based on the use of 2 tasks for the representation of each of *Fast* and *GNC*. The first task represents the minimum task duration (WCET) of the two modes, whereas the second task represents the remainder, which is only needed in modes where the duration is longer.

The resulting model is pictured in Figure 6. Here, *Fast* has been split into *F1* and *F2*, the second one being executed only in mode 2. *GNC* has been split into *GNC1* and *GNC2*, the second one being executed only in mode 1. The mode change automaton ensures that $(GNC2, F2_i, 0) \in EX_{Simple}$ and $(GNC2, F2_i, 1) \in EX_{Simple}$ for $1 \leq i \leq 10$, a property that will be used by the scheduling algorithms of Section 6. In our example we assumed state change code is executed in the first instance of *F1*.

The method we intuitively defined here for tasks with 2 durations can be generalized. A task having $n$ durations depending on the mode will need an expansion into $n$ tasks.

## 4.2 Non-functional Properties

Our task model considers non-functional properties of 4 types: real-time, allocation, partitioning, and preemptability.

### 4.2.1 Period, Release Dates, and Deadlines

The initial functional specification of a system is usually provided by the control engineers, which must also provide a *real-time characterization* in terms of *periods*, *release dates*, and *deadlines*. This characterization is directly derived from the analysis of the control system, and does not depend on architecture details such as number of processors, speed, etc. The architecture may impose its own set of real-time characteristics. Our model allows the specification of all these characteristics in a specific form adapted to our functional specification model and time-triggered implementation paradigm.

#### 4.2.1.1 Period

Recall from the previous section that after hyper-period expansion all the tasks of a dependent task system $D$ have same period. We shall call this period the *major time frame* of the dependent task system $D$ and denote it $MTF(D)$. We will require it to be equal to the $MTF$ of its time-triggered implementation, as defined in Section 3.1.2.

Throughout this paper, we will assume that $MTF(D)$ is an input to our scheduling problem. Other scheduling heuristics, such as those of [39] can be used in the case where the $MTF$ must be computed.

#### 4.2.1.2 Release Dates and Deadlines

For each task $\tau \in T_D$, we allow the definition of a release date $r(\tau)$ and a deadline $d(\tau)$. Both are positive offsets defined with respect to the start date of the current MTF (period). To signify that a task has no release date constraint, we set $r(\tau) = 0$. To signify that it has no deadline we set $d(\tau) = \infty$.

The main intended use of release dates is to represent constraints related to input acquisition. Recall that in a time-triggered system all inputs are sampled. We assume in our work that these sampling dates are known (a characteristic of the execution platform), and that they are an input to our scheduling problem. This is why they can be represented with fixed time offsets. Under

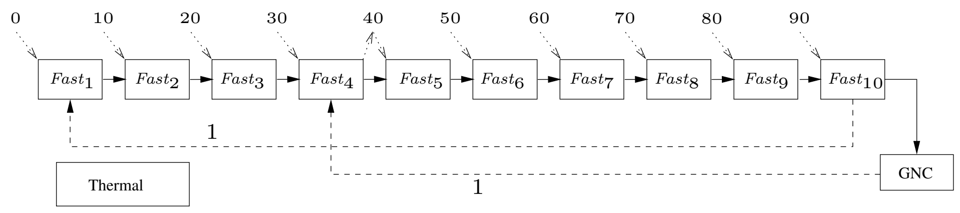

**Figure 7** Real-time characterization of the *Simple* example (MTF = 100 ms).

these assumptions, a task using some input should have a release date equal to (or greater than) the date at which the corresponding input is sampled. The inputs themselves (their values) are not explicitly represented (they are implicitly used by the task subjected to the associated release dates).

End-to-end latency requirements are specified using a combination of both release dates and deadlines. End-to-end latency constraints are defined on *flows*, which are chains of dependent task instances. Formally, a flow $\phi$ is a sequence of task instances $\tau_1^{k_1}, \tau_2^{k_2}, \ldots, \tau_m^{k_m}$ such that $\tau_i^{k_i}$ and $\tau_{i+1}^{k_{i+1}}$ are connected by a (direct or delayed dependency) for all $1 \leq i \leq m-1$. An end-to-end latency over flow $\phi$ requires that the duration between the start of $\tau_1^{n+k_1}$ and the end of $\tau_m^{n+k_m}$ is of less than a given duration $l(\phi)$ for all $n$. In this paper, we require that the end-to-end latency $l(\phi)$ is counted from the release date of first task instance $\tau_1^{n+k_1}$. This amounts to assuming that flows start with an input acquisition performed at the release date of the first task.

Under these assumptions, imposing the latency constraint $l(\phi)$ on $\phi$ is the same as imposing on the last task of $\phi$ ($\tau_m$) the deadline $l(\phi) + r(\tau_1) - (k_m - k_1) * MTF$.

Before providing an example, it is important to recall that our real-time implementation approach is based on off-line scheduling. The release dates and deadlines defined here are specification objects used by the off-line scheduler alone. These values have no direct influence on implementations, which are exclusively based on the scheduling table produced off-line. In the implementation, task activation dates are always equal to the start dates computed off-line, which can be very different from the specification-level release dates.

#### 4.2.1.3 Modeling of the Case Study

The specification in Figure 7 adds a real-time characterization to the *Simple* example of Figure 5. Here, $MTF(Simple) = 100$ ms. Release dates and deadlines are respectively represented with descending and mounting dotted arcs. The release dates specify that task *Fast* uses an input that is sampled with a period of 10ms, starting at date 0, which imposes a release date of $(n-1)*10$ for $Fast_n$. Note that the release dates on $Fast_n$ constrain the start of *GNC*, because *GNC* can only start after $Fast_{10}$. However, we do not consider these constraints to be a part of the specification itself. Thus, we set the release dates of tasks *GNC* and *Thermal* to 0 and do not represent them graphically.

Only task $Fast_4$ has a deadline that is different from the default $\infty$. In conjunction with the 0 release date on $Fast_1$, this deadline represents an end-to-end constraint of 140ms on the *flow* defined by the chain of dependent task instances

$$Fast_1{}^n \rightarrow Fast_2{}^n \rightarrow \ldots \rightarrow Fast_{10}{}^n \rightarrow GNC^n \rightarrow Fast_4{}^{n+1}$$

for $n \geq 0$. Under the notation for task instances that was introduced in Section 4.1.1, this constraint requires that no more than 140ms separate the start of the $n^{th}$ instance of task $Fast_1$ from the end of the $(n+1)^{th}$ instance of task $Fast_4$. Since the release date of task instance $Fast_1{}^n$ in the *MTF* of index $n$ is 0, this flow constraint translates into the requirement that $Fast_4{}^{n+1}$

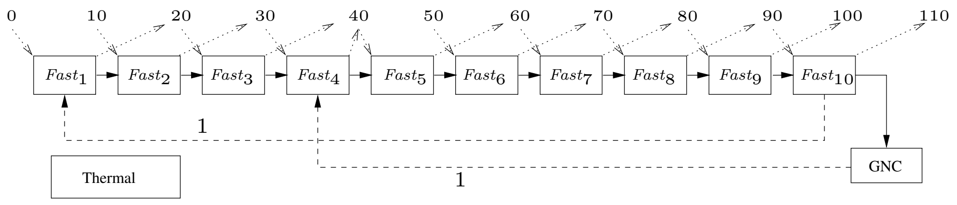

**Figure 8** Adding 3-place circular buffer constraints to our example.

terminates 140ms after the beginning of the *MTF* of index $n$. This is the same as 40ms after the beginning of *MTF* of index $n + 1$ (because the length of one *MTF* is 100ms). The deadline of $Fast_4$ is therefore set to 40ms.

#### 4.2.1.4 Architecture-dependent Constraints

The period, release dates and deadlines of Figure 7 represent architecture-independent real-time requirements that must be provided by the control engineer. But architecture details may impose constraints of their own, to be modeled using release dates and deadlines.

We provide here only one such example taken from the case study: Assume that the samples used by task *Fast* are stored in a 3-place circular buffer. At each given time, *Fast* uses one place for input, while the hardware uses another to store the next sample. Then, to avoid buffer overrun, the computation of $Fast_n$ must be completed before date $(n + 1) * 10$, as required by the new deadlines of Figure 8. Note that these deadlines can be both larger than the period of task *Fast*, and larger than the *MTF* (for $Fast_{10}$). By comparison, the specification of Figure 7 corresponds to the assumption that input buffers are infinite, so that the architecture imposes no deadline constraint. Also note in Figure 8 that the deadline constraint on $Fast_3$ is redundant, given the deadline of $Fast_4$ and the data dependency between $Fast_3$ and $Fast_4$. Such situations can easily arise when constraints from multiple sources are put together, and do not affect the correctness of the scheduling approach.

Real-time requirements coming from the control engineers and those due to the architecture are represented using the same constructs: period, release dates, deadlines. By consequence, the scheduling algorithms of the following sections will make no distinction between them.

### 4.2.2 Worst-case Durations, Allocations, Preemptability

We also need to describe the processing capabilities of the various processors and the bus. More precisely:

- For each task $\tau \in T_D$ and each processor $P \in Procs(Arch)$ we provide the *capacity*, or *duration* of $\tau$ on $P$. We assume this value is obtained through a worst-case execution time (WCET) analysis, and denote it $WCET(\tau, P)$. This value is set to $\infty$ when execution of $\tau$ on $P$ is not possible.
- Similarly, for each data type $type_a$ used in the specification, we provide a worst-case communication time estimate $WCCT(type_a)$ as an upper bound on the transmission time of a value of type $type_a$ over the bus. We assume this value is always finite.

Note that the *WCET* information may implicitly define *absolute allocation constraints*, as $WCET(t, P) = \infty$ prevents $t$ from being allocated on $P$. Such allocation constraints are meant to represent hardware platform constraints, such as the positioning of sensors and actuators, or designer-imposed placement constraints. *Relative allocation constraints* can also be defined, under the form of *task groups* which are subsets of $T_D$. The tasks of a task group must be allocated on the same processor. Task groups are necessary in the representation of mode-dependent task durations, as presented in Section 4.1.3 (to avoid task migrations). They are also needed in the transformations of Section 5.

Our task model allows the representation of both preemptive and non-preemptive tasks. The preemptability information is represented for each task $\tau$ by the flag $\mathit{is\_preemptive}(\tau)$. To simplify the presentation of our algorithms, we make in this paper two simplifying assumptions: that bus communications are non-interruptible and that preemption and partition context switch costs are negligible.

### 4.2.3 Partitioning

Recall from Section 3.2 that there are two aspects to partitioning: the partitioning of the application and that of the resources (in our case, CPU time). On the application part, we assume that every task $\tau$ belongs to a partition $\mathit{part}_\tau$ of a fixed partition set $\mathit{Part} = \{\mathit{part}_1, \ldots, \mathit{part}_k\}$.

Also recall from Section 3.2 that CPU time partitioning, *i.e* the time windows on processors and their allocation to partitions can be either provided as part of the specification or computed by our algorithms. Thus, our specification may include window definitions which cover none, part, or all of CPU time of the processors. We do not specify a partitioning of the shared bus, but the algorithms can be easily extended to support a per-processor time partitioning like that of TTA [44].

## 5 Removal of Delayed Dependencies

The first step in our scheduling approach is the transformation of the initial task model specification into one having no delayed dependency. This is done by a modification of the release dates and deadlines for the tasks related by delayed dependencies, possibly accompanied by the creation of new helper tasks that require no resources but impose scheduling constraints. Doing this will allow in the next section the use of simpler scheduling algorithms that work on acyclic task graphs.

The first part of our transformation ensures that delayed dependencies only exist between tasks that will be scheduled on the same processor, so that associated communication costs are 0. Let $\delta \in \Delta_D$ and assume that $\mathit{src}_\delta$ and $\mathit{dst}_\delta$ are not forced by absolute or relative allocation constraints to execute on the same processor. Then, we add a new task $\tau^\delta$ to $D$. The source of $\delta$ is reassigned to be $\tau^\delta$, and a new (non-delayed) dependency is created between $\mathit{src}_\delta$ and $\tau^\delta$. Relation $EX_D$ is augmented to place $\tau^\delta$ in exclusion with all tasks that are exclusive with $\mathit{src}_\delta$, and at the same depths. Task $\tau^\delta$ is assigned durations of 0 on all processors where $\mathit{dst}_\delta$ can be executed, and $\infty$ elsewhere. Finally, a task group is created containing $\tau^\delta$ and $\mathit{dst}_\delta$[12].

The second part of our transformation performs the actual removal of the delayed dependencies. It does so by imposing for each delayed dependency $\delta$ that $\mathit{src}_\delta$ terminates its execution before the release date of $\mathit{dst}_\delta$. This is done by changing the deadline of $\mathit{src}_\delta$ to $r(\mathit{dst}_\delta) + \mathit{depth}_\delta * \mathit{MTF}(D)$ whenever this value is smaller than the old deadline. Clearly, doing this may introduce real-time requirements (deadlines) that were not part of the original specification, which in turn implies that the method is non-optimal (it is a heuristic meant to make the dependent task system acyclic).

Once delayed dependencies are removed, we recompute the deadlines of all tasks. The objective is to ensure that the deadline of a task reflects the urgency of all tasks following it. For instance, task $\mathit{GNC}$ of Figure 8 has no deadline of its own, but it must be finished before the start of $\mathit{Fast}_4$

[12] The graph is made acyclic by removing the arcs in $\Delta_D$. Nevertheless, code must be generated that corresponds to these arcs. This generated code creates a variable shared between the helper task and the destination task. These two tasks must therefore be placed on the same processor, which explains the grouping constraint.

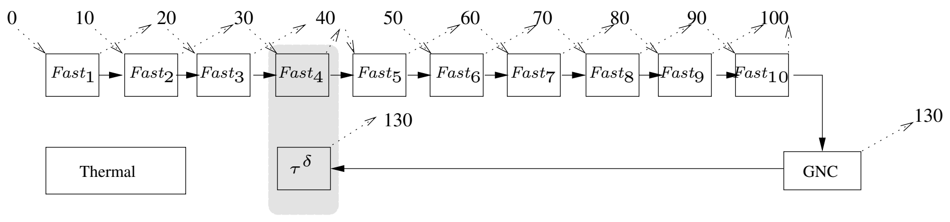

**Figure 9** Delay removal result for the example in Figure 8.

in the next cycle. Since the scheduling part of our process will be performed later, we do not know yet when $Fast_4$ will start. Still, we know it cannot start before date 30. Therefore we enforce that GNC has a deadline smaller or equal to the release date of $Fast_4$ in the next cycle (130).

For dependencies that do not span over multiple cycles, we use a form of deadline re-computation that is similar to the approach of Blazewicz [7], and Chetto *et al.* [12]. More precisely, the deadline of each task is changed into the minimum of all deadlines of tasks depending transitively on it (including itself), once the delayed dependencies have been removed.

The result of delayed dependency removal and deadline recomputation for the example in Figure 8 is pictured in Figure 9. We have assumed that tasks $Fast_4$ and $GNC$ can be allocated on different processors, and thus a helper task is needed. The new task group formed of tasks $\tau^\delta$ and $Fast_4$ is represented by the gray box. We assume that all tasks $Fast_n$ must be executed on the same processor, due to an absolute allocation constraint. This is why no helper task is needed when removing the delayed dependency from $Fast_{10}$ to $Fast_1$.

Note that the two transformations are another source of deadlines larger than the periods. Also note that all the transformations described above are linear in the size of the number of arcs (delayed or not), and thus very fast.

## 6 Offline Real-time Scheduling

On the transformed task models we apply an *offline scheduling* algorithm whose output is a system-wide *scheduling table* defining the allocation of processor and bus time to the various computations and communications. The length of this table is equal to the *MTF* of the task model.

Our offline scheduling algorithm is a significant extension of the one proposed by Potop *et al.* [40]. New features are the handling of *preemptive* tasks, *release dates* and *deadlines*, the *MTF*, and the *partitioning* constraints. The handling of conditional execution and bus communications remains largely unchanged, which is why we do not present these features in detail. Instead, we insist on the novelty points, like partitioning or the use of a deadline-driven criterion for choosing the order in which tasks are considered for scheduling. The deadline-driven criterion was inspired by existing work by Blazewicz [7] and by Chetto *et al.* [12]. By comparison with Blazewicz's works, our algorithm takes into account the *MTF*, the *partitioning* constraints, and *conditional execution.*

### 6.1 Scheduling Tables

As earlier explained, our algorithm computes a scheduling table. This is done by associating to each task:

- A *target processor* on which it will execute.
- A set of *time intervals* that will be reserved for its execution.
- A date of *first start.*

**Procedure 1 scheduler_driver**

```
Input:  $T_D$ : dependent task system
        $Arch$: architecture description
        $input\_schedule$ : schedule (complex data structure comprising a scheduling table
                           and a representation of the free intervals)
Output: $result\_schedule$ : schedule
  $result\_schedule := input\_schedule$
  while $T_D \neq \emptyset$ and $result\_schedule \neq invalid\_schedule$ do
    $\tau$:= choose_task_to_schedule($T_D$)
    $T_D$ := remove_task($\tau, T_D$)
    $new\_schedule := invalid\_schedule$
    $new\_cost := \infty$
    for all processor $P$ in $Archi$ do
      if $WCET(\tau, P) \neq \infty$ and group_ok($\tau$,$P$,$result\_schedule$) then
        $temp\_schedule$:= schedule_task_on_proc($\tau$,$P$,$result\_schedule$,$T_D$)
        if $temp\_schedule \neq invalid\_schedule$ then
          $temp\_cost$ := cost_function($temp\_schedule$,$\tau$)
          if $temp\_cost < new\_cost$ then
            $new\_schedule := temp\_schedule$
            $new\_cost := temp\_cost$
    $result\_schedule := new\_schedule$
  return $result\_schedule$
```

The conditional execution paradigm of our task model requires the use of *conditional reservations*: The same time interval can be reserved for two or more tasks if their execution conditions are mutually exclusive, as defined by relation $EX_D$. A similar reservation model is used for the bus.

Given a task system $D$, a scheduling table $S$ for $D$, and $\tau \in T_D$, we shall denote with $S.proc(\tau)$ the target processor of $\tau$, with $S.start(\tau)$ the date of first start, and with $S.intervals(\tau)$ the set of time intervals reserved for $\tau$. A time interval $i$ is defined by its start date $start(i)$ and end date $end(i)$. It is required that the intervals of $S.intervals(\tau)$ are disjoint, and that the start date of one of them is $S.start(\tau)$ **mod** $MTF(D)$.

Recall from Section 3.3 that the execution model is as follows: The $n^{th}$ instance of task $\tau$ will start (modulo conditional execution) at date $S.start(\tau) + (n-1) * MTF(D)$. Execution of the task is confined to its reserved time slots (it is suspended between such slots).

The choice of processor, start date, and intervals by the scheduling algorithm must ensure that:

- The intervals reserved for a task allow the complete execution of a task instance before the next instance is started.
- Intervals reserved for two tasks can only overlap if the two tasks belong to the same partition and have exclusive execution conditions. Moreover, an interval allocated to task $\tau$ of a partition *part* cannot overlap with windows allocated to other partitions.
- The task and communication execution order imposed by the direct and delayed dependencies is respected.
- The release date of a task precedes its start date, and deadline constraints are respected.

## 6.2 Scheduling Algorithm

The scheduling algorithm, whose top-level routine is Procedure 1, follows a classical list scheduling approach. It works by iteratively choosing a new task to schedule and then scheduling it along with the necessary communications.

**Procedure 2 schedule_task_on_proc**

```
Input: $\tau$ : task to schedule
       $P$ : processor on which to schedule
       $input\_schedule$ : schedule (before adding $\tau$)
       $D$ : dependent task system
Output: $result\_schedule$ : schedule (after adding $\tau$)
  $(result\_schedule, d_{earliest})$ := schedule_bus_communications($\tau, P, input\_schedule, D$)
  if $d_{earliest} + WCET(\tau, P) > d(\tau)$ then
    $result\_schedule := invalid\_schedule$
  else
    $deadline := d(\tau)$
    $needed\_duration := WCET(\tau, P)$
    $failure := false$
    /*We start our exploration at the start of the MTF that contains $d_{earliest}$*/
    $iteration := \lfloor d_{earliest}/MTF \rfloor$
    $d_{earliest} := d_{earliest} \bmod MTF$
    while $needed\_duration > 0$ and not $failure$ do
      /*Search for a new interval*/
      $(interval, result\_schedule)$:= get_first_interval($result\_schedule, needed\_duration,$
                                         $d(\tau), d_{earliest}, iteration, part_\tau, is\_preemptive(\tau)$)
      if $interval = invalid\_interval$ then
        /*No interval found, attempt to move the search into the next MTF*/
        if $deadline \leq MTF$ then
          $failure := true$
        else
          $deadline$:=$deadline$-$MTF(D)$
          $d_{earliest}$:=0
      else
        /*Good interval found*/
        $d_{earliest}$:=$end(interval)$
        $needed\_duration$:=$needed\_duration$-$len(interval)$
    if $failure$ then
      $result\_schedule := invalid\_schedule$
  return $result\_schedule$
```

Among the not-yet-scheduled tasks of whom all predecessors have been executed, function *choose_task_to_schedule*, not provided here, returns one of minimal deadline. If several tasks satisfy this criterion, then we determine for each of them the earliest start date in the current scheduling state, and we choose one with maximal earliest start date. For instance, the tasks in Figure 9 are chosen in the order $Fast_1, \ldots, Fast_{10}$, *GNC*, $\tau^\delta$, *Thermal*.

The body of the **while** loop allocates and schedules a single task $\tau$, along with the communications needed to gather the input data of $\tau$. It works by attempting to allocate and schedule $\tau$ on each of the processors that can execute it. Function *group_ok* determines if the relative allocation constraints and the current scheduling state allow $\tau$ to be allocated on $P$.

Among all the possible allocations of $\tau$, Procedure 1 chooses the one resulting in a – partial – schedule of minimal cost. In our case, *cost_function* chooses the schedule ensuring the earliest termination of $\tau$. If scheduling is not possible on any of the processors Procedure 1 returns *invalid_schedule* to identify the failure.

The mapping of a task $\tau$ onto a processor $P$ is realized through a call to *schedule_task_on_proc*, whose code is provided in Procedure 2. This procedure follows a classical ASAP (as soon as possible) scheduling strategy. The scheduling is done as follows. First, the transmission of data needed by $\tau$ and not yet present on $P$ is scheduled for communication on the bus using function *schedule_bus_communications*. This function schedules the transmission of both input data of $\tau$ and state variables needed to compute the execution condition of $\tau$. We do not provide the function here, interested readers being directed to [40].

Once communications are scheduled, we attempt to schedule the task at the earliest date *after* the date where all needed data is available. If this is not possible without missing the deadline, *invalid_schedule* is returned to identify the failure.

Looking for free intervals for the task to schedule is done by function *get_first_interval*, not provided here because it is too complex and because it requires explicit manipulations of Boolean predicates instead of the abstract exclusion relations $EX$.[13] For non-preemptive tasks, this function looks for the first free interval long enough to allow the execution of the task and satisfying the execution condition and partitioning constraints. For preemptive tasks, this function may be called several times to find the first free intervals satisfying the execution condition and partitioning constraints and of sufficient cummulated length to cover the needed duration. When unable to find a valid interval, this function returns *invalid_interval*. For instance, consider the scheduling of the example of Figure 3 to obtain the scheduling table of Figure 2, and assume that all tasks were scheduled save $G$. Procedure 1 will first attempt to schedule $G$ on processor $P1$. We assume that the partitioning of the processors is fixed as described in Section 3.3. Therefore, we are looking for time intervals where $P1$ is not used inside the $MTF$ windows $w_2$ and $w_4$. Search starts at the earliest start date of $G$, which is 29 (after $F_2$ terminates and $w_4$ starts). Given that $G$ is preemptible and that the execution condition of $G$ is exclusive with the execution conditions of $R_2'$ and $G'$, the first free interval is [29,39]. At the end of this interval $G$ must be preempted and resumed in the next execution iteration of the $MTF$ (if the other constraints allow it). In our example, resumption is possible at date 12. It is not possible at date 9, because the execution condition of $R_1'$ (the instance of the next execution cycle) is not exclusive with that of $G$.

When the partitioning of the $MTF$ is provided (fully or partially), this information is transmitted to Procedure 1 through parameter *input_schedule*. This initial state contains no task allocation, but may constrain the free interval set due to partitioning.

### 6.2.1 Complexity and Optimality Considerations

The complexity of Procedure 1 is linear in the number of tasks and in the number of processors in the architecture, and the complexity of Procedure 2 is sub-linear in the number of tasks (which bounds the number of calls to *get_first_interval*). But the real complexity of the scheduling algorithm is hidden inside function *get_first_interval*, which is called by Procedure 2. This function maintains a representation of free intervals. When working on dependent task systems without execution conditions and modes, its complexity is bounded by the number of tasks in the system, making for a globally polynomial complexity. But when execution conditions are taken into account, the representation of free intervals can grow in size exponentially. Moreover, determining if the execution condition of a free interval is compatible with that of a task requires solving instances of the Boolean satisfiability problem (of NP complete complexity). In practice, however, these execution conditions remain quite simple, and both SAT instances and the representation of free intervals remain small. **The duration of all the scheduling and code generation process was negligible in all our tests.**

From an optimality perspective, our scheduling algorithm is a safe heuristic that never provides an incorrect result, but may fail when a solution exists because it never reconsiders an allocation

[13] More informations on the handling of execution conditions can be found in references [40, 10].

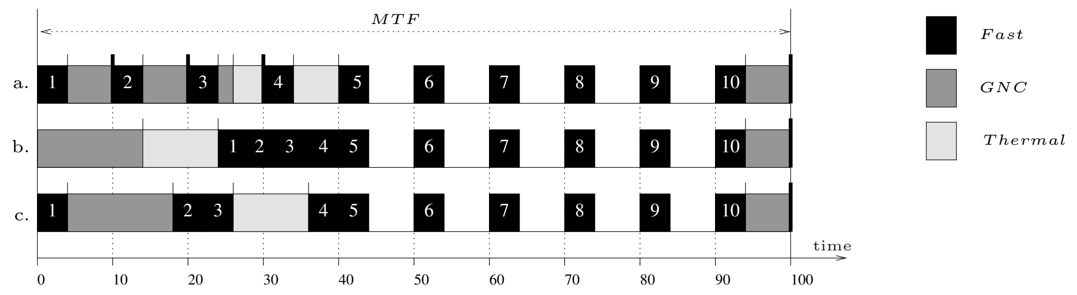

**Figure 10** Scheduling result for the examples in Figures 7, 8, and 9 on a single-processor architecture (a). The result of applying the post-scheduling slot optimization of Section 7 for the example in Figure 7 (b) and for the example in Figures 8 and 9 (c).

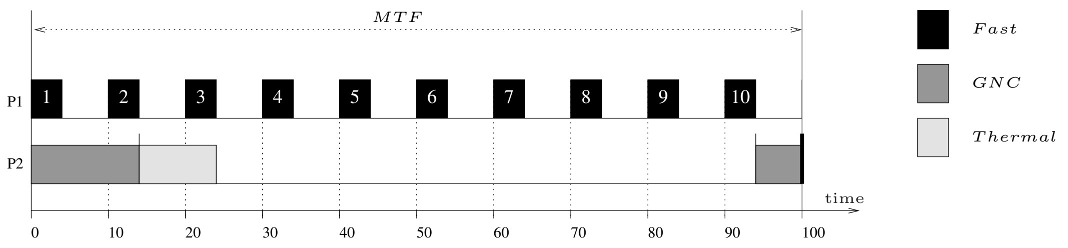

**Figure 11** Scheduling result for the example in Figure 7 on a two-processor architecture with zero communication costs.

or scheduling decision done for a task. Simple extensions of this algorithm would allow it to be optimal under restrictive hypotheses.[14] However, in the absence of extensive benchmarks it is unclear how optimality under restrictive hypotheses helps when scheduling problems involving a multiprocessor architecture, a complex control structure, and complex non-functional requirements. We therefore prefered here a compilation-like approach using low-complexity algorithms that can be easily tailored to take into account the previously-mentioned functional and non-functional properties.

As part of previous work [23, 10] we have also considered the use of integer linear programming (ILP) constraint solving engines (such as CPLEX, Yices, glpsol) to solve scheduling problems that were simpler than the ones we consider here (no conditional execution, no partitioning, *etc.*). We considered both optimization problems (*e.g.* finding a schedule of minimal makespan) and classical schedulability problems (finding one correct schedule respecting all deadlines). For single-period, non-preemptive, dependent task systems no makespan optimization problem could be solved beyond 15 tasks without using advanced symmetry-breaking techniques, which only apply for applications and architectures exhibiting significant regularity,[15] and significant numbers of timouts occurred beyond 10 tasks. Our results also show that schedulability solving for preemptive, multi-periodic systems of *non-dependent* tasks does not scale beyond 50 tasks, where the 1-hour timeout rate already reaches 50% (at 75% average system load). Given these experimental data, and the increase in problem complexity, we do not expect exact techniques to scale for the exact solving of the problem proposed in this paper beyond 20 tasks.

[14] Single processor, all tasks preemptive, no imposed partitioning, no execution conditions, zero communication and preemption costs. This is the same class of systems handled by Chetto *et al.* [12] and Forget *et al.* [37]. Obtaining optimality for this class of problems requires performing the off-line deadline-driven scheduling for more than one execution cycle of the specification, and thus increases complexity, as explained in [31].

[15] Only one application passed when using symmetry-breaking techniques.

## 6.3 Scheduling Results

We have implemented our scheduling algorithms into a tool, which allowed us to schedule our models of the space launcher application, a railway systems application [15], and some toy examples, like the one in Section 3.3. We present here only the results for the space launcher application.

We have started our evaluation with the reduced model defined by the dependent task system of Figure 7. In our model, all tasks were preemptible. Our first test performed the scheduling of this task system on an architecture with one processor ($P$). The durations of the tasks are $\mathit{WCET}(\mathit{Fast}, P) = 4$, $\mathit{WCET}(\mathit{GNC}, P) = 20$, and $\mathit{WCET}(\mathit{Thermal}, P) = 10$. We assumed that the 3 tasks have each its own partition. We also assumed that the partitioning of the *MTF* was not constrained (it was fully synthesized by our tool). The result of the scheduling phase is provided in Figure 10(a). Like in our example of Section 3.3, the partitioning of the *MTF* into windows is represented by solid vertical bars. Partition changes are set only at the beginning of an interval when this interval is allocated to a task and the previous allocated interval belongs to another partition. In our example, there are 11 partition changes (counting the final one at the end of the *MTF*). Following the graphical convention of Section 3.3, thicker partition separation bars represent partition changes where a task undergoes a precomputed preemption. There are 4 of them in our example. Note how function *get_first_interval* loops over the *MTF* in its search for reservations for tasks *GNC* and *Thermal*. For instance, the scheduling of *GNC* is realized after the one of the $\mathit{Fast}_i$ tasks, and its earliest start date is given by the end of $\mathit{Fast}_{10}$. After reserving the interval [95,100], the search loops over and reserves successively intervals [5,10], [15,20], and [25,26], in order to cover $\mathit{WCET}(\mathit{GNC}, P)$.

In our second test, we scheduled the same dependent task system on an architecture with two processors ($P1$ and $P2$) where inter-processor communication takes no time, such as in a shared memory system, when memory acces interferences are taken into account in the WCET analysis. We assumed that the two processors are identical and that the durations of the tasks on each processor are the ones provided above for the single-processor. We assumed there are no allocation constraints. The resulting scheduling table is provided in Figure 11. Only 4 partition changes remain (counting the mandatory 4 at the end of the *MTF*), and no pre-computed preemptions.

The third test considered the dependent task system of Figure 6, which features mode-dependent task durations. Scheduling is done here on a single processor $P$. We assumed that the durations of the various tasks in the example are: $\mathit{WCET}(F1, P) = 20$, $\mathit{WCET}(F2, P) = 20$, $\mathit{WCET}(\mathit{GNC}1, P) = 60$, $\mathit{WCET}(\mathit{GNC}2, P) = 120$, and $\mathit{WCET}(\mathit{Thermal}, P) = 100$. We also assumed that tasks *Fast* ($F1$ and $F2$) and *GNC* belong to one partition, and that task *Thermal* belongs to another. Again, we assumed that partitioning is fully synthesized. The resulting scheduling table is pictured in Figure 12. This example shows how taking into account execution conditions (even in the restricted form allowed by our space launcher application) allows double reservation and (in our example) ensure schedulability.

Finally, we have been able to schedule the large-scale model provided by Astrium. We have pictured in Figure 13 its architecture and the allocation of tasks to partitions and processors. The architecture is formed of 4 processors connected by a broadcast bus. There are 13 tasks divided in 7 partitions. Our figure also provides the periods and durations of the tasks. The *MTF* is 100ms. The tasks are statically allocated to processors. The direct communications between tasks are represented in Figure 13 with directed arcs.

As explained in Section 4.1.3, the task periods and the information flows definitions allow the construction of a fully deterministic functional specification by using Giotto-like communication rules.[16] For this large example, no manual modification of the dependencies was needed.

[16] The exact rules are the following: Direct communication is possible only between two tasks having harmonic

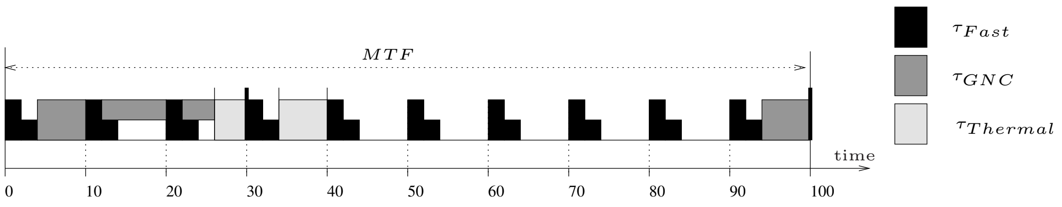

**Figure 12** Scheduling result for the two-mode example of Figure 6 on a single-processor architecture.

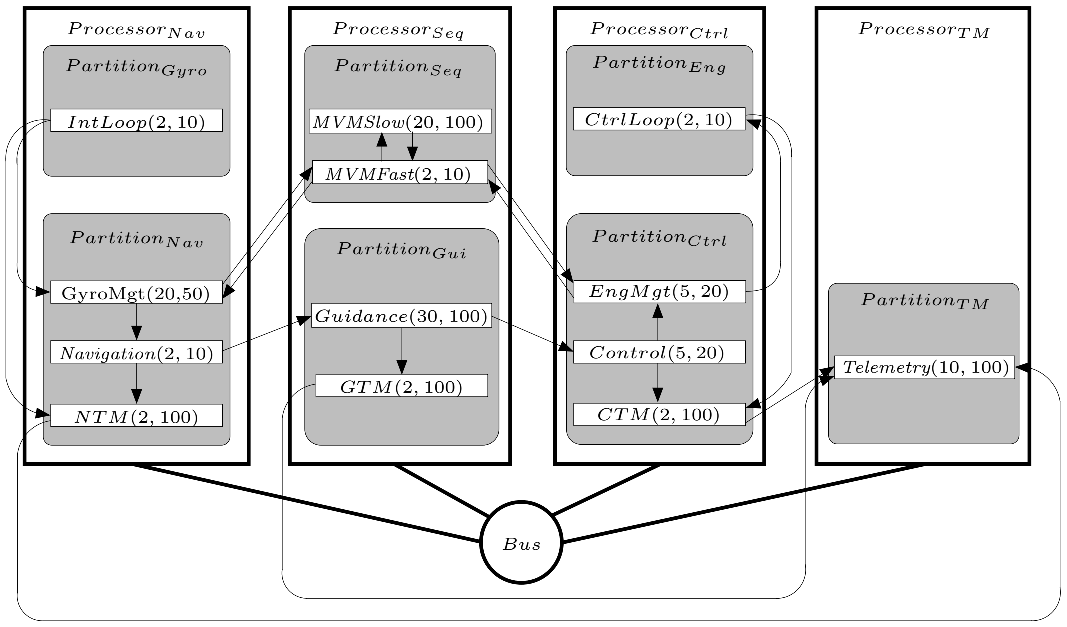

**Figure 13** Architecture, partitioning and task allocation for the large-scale model of the space launcher. The two integers inside each task define its duration and period (in milliseconds).

The specification defines ten flows (some of them ending with the same task), with end-to-end latency constraints ranging from 100ms to 450ms. Translated into our formalism they amount to the following constraints on the deadlines of tasks (after hyperperiod expansion): $d(GyroMgt_2) = 75$, $d(EngMgt_2) = 75$,$d(MVMSlow) = 50$, $d(CtrlLoop_{10}) = 90$, $d(Telemetry) = 100$,

Our tool built a correct scheduling table for this example, meaning that implementation is possible without manual changes to the dependencies. The 4 processors are loaded respectively at 82%, 72%, 72%, and 10% (the 4th processor is dedicated to telemetry). The bus is loaded at 81%.

## 7 Post-scheduling Slot Minimization

The algorithm of the previous section follows a classical ASAP deadline-driven scheduling policy, which is good for ensuring schedulability.

periods. In this case, the dependencies between task instances are determined as if the two tasks form a Giotto *mode* of period equal to the largest of the periods of the two tasks (*cf.* [25], figure 7). The only exception to this rule is when the two tasks belong to the same partition. In this case, if a dependency exists from the fast task to the slow one, then it is realized inside the *round*, between the first instance of the fast task and the instance of the slow task.

**Procedure 3** PostSchedulingOptimizationForMonoprocessor

```
Input:  $input\_schedule$ : schedule
Output: $result\_schedule$ : schedule (optimized)
  /* Initialisation of the interval lists. */
  $inputIntervalList$ := SortReservedIntervalsByIncreasingStartDate($input\_schedule$)
  $resultIntervalList$ := $empty\_list$
  repeat
    $I1$ := GetLastInterval($inputIntervalList$)
    $inputIntervalList$ := RemoveLastInterval($inputIntervalList$)
    $resultIntervalList$ := InsertInterval($I1$,$resultIntervalList$)
    /* Find in $inputIntervalList$ the last interval of the same partition as I1 */
    $I2$ := FindLast($inputIntervalList$,GetPartition($I1$))
    if $I2 \neq not\_found$ then
      /* Partition inputIntervalList around the start date of I2 (I2 is in
         neither interval). */
      ($intervalsBeforeI2$, $intervalsAfterI2$) := Partition($inputIntervalList$,GetStartDate($I2$))
      /* If there are no other intervals between I2 and I1, there is no partition change
         and therefore no need to move intervals. */
      if $intervalsAfterI2 \neq empty\_list$ then
        /* Attempt to move I2 after the intervals of intervalsAfterI2,
           if necessary also moving the intervals of intervalsAfterI2 earlier. */
        /* By how much can I2 be delayed if the intervals of
           intervalsAfterI2 are removed from the scheduling table? */
        $maxI2Delay$ := MaxI2Delay($I2$,$resultIntervalList$,GetLength($input\_schedule$))
        /* By how much can the intervals of intervalsAfterI2 be advanced
           if I2 is removed from the scheduling table? */
        $maxAdv$ := MaxAdvance($intervalsAfterI2$,$intervalsBeforeI2$,GetLength($input\_schedule$))
        if $start(I2) + maxI2Delay \geq \max_{i \in intervalsAfterI2}$ GetEndDate($i$) $+ maxAdv$ then
          /* It is possible to move I2 after intervalsAfterI2.
             Perform the move, interval by interval. */
          $I2$ := MoveInterval($I2$,$maxI2Delay$)
          $intervalsAfterI2$ := {MoveInterval($i$,$maxAdvance$)$|i \in intervalsAfterI2$}
          $inputIntervalList$ := Concatenate($intervalsBeforeI2$,$intervalsAfterI2$)
          $inputIntervalList$ := Append($inputIntervalList$,$I2$)
  until $inputIntervalList = empty\_list$
  /* Rebuild the scheduling table. */
  $resultSchedule$ := BuildScheduleFromList($resultIntervalList$)
```

However, resulting schedules may have a lot of unneeded preemptions and, most importantly, partition changes which are notoriously expensive. For instance, the scheduling table of Figure 10(a) features no less than 11 partition changes.

To reduce the number of partition changes, we perform a heuristic post-scheduling optimization of our scheduling tables. The algorithm we use in case of mono-processor architectures featuring no conditional execution is Procedure 3. Intuitively, the transformation we apply is the following: The scheduling table is traversed from end to the beginning. Whenever two intervals $I_1$ and $I_2$ allocated to tasks of the same partition are separated by intervals of other partitions, we attempt to group $I_1$ and $I_2$ together. Assuming $I_2$ starts before $I_1$, our technique attempts to move $I_2$ just before $I_1$ while moving all operations between $I_2$ and $I_1$ to earlier dates. The transformation step is only performed when the resulting schedule respects the correctness properties of Section 6.1. The complexity of this transformation is quadratic in the number of windows in the initial schedule.

The result of applying this algorithm on our simple example is provided in Figure 10(b). The number of partition changes is significantly reduced (from 11 to 3). Note that all instances of task *Fast* inside an $MTF$ are grouped in a single window inside the MTF. This is possible under the release date and deadline constraints of the dependent task system of Figure 7, where no architectural constraints are taken into account (we assumed that input buffers are infinite).

When the input buffering constrains are taken into account in the dependent task system of Figure 8, there is no change in the output of the scheduling algorithm. However, the supplementary constraints limit the efficiency of slot optimization, leading to the scheduling table of Figure 10(c), which has 6 partition changes. Note that our technique also reduces the number of preemptions. In our example, we move from 4 preemptions in the unoptimized example to only 1.

We have extended the previous algorithm to deal with applications featuring conditional execution (and modes) running on multi-processors, but the presentation of these extensions would go beyond the scope of this paper.

## 8 Conclusion and Future Work

Our objective here was to provide models and algorithms that facilitate the development of complex embedded systems by automating the allocation and scheduling step. The yardstick we used to measure our success was a model of a space launcher application.

The main originality of our work is that it takes into account at the same time multiple complexity elements of both functional and non-functional type. Our formalism allows the modeling of applications whose functional specification features dependent tasks, multiple periods and conditional execution (and modes). Our implementation targets are time-triggered partitioned multi-processor (bus-based distributed) architectures. Scheduling must be done under real-time, partitioning, and preemptability constraints.

Our off-line scheduling and optimization algorithms take into account these parameters and synthesize scheduling tables under complex non-functional requirements that no existing tool can handle. These scheduling tables can be automatically translated into time-triggered implementations.

We have been able to model and automatically map our aerospace case study. As all scheduling and optimization algorithms used in our tool are fast, we were able to perform significant design space exploration over the chosen case study. This shows the potential for full automation in the system-level engineering of complex real-time embedded systems.

## References

1 Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software pipelining. *ACM Computing Surveys*, 27(3):367–432, 1995. doi:10.1145/212094.212131.

2 Mouaiad Alras, Paul Caspi, Alain Girault, and Pascal Raymond. Model-based design of embedded control systems by means of a synchronous intermediate model. In Tianzhou Chen, Dimitrios N. Serpanos, and Walid Taha, editors, *International Conference on Embedded Software and Systems, ICESS'09, Hangzhou, Zhejiang, P. R. China, May 25-27, 2009.*, pages 3–10. IEEE Computer Society, 2009. doi:10.1109/ICESS.2009.36.

3 ARINC 653: Avionics application software standard interface, 2005. URL: http://www.arinc.org.

4 Cédric Auger. *Compilation certifiée de SCADE/LUSTRE*. PhD thesis, Université Paris Sud – Paris XI, 2013. In French. URL: https://tel.archives-ouvertes.fr/tel-00818169.

5 Autosar (automotive open system architecture), release 4, 2009. URL: http://www.autosar.org/specifications/release-40/.

6 Sanjoy K. Baruah. Dynamic- and static-priority scheduling of recurring real-time tasks. *Real-Time Systems*, 24(1):93–128, 2003. doi:10.1023/A:1021711220939.

7 Jacek Blazewicz. Scheduling dependent tasks with different arrival times to meet deadlines. In Heinz Beilner and Erol Gelenbe, editors, *Modelling and Performance Evaluation of Computer Systems, Proceedings of the International Workshop organized by the Commision of the European Communities, Ispra (Varese), Italy, October 4-6, 1976*, pages 57–65. North-Holland, 1976.

8 Vicent Brocal, Miguel Masmano, Ismael Ripoll, Alfons Crespo, Patricia Balbastre, and Jean-Jacques Metge. Xoncrete: a scheduling tool for partitioned real-time systems. In *Embedded and Real*

*Time Software Systems (ERTS$^2$ 2010)*, Toulouse, France, 2010. URL: `http://www.fentiss.com/documents/xoncrete_overview.pdf`.

9 Thomas Carle. *Efficient compilation of embedded control specifications with complex functional and non-functional properties.* Theses, Université Pierre et Marie Curie – Paris VI, October 2014. URL: `https://hal.inria.fr/tel-01088786`.

10 Thomas Carle and Dumitru Potop-Butucaru. Predicate-aware, makespan-preserving software pipelining of scheduling tables. *ACM Transactions on Architecture and Code Optimization*, 11(1):12, 2014. `doi:10.1145/2579676`.

11 Paul Caspi, Adrian Curic, Aude Maignan, Christos Sofronis, Stavros Tripakis, and Peter Niebert. From simulink to scade/lustre to TTA: a layered approach for distributed embedded applications. In Frank Mueller and Ulrich Kremer, editors, *Proceedings of the 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'03). San Diego, California, USA, June 11-13, 2003*, pages 153–162. ACM, 2003. `doi:10.1145/780732.780754`.

12 Houssine Chetto, Maryline Silly, and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems*, 2(3):181–194, 1990. `doi:10.1007/BF00365326`.

13 Yi-Sheng Chiu, Chi-Sheng Shih, and Shih-Hao Hung. Pipeline schedule synthesis for real-time streaming tasks with inter/intra-instance precedence constraints. In *Design, Automation and Test in Europe, DATE 2011, Grenoble, France, March 14-18, 2011*, pages 1321–1326. IEEE, 2011. `doi:10.1109/DATE.2011.5763212`.

14 Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. *N*-synchronous kahn networks: a relaxed model of synchrony for real-time systems. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 180–193. ACM, 2006. `doi:10.1145/1111037.1111054`.

15 Albert Cohen, Valentin Perrelle, Dumitru Potop-Butucaru, Elie Soubiran, and Zhen Zhang. Mixed-criticality in Railway Systems: A Case Study on Signalling Application. In *Workshop on Mixed Criticality for Industrial Systems (WMCIS'2014)*, Paris, France, June 2014. URL: `https://hal.inria.fr/hal-01095111`.

16 Stephen A. Edwards, Sungjun Kim, Edward A. Lee, Isaac Liu, Hiren D. Patel, and Martin Schoeberl. A disruptive computer design idea: Architectures with repeatable timing. In *27th International Conference on Computer Design, ICCD 2009, Lake Tahoe, CA, USA, October 4-7, 2009*, pages 54–59. IEEE Computer Society, 2009. `doi:10.1109/ICCD.2009.5413177`.

17 Stephen A. Edwards and Edward A. Lee. The case for the precision timed (PRET) machine. In *Proceedings of the 44th Design Automation Conference, DAC 2007, San Diego, CA, USA, June 4-8, 2007*, pages 264–265. IEEE, 2007. `doi:10.1145/1278480.1278545`.

18 Peter H. Feiler and David P. Gluch. *Model-Based Engineering with AADL – An Introduction to the SAE Architecture Analysis and Design Language.* SEI series in software engineering. Addison-Wesley, 2012. URL: `http://www.pearsoned.co.uk/bookshop/detail.asp?item=100000000518651`.

19 Sebastian Fischmeister, Oleg Sokolsky, and Insup Lee. Network-code machine: Programmable real-time communication schedules. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006), 4-7 April 2006, San Jose, California, USA*, pages 311–324. IEEE Computer Society, 2006. `doi:10.1109/RTAS.2006.31`.

20 Gerhard Fohler. Changing operational modes in the context of pre run-time scheduling. *IEICE Transactions on Information and Systems*, Special Issue on Responsive Computer Systems:1333–1340, November 1993.

21 Gerhard Fohler and Krithi Ramamritham. Static scheduling of pipelined periodic tasks in distributed real-time systems. In *Proceedings of the Ninth Euromicro Workshop on Real-Time Systems, RTS 1997, 11-13 June, 1997, Toledo, Spain*, pages 128–135. IEEE Computer Society, 1997. `doi:10.1109/EMWRTS.1997.613773`.

22 Alain Girault, Hamoudi Kalla, Mihaela Sighireanu, and Yves Sorel. An algorithm for automatically obtaining distributed and fault-tolerant static schedules. In *2003 International Conference on Dependable Systems and Networks (DSN 2003), 22-25 June 2003, San Francisco, CA, USA, Proceedings*, pages 159–168. IEEE Computer Society, 2003. `doi:10.1109/DSN.2003.1209927`.

23 Raul Gorcitz, Emilien Kofman, Thomas Carle, Dumitru Potop-Butucaru, and Robert de Simone. On the scalability of constraint solving for static/off-line real-time scheduling. In Sriram Sankaranarayanan and Enrico Vicario, editors, *Formal Modeling and Analysis of Timed Systems – 13th International Conference, FORMATS 2015, Madrid, Spain, September 2-4, 2015, Proceedings*, volume 9268 of *Lecture Notes in Computer Science*, pages 108–123. Springer, 2015. `doi:10.1007/978-3-319-22975-1_8`.

24 Paul Le Guernic, Jean-Pierre Talpin, and Jean-Christophe Le Lann. POLYCHRONY for system design. *Journal of Circuits, Systems, and Computers*, 12(3):261–304, 2003. `doi:10.1142/S0218126603000763`.

25 Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003. `doi:10.1109/JPROC.2002.805825`.

26 Thomas A. Henzinger and Christoph M. Kirsch. The embedded machine: Predictable, portable real-time code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(6), 2007. `doi:10.1145/1286821.1286824`.

27 Damir Isovic and Gerhard Fohler. Handling mixed sets of tasks in combined offline and online scheduled real-time systems. *Real-Time Systems*, 43(3):296–325, 2009. `doi:10.1007/s11241-009-9088-3`.

28 Hermann Kopetz. Event-triggered versus time-triggered real-time systems. In Arthur I. Karsh-

mer and Jürgen Nehmer, editors, *Operating Systems of the 90s and Beyond, International Workshop, Dagstuhl Castle, Germany, July 8-12, 1991, Proceedings*, volume 563 of *Lecture Notes in Computer Science*, pages 87–101. Springer, 1991. `doi:10.1007/BFb0024530`.

**29** Hermann Kopetz and Günther Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

**30** Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceeding of the IEEE*, 75(9):1235–1245, September 1987. `doi:10.1109/PROC.1987.13876`.

**31** Joseph Y.-T. Leung and M. L. Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Information Processing Letters*, 11(3):115–118, 1980. `doi:10.1016/0020-0190(80)90123-4`.

**32** Stéphane Louise, Matthieu Lemerre, Christophe Aussaguès, and Vincent David. The OASIS kernel: A framework for high dependability real-time systems. In Taghi M. Khoshgoftaar, editor, *13th IEEE International Symposium on High-Assurance Systems Engineering, HASE 2011, Boca Raton, FL, USA, November 10-12, 2011*, pages 95–103. IEEE Computer Society, 2011. `doi:10.1109/HASE.2011.38`.

**33** Mohamed Marouf, Laurent George, and Yves Sorel. Schedulability analysis for a combination of non-preemptive strict periodic tasks and preemptive sporadic tasks. In *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation, ETFA 2012, Krakow, Poland, September 17-21, 2012*, pages 1–8. IEEE, 2012. `doi:10.1109/ETFA.2012.6489569`.

**34** Uml profile for marte: Modeling and analysis of real-time embedded systems, version 1.1, 2011. OMG document number: formal/2011-06-02. URL: `http://www.omg.org/spec/MARTE/1.1/`.

**35** James F. Mason, Kenn R. Luecke, and Jahn A. Luke. Device drivers in time and space partitioned operating systems. In *25th Digital Avionics Systems Conference, IEEE/AIAA*, pages 1–9, Portland, OR, USA, Oct. 2006. `doi:10.1109/DASC.2006.313742`.

**36** Alix Munier. The basic cyclic scheduling problem with linear precedence constraints. *Discrete Applied Mathematics*, 64(3):219–238, 1996. `doi:10.1016/0166-218X(94)00126-X`.

**37** Claire Pagetti, Julien Forget, Frédéric Boniol, Mikel Cordovilla, and David Lesens. Multi-task implementation of multi-periodic synchronous programs. *Discrete Event Dynamic Systems*, 21(3):307–338, 2011. `doi:10.1007/s10626-011-0107-x`.

**38** Paul Pop, Petru Eles, and Zebo Peng. Scheduling with optimized communication for time-triggered embedded systems. In Ahmed Amine Jerraya, Luciano Lavagno, and Frank Vahid, editors, *Proceedings of the Seventh International Workshop on Hardware/Software Codesign, CODES 1999, Rome, Italy, 1999*, pages 178–182. ACM, 1999. `doi:10.1145/301177.303812`.

**39** Dumitru Potop-Butucaru, Akramul Azim, and Sebastian Fischmeister. Semantics-preserving implementation of synchronous specifications over dynamic TDMA distributed architectures. In Luca P. Carloni and Stavros Tripakis, editors, *Proceedings of the 10th International conference on Embedded software, EMSOFT 2010, Scottsdale, Arizona, USA, October 24-29, 2010*, pages 199–208. ACM, 2010. `doi:10.1145/1879021.1879048`.

**40** Dumitru Potop-Butucaru, Robert de Simone, Yves Sorel, and Jean-Pierre Talpin. Clock-driven distributed real-time implementation of endochronous synchronous programs. In Samarjit Chakraborty and Nicolas Halbwachs, editors, *Proceedings of the 9th ACM & IEEE International conference on Embedded software, EMSOFT 2009, Grenoble, France, October 12-16, 2009*, pages 147–156. ACM, 2009. `doi:10.1145/1629335.1629356`.

**41** Wolfgang Puffitsch, Eric Noulard, and Claire Pagetti. Mapping a multi-rate synchronous language to a many-core processor. In *19th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2013, Philadelphia, PA, USA, April 9-11, 2013*, pages 293–302. IEEE Computer Society, 2013. `doi:10.1109/RTAS.2013.6531101`.

**42** Krithi Ramamritham, Gerhard Fohler, and Juan Manuel Adan. Issues in the static allocation and scheduling of complex periodic tasks. In *10th IEEE Workshop on Real-Time Operating Systems and Software, RTOSS 1993*, pages 11–16, 1993. URL: `http://dl.acm.org/citation.cfm?id=192806.185839`.

**43** Pascal Richard, Francis Cottet, and Claude Kaiser. Précédences généralisées et ordonnançabilité des tâches de suivi temps réel d'un laminoir. *Journal européen des systèmes automatisés*, 35:1055–1071, 2001. URL: `http://www.lias-lab.fr/publications/5810/2001-jesa-richard.pdf`.

**44** John M. Rushby. Bus architectures for safety-critical embedded systems. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *Embedded Software, First International Workshop, EMSOFT 2001, Tahoe City, CA, USA, October, 8-10, 2001, Proceedings*, volume 2211 of *Lecture Notes in Computer Science*, pages 306–323. Springer, 2001. `doi:10.1007/3-540-45449-7_22`.

**45** Ahmad Al Sheikh, Olivier Brun, Pierre-Emmanuel Hladik, and Balakrishna J. Prabhu. Strictly periodic scheduling in ima-based architectures. *Real-Time Systems*, 48(4):359–386, 2012. `doi:10.1007/s11241-012-9148-y`.

**46** Jia Xu. Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations. *IEEE Transactions on Software Engineering (TSE)*, 19(2):139–154, 1993. `doi:10.1109/32.214831`.

**47** Wei Zheng, Jike Chong, Claudio Pinello, Sri Kanajan, and Alberto L. Sangiovanni-Vincentelli. Extensible and scalable time triggered scheduling. In *Fifth International Conference on Application of Concurrency to System Design (ACSD 2005), 6-9 June 2005, St. Malo, France*, pages 132–141. IEEE Computer Society, 2005. `doi:10.1109/ACSD.2005.13`.