Developing critical embedded systems on multicore architectures: the Prelude-SchedMCore toolset

Mikel Cordovilla[1] Frédéric Boniol[1] Julien Forget[2] Eric Noulard[1]
Claire Pagetti[1]
[1] ONERA - Toulouse, [2] LIFL/INRIA - Lille

Abstract

In this paper we present an end-to-end framework for the design and the implementation of embedded systems on a symmetric multicore. The developer first specifies the system using the Prelude *language, a formal real-time architecture description language. The compiler then translates the program into a set of communicating periodic tasks that preserves the semantics of the original program. The schedulability analysis is performed by the* SchedMCore *analyzer. If the program is schedulable, it can finally be executed on the target multicore architecture using the* SchedMCore *execution environment.*

1 Introduction

The development of critical embedded systems undergoes strict development processes. In this context, the purpose of this paper is to propose an integrated development framework for critical embedded systems, that goes from the programming of high level specifications, using the Prelude language, to the execution of a semantically equivalent multithreaded code (generated automatically) on a symmetric multicore architecture, using the SchedMCore framework.

1.1 Targeted systems

As an example, let us consider an adapted version of the Flight Application Software (FAS) of the Automated Transfer Vehicle (ATV) designed by Astrium for resupplying the International Space Station (ISS). Figure 1 provides a simplified informal description of its software architecture. The FAS acquires several data treated by dedicated sub-functions: orientation and speed (`Gyro Acq`), position (`GPS Acq` and `Str Acq`) and telecommands from the ground station (`TM/TC`). The *Guidance Navigation and Control* function (divided into `GNC_US` and `GNC_DS`) computes the commands to apply while the *Failure Detection Isolation and Recovery* function (`FDIR`) verifies the state of the FAS and checks for possible failures. Commands are sent to the control devices: thruster orders (`PDE`), power distribution orders (`PWS`), solar panel positioning orders (`SGS`) and telemetry towards the ground station (`TM/TC`).

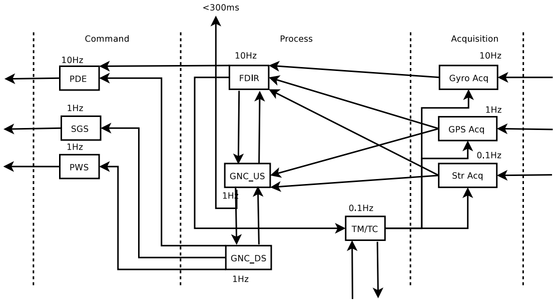


Figure 1. FAS architecture

The specification of such a system must precisely define three things: (1) the functional behaviour of each function: each function can be programmed separately with existing languages, such as Simulink, Scade or directly in low level code (C for instance); (2) the real-time characteristics of the system: periods and deadlines which depend on the physical characteristics of the system; (3) multi-rate communication patterns: the way data is exchanged between functions is complex since functions of different periods communicate.

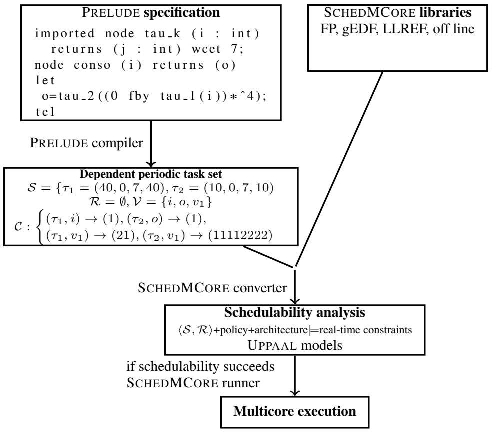
A correct implementation must respect all the real-time constraints of the system and must also be functionally deterministic, meaning that the outputs of the system must always be the same for a given sequence of inputs. This requires task communications to be fully deterministic, which means that for any execution of the system the same job (or instance) of the producer must communicate with the same job of the consumer of the communication.

1.2 Contribution

Figure 2 shows the development process within the Prelude-SchedMCore framework. In this paper, we focus more on presenting a complete operational toolset rather than on new theoretical results (we published most results separately previously). We show how the different steps of the development come together inside our framework and we detail the tools developed to this intent. The framework is made up of three components:

- The Prelude compiler which has been extended to generate code executable by the SCHEDMCORE environment (Section 3);

- The SCHEDMCORE multiprocessor schedulability analyzer (Section 4);
- The SCHEDMCORE environment which is an extensible, easy-to-use and portable real-time scheduler framework for multicore (Section 5).



```
imported node tau_k (i : int)
  returns (j : int) wcet 7;
node conso (i) returns (o)
let
 o=tau_2((0 fby tau_1(i))*^4);
tel
```

PRELUDE specification
SCHEDMCORE libraries
FP, gEDF, LLREF, off line
PRELUDE compiler
Dependent periodic task set
$\mathcal{S} = \{\tau_1 = (40, 0, 7, 40), \tau_2 = (10, 0, 7, 10)$
$\mathcal{R} = \emptyset, \mathcal{V} = \{i, o, v_1\}$
$\mathcal{C} : \begin{cases} (\tau_1, i) \rightarrow (1), (\tau_2, o) \rightarrow (1), \\ (\tau_1, v_1) \rightarrow (21), (\tau_2, v_1) \rightarrow (11112222) \end{cases}$
SCHEDMCORE converter
Schedulability analysis
$\langle \mathcal{S}, \mathcal{R} \rangle$+policy+architecture$\models$real-time constraints
UPPAAL models
if schedulability succeeds
SCHEDMCORE runner
Multicore execution

Figure 2. Development process

1.2.1 PRELUDE: a high level specification language

PRELUDE [20, 18][1] is a formal language designed for the specification of the software architecture of a critical embedded control system. It belongs to the synchronous data-flow languages [4] and focuses on dealing with the functional and real-time aspects of multi-periodic systems conjointly. From a PRELUDE program the compiler generates a program consisting of a periodic dependent task set. For each pair of producer job and consumer job of the task set the following properties must be ensured: (1) the producer completes before the consumer starts. This is modeled by adding a precedence constraint from the producer to the consumer. The respect of the precedence constraints must be ensured by the scheduler. (2) Produced data remains available until the completion of the consumer. This is fulfilled using a specific communication protocol (directly generated by the compiler) derived from [32] and detailed in [18].

The initial release of the compiler (described in [18]) was targeted for a uniprocessor platform. The task set was executed with MARTE OS [31], using a scheduling policy derived from EDF [28] and from the work of [7]. In this paper we present how it was extended to enable the execution of PRELUDE programs on a multicore architecture.

1.2.2 Multiprocessor schedulability analyzer

The task set generated by PRELUDE must be scheduled in a way that respects the real-time attributes of each task and the *extended precedence constraints* between the tasks (the constraints can relate tasks with different periods).

Scheduling policies on multiprocessor platform have been summarized in surveys such as [15]. We consider preemptive policies accepting *(full) migration*, meaning that a suspended instance may be resumed on a different processor. The SCHEDMCORE[2] framework implements the policies FP (fixed priority), gEDF (global Earliest Deadline First), gLLF (global Least Laxity First) and LLREF (Largest Local Remaining Execution First) [8].

Before executing the program, we must check that the execution will respect the real-time constraints of the system. This *schedulability analysis* determines whether a task set is correctly scheduled by a given policy on an architecture. We have developed an exact approach based on an efficient enumerative technique [10] for dependent task sets. In summary, the possible execution sequences of a system are first encoded as a finite automaton. Then, using either the UPPAAL model checker [3] or a custom C program, we explore this automaton to find which executions violate the constraints of the system. Though this brute force method may seem expansive, we obtain pretty good results even with complex task sets (huge number of tasks, utilization factor close to the number of processors or huge least common multiple of periods). The automaton model can also be used to generate a schedule computed off-line.

1.2.3 SCHEDMCORE execution environment

The objective of SCHEDMCORE execution environment (or runtime) is to provide a validation and execution tool, somewhere between a pure simulator and a true hard real-time execution environment. It allows to run a set of tasks written in C using various real-time multicore scheduling policies on a standard multicore system. Most simulators are dedicated either to functional simulation or temporal simulation, while SCHEDMCORE enables to do both simultaneously. Even if SCHEDMCORE runtime cannot be considered as a hard real-time runtime (yet), it is far more realistic than a simulator: it can schedule real C functions on a real-time timescale. This provides a "simulation" pretty faithfull to the actual system, while remaining portable, easy to use and to extend. SCHEDMCORE provides libraries for most common multicore scheduling policies but was designed so that a user can easily add his own new policy.

2 Related works

Development frameworks for multi-periodic systems SIMULINK [33] is widely used in many industrial appli-

[1] The PRELUDE compiler is available for download at http://www.lifl.fr/~forget/prelude.html

[2] The SCHEDMCORE environment is available for download at http://sites.onera.fr/schedmcore/

cation domains and allows the description of communicating multi-periodic systems. Current code generators, such as Real-time workshop-embedded coder from the MathWorks or TargetLink from dSpace, provide a multi-threaded code generation on uniprocessor. In [6] the authors translate a multi-periodic SIMULINK specification into a semantical equivalent synchronous program that is executed on a multiprocessor time triggered architecture (TTA) where no preemption is allowed.

Synchronous data-flow languages, such as LUSTRE [24] or SCADE [16], have been extended with operators that enable the specification of real-time constraints in order to program multi-threaded system more easily [13]. Thread synchronizations rely on a specific communication protocol initially defined in [32] for uniprocessor and later extended specifically for Loosely Time Triggered Architectures (LTTA) in [34].

Finally, automated distribution of synchronous program has been studied in [21, 1]. However, these studies are not dedicated to multi-periodic systems and thus scheduling policies are not considered.

Schedulability analysis Lots of theoretical results are already available for independent task sets (see surveys [2, 15]). However there are not so many for dependent task sets and not so many tools (even for independent task sets) are available yet. STORM [35] is a multiprocessor scheduling simulator. In [14] a framework allowing the analysis of tasks configuration for multiprocessor machines with UPPAAL models is proposed. This framework supports rich task models including timing uncertainties in arrival and execution times, and dependencies between tasks (such as precedences and resource occupation). However the task set should not exceed 30 tasks due to performance concerns.

Execution environments Lots of real-time execution environments or operating systems are available: industrial ones like VxWorks, LynxOS or PikeOS; modified Linux like Xenomai, LitmusRT or SCHED_EDF for Linux (see [17] and references therein); academics OS like MarteOS [31] and even user space runtime like Meta-scheduler [27]. We need an open environment that enables the implementation of user-specific scheduling policies, which makes us rule out industrial solutions. Modified Linux like LitmusRT or sub-kernel approaches like Xenomai are too closely tied to the kernel for our purpose as this requires to patch and recompile the kernel when new releases of Linux occur. SCHEDMCORE can run on top of these environments but does not rely on them. MARTE OS [31] and the Meta-scheduler [27] answer our needs partially but, in both cases, those environments only support uniprocessor platform. In the end, we reused the conceptual idea of a pluggable scheduler framework (of MARTE OS and the Meta-scheduler) but started with a fresh new source code.

3 The PRELUDE language

For the first step of the development, that is the specification of the system, we rely on the PRELUDE language previously introduced in [20, 18]. PRELUDE is a formal Real-Time Architecture Description Language with synchronous semantics dedicated to the specification of critical embedded control systems.

3.1 Language definition

Flows and clocks PRELUDE is a synchronous data-flow language and thus shares similarities with LUSTRE [24], SIGNAL [5] or LUCID SYNCHRONE [30]. Variables and expressions are *flows*. A flow is a sequence of pairs $(v_i, t_i)_{i\in\mathbb{N}}$, where v_i is a value in some domain $\mathcal{V}$ and t_i is a date in $\mathbb{Q}$ $(\forall i, t_i < t_{i+1})$. The *clock* of a flow, is its projection on $\mathbb{Q}$. It defines the set of instants during which the values of the flow are computed: value v_i must be computed during the interval (or instant) $[t_i, t_{i+1}[$. According to this relaxed synchronous hypothesis, different clocks have different durations for their instants. Two flows are *synchronous* if they have the same clock. We distinguish a specific class of clocks corresponding to periodic task activations called *strictly periodic clocks*: the clock $h = (t_i)_{i\in\mathbb{N}}$ is strictly periodic if there exists some n such that $t_{i+1} - t_i = n$ for all i. n is the *period* of h and t_0 is its *phase*.

Nodes hierarchy A PRELUDE program consists of a set of *nodes*. A node defines its outputs flows from its input flows. It can either be hierarchical, in which case the relationship between its inputs and outputs is defined by a set of equations, or it can be imported, in which case the node is implemented outside the program using another existing language (C or LUSTRE for instance). A simple program is given below:

```
imported node plus1(i: int) returns (o:int) wcet 5;

node N(i:int rate (10,0)) returns (o:int rate (10,0))
let o=plus1(i); tel
```

Following the data-flow programing style, the node `N` is activated each time it receives an input `i`, i.e. each 10 time units (the term `(10,0)` denotes the clock of period 10 and of phase 0). At each activation, it computes the output `o` as the result of the application of the imported node `plus1` to `i`.

Rate transition operators PRELUDE defines a set of *rate transition operators* that enable the definition of user-specified communication patterns between nodes of different rates. Let e be a flow expression, $k \in \mathbb{N}$ and $q \in \mathbb{Q}$:

- cst `fby` e is the delay operator borrowed from LUCID SYNCHRONE: it first produces cst and then produces each value of e delayed by one instant. The clock of this flow is the same as that of e;

- $e *\hat{} k$ produces a flow whose period is k times shorter than that of e. Each value of e is duplicated k times in the result;

- $e / \hat{} k$ produces a flow k times slower than e. The result only keeps the first value of each k successive values of e;

- $e \sim> q$ produces a flow where each value of e is delayed by $q * n$, where n is the period of e.

These operators are illustrated on the example below (nodes τ_1, τ_2 and τ_3 are imported, their declaration is omitted here):

```
node sampling (i : rate (10,0)) returns (o1, o2)
var vf, vs;
let
  (o1,vf)=tau_1(i, (0 fby vs)*^3);
  vs=tau_2(vf/^3);
  o2 = tau_3((vf~>1/10)/^6,(vs~>1/30)/^2);
tel
```

The behaviour of this program is the following (we detail the values produced and their dates of production):

date	0	1	10	20	30	40	50	60	61	...
`vf`	v_0		v_1	v_2	v_3	v_4	v_5	v_6		...
`vf/^3`	v_0				v_3			v_6		...
`vs`	v'_0				v'_1			v'_2		...
`0 fby vs`	0				v'_0			v'_1		...
`(0 fby vs)*^3`	0		0	0	v'_0	v'_0	v'_0	v'_1		...
`vf~>1/10/^6`		v_0							v_6	...
`vs~>1/30/^2`		v'_0							v'_2	...

3.2 Compilation

The compiler translates a PRELUDE program into a set of dependent periodic tasks. The preservation of the semantics of the program through the different steps of the compilation was formally proved in [18].

3.2.1 Static analyses

The compiler first performs a series of static analysis, to ensure that the program is correct, i.e. that its semantics is well-defined. First a standard *type-checking* is performed [29]. The language is strongly typed, in the sense that the execution of a program cannot produce a run-time type error. Then, the *causality check* (derived from [25]) verifies that the program is causal, which means that no variable instantaneously depends on itself (not without a **fby** in the dependencies). Finally, the *clock calculus* [20] verifies that the program only combines flows that have the same clock. This ensures that the program only accesses values at dates at which they are indeed available.

3.2.2 Translation into a dependent task set

The *system model* produced by PRELUDE consists of a set of concurrent periodic tasks communicating via shared variables stored in buffers. This model is encoded in a C file generated by the compiler. More formally, a system is defined as a tuple $\langle \mathcal{S}, \mathcal{V}, \mathcal{R}, \mathcal{C} \rangle$ where:

- $\mathcal{S} = \{\tau_i\}_{i=1,\dots,n}$ is a finite periodic task set;

- $\mathcal{V}$ is the set of variables produced and consumed by the tasks. $\tau_i.in \subseteq \mathcal{V}$ (resp. $\tau_i.out \subseteq \mathcal{V}$) stands for the set of variables that are consumed (resp. produced) by τ_i;

- $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{P} \times \mathcal{S}$ is the precedence relation, where $\mathcal{P}$ is the set of *periodic extended precedence constraints*;

- $\mathcal{C} : \mathcal{S} \times \mathcal{S} \times \mathcal{V} \to \mathcal{W}$ is the (partial) communication function, where $\mathcal{W}$ is the set of *dependence words*.

Periodic tasks A periodic task set $\mathcal{S} = \{\tau_i\}_{i=1,\dots,n}$ is a finite set of tasks, where each task τ_i has four real-time attributes (T_i, C_i, O_i, D_i). T_i is the period of repetition, C_i is the worst case execution time estimation (wcet), O_i is the first arrival time and D_i is the relative deadline, with $D_i \leq Ti$. We denote $\tau_{i.k}$ the k^{th} instance of τ_i (starting with instance 0), which we will call a *job* (or task job). The job $\tau_{i.k}$ is released at date $o_{i.k} = O_i + kT_i$. The *hyper-period* is the least common multiple of all the task periods, i.e. $H = lcm_{1 \leq i \leq n}(T_i)$. This task model is a standard adaptation from [28].

Extended precedence constraints Tasks are dependent because they are related by precedence constraints. A *simple precedence constraint* $\tau_i \to \tau_j$ relates two tasks with the same period and imposes that each job of the producer executes before one job of the consumer. An *extended precedence constraint* relates a subset of the jobs of the communicating tasks. Let $\tau_{i.n} \to \tau_{j.n'}$ denote a precedence constraint from the instance n of τ_i to the instance n' of τ_j. For any $n \in \mathbb{N}$, let $\mathcal{I}_n$ denote the set of integers of the interval $[0, n[$.

Definition 1 (Periodic Extended Precedence Constraint) *Let τ_i and τ_j be two tasks, $p = lcm(T_i, T_j)$ and $M_{i,j} \subseteq \mathcal{I}_{p/T_i} \times \mathcal{I}_{p/T_j}$ (notice that $M_{i,j}$ is a finite set). The periodic extended precedence constraint $\tau_i \xrightarrow{M_{i,j}} \tau_j$ is defined as the following set of task instance precedence constraints:*

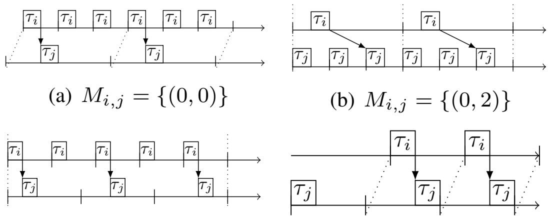
$$\forall (n, n') \in M'_{i,j}, \tau_{i.n} \to \tau_{j.n'}$$

$$M'_{i,j} = \{(n, n') | \begin{array}{l} \exists k \in \mathbb{N}, (m, m') \in M_{i,j}, \\ (n, n') = (m, m') + (k\frac{p}{T_i}, k\frac{p}{T_j}) \end{array} \}$$

A simple precedence constraint $\tau_i \to \tau_j$ is actually a particular case of periodic extended precedence constraints where $M_{i,j} = \{(0, 0)\}$. Other examples are drawn below.

Communication protocol Tasks are synchronous meaning that all their inputs $\tau_i.in$ must be available at the beginning of the execution of each job and all their outputs $\tau_i.out$ are produced simultaneously at the end of the execution of each job.

There is a data-dependency between two tasks τ_i and τ_j when there exists a variable v that is produced by τ_i and



(a) $M_{i,j} = \{(0,0)\}$

(b) $M_{i,j} = \{(0,2)\}$

(c) $M_{i,j} = \{(0,0),(2,1),(4,3)\}$

(d) Not representable

consumed by τ_j. The compiler allocates a finite buffer $b_{i,j,v}$ where τ_i writes values and τ_j reads values. The protocol specifying how a task accesses the buffer is defined by a *dependence word*. Let the *ultimately periodic word* $w = v(u)^\omega$ denote the infinite sequence of integers consisting of the finite prefix v followed by the infinite repetition of the finite sequence u. Let $w[k]$ denote the k^{th} value of w.

Definition 2 (Dependence word) *Let τ_i, τ_j be two tasks and v be a variable such that $v \in \tau_i.out$ and $v \in \tau_j.in$. The dependence words $w = \mathcal{C}(\tau_i, v, \tau_j)$ and $w' = \mathcal{C}(\tau_j, v, \tau_i)$ are ultimately periodic words such that $w[k]$ denotes the index of the cell of $b_{i,j,v}$ where $\tau_{i.k}$ writes and $w'[k]$ denotes the cell where $\tau_{j.k}$ reads.*

For instance, if $\mathcal{C}(\tau_i, v, \tau_j) = 0(102)^\omega$, then values produced by τ_i are stored in the buffer as follows: $\tau_{i.0}$ is not stored, $\tau_{i.1}$ is stored in cell 1, $\tau_{i.2}$ is not stored, $\tau_{i.3}$ is stored in cell 2, then $\tau_{i.4}$ is stored in cell 1, $\tau_{i.5}$ is not stored and so on (the pattern 102 repeats indefinitely).

Example 1 *Let us consider the node sampling. The system generated by* PRELUDE *is $\langle \mathcal{S}, \mathcal{V}, \mathcal{R}, \mathcal{C} \rangle$ with:*

1. $\mathcal{S} = \{\tau_1 = (10,2,0,10), \tau_2 = (30,5,0,30), \tau_3 = (60,30,1,60)\}$
2. $\mathcal{V} = \{i, o_1, o_2, v_f, v_s\}$ *such that* $\tau_1.in = \{i, v_s\}$, $\tau_1.out = \{o_1, v_f\}$, $\tau_2.in = \{v_f\}$, $\tau_2.out = \{v_s\}$, $\tau_3.in = \{v_s, v_f\}$ *and* $\tau_3.out = \{o_2\}$,
3. $\mathcal{R} = \{(\tau_1, \{(0,0)\}, \tau_2), (\tau_1, \{(0,0)\}, \tau_3), (\tau_2, \{(0,0)\}, \tau_3)\}$
4. $\mathcal{C}$ *is defined as follows:*

	i	o_1	o_2	v_f	v_s
τ_1	(1)	(1)		$b_{1,2}$: (100) $b_{1,3}$: (100000)	$b_{2,1}$: (111222)
τ_2				$b_{1,2}$: (1)	$b_{2,1}$: (21) $b_{2,3}$: (10)

Let us for instance consider the words for v_s. v_s is stored in two different buffers $b_{2,1}$ for $\tau_2 \to \tau_1$ and $b_{2,3}$ for $\tau_2 \to \tau_3$. v_s is produced once by τ_2 while τ_1 executes three times. v_s is consumed with an `fby` *, therefore there are two cells in the buffer and the initial value is in the first cell. During the period of τ_2, $\tau_{2.0}$ stores the value in the second cell so that it does not interfere with τ_1. Then, $\tau_{2.1}$ writes in cell 1 while τ_1 reads three times the previous value in cell 2. This behaviour repeats indefinitely.*

4 SCHEDMCORE schedulability analyzer

The second step of the development consists in choosing a multiprocessor scheduling policy and verifying off-line the correctness of the schedule. Schedulability analysis with SCHEDMCORE relies on the `lsmc_converter` tool, which transforms the task set $\langle \mathcal{S}, \mathcal{R} \rangle$ produced by PRELUDE[3] into a finite automaton describing all the possible executions. The automaton is then verified either with the UPPAAL model checker [3] or with an ad hoc exhaustive search we developped in C. The `lsmc_converter` can also compute off-line optimal scheduling strategies (either static or dynamic priority based), which are correct by construction.

4.1 Handling precedence constraints

Let us first define formally the conditions that need to be verified. Let $\mathcal{J}$ denote the infinite set of jobs $\mathcal{J} = \{\tau_{i.k}, 1 \leq i \leq n, k \in \mathbb{N}\}$. Given a schedule, we define two functions $s, e : \mathcal{J} \to \mathbb{N}$ where $s(\tau_{i.k})$ is the start time and $e(\tau_{i.k})$ is the completion time of $\tau_{i.k}$ in the considered schedule. We say that a dependent task set is *schedulable* under a given scheduling policy if the schedule produced by this policy respects all the constraints of the task set and all the job precedence constraints:

Definition 3 *Let $\langle \mathcal{S}, \mathcal{R} \rangle$ be a dependent task set. It is* schedulable *under a given scheduling policy if and only if:*

$$\begin{cases} \forall \tau_{i.k},\ e(\tau_{i.k}) \leq d_{i.k} \wedge s(\tau_{i.k}) \geq o_{i.k} \\ \forall \tau_{i.k} \to \tau_{j.k'},\ e(\tau_{i.k}) \leq s(\tau_{j.k'}) \end{cases}$$

There are mainly two different approaches available for handling precedence constraints. The first approach relies on the use of (binary) semaphore synchronizations: a semaphore is allocated for each precedence and the destination task of the precedence constraint must wait for the source task of the precedence constraint to release the semaphore before it can start its execution. In the second approach, as shown in [7, 19], if there is a precedence constraint from τ_i to τ_j, it is sufficient to ensure that τ_j always starts after τ_i and that τ_i has a higher priority than τ_j to respect the constraint. Unfortunately, this is difficult to apply to the multiprocessor case because tasks can execute simultaneously on different processors, regardless of their relative priorities. We would have to force τ_j to be released after the worst response time of τ_i, which is clearly sub optimal.

We therefore choose to ensure the precedence constraints by (1) using explicit synchronization mechanisms, there is simply no transition in our automaton model that enables a task to start executing before its predecessors (precedence constraints are taken as is) (2) forcing the execution of a task to take the wcet. Forcing the wcet prevents scheduling anomaly issues where execution times

[3] `lsmc_converter` accepts its own task file format description as well, one does not need PRELUDE in order to do schedulability analysis.

lower than the wcet lead to a system becoming unschedulable on-line while it has been proved schedulable off-line [22] (the off-line analysis is based on the wcet).

4.2 Modeling system execution

The scheduling of the system can be represented as a sequence of configurations. At time t, the state of the system is represented by $\mathit{conf}(t) = \langle \mathit{conf}(\tau_1, t), \ldots, \mathit{conf}(\tau_n, t)\rangle$ where for $i \in [1, n]$, $\mathit{conf}(\tau_i, t) = (T_i^c(t), C_i^c(t), O_i^c(t), D_i^c(t))$. $O_i^c(t) = max(O_i - t, 0)$ is the time remaining until the first release of τ_i. If $O_i^c(t) > 0$, all the other parameters are null. $O_i^c(t) = 0$ means that at time t the task has been released (at least once), in which case the other parameters have a value. $T_i^c(t) = T_i + ((t - O_i) \ \mathit{mod}\, T_i)$ is the time remaining until the next release of τ_i, $D_i^c(t) = max(0, T_i^c(t) - (T_i - D_i))$ is the time remaining until the next deadline and $C_i^c(t)$ is the remaining execution time for the current job of τ_i. Since all the decisions are taken at integer dates, it is sufficient to describe a discrete model as shown in [23].

Example 2 *The real-time execution of the task set below on the left ($O_i = 0$ and $T_i = D_i$), with a fixed priority policy on two cores, can be represented as the Gantt diagram below on the right:*

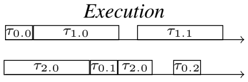
Task set

	T	C	prio
τ_0	5	1	1
τ_1	7	5	2
τ_2	10	7	3

$$\tau_0 \xrightarrow{\{(0,0),(3,2)\}} \tau_1$$

$$\tau_2 \xrightarrow{\{(0,1)\}} \tau_1$$

Execution



The equivalent sequence of configurations is:

time	*0*	*1*	*2*	…
τ_0	$(5, 1)$	$(4, 0)$	$(3, 0)$	
τ_1	$(7, 5)$	$(6, 5)$	$(5, 4)$	
m_1	$\begin{cases}(0, (0, 0), (3, 2)) \\ (2, (0, 1))\end{cases}$	$\begin{cases}(0, (3, 2)) \\ (2, (0, 1))\end{cases}$	$\begin{cases}(0, (3, 2)) \\ (2, (0, 1))\end{cases}$	
τ_2	$(10, 7)$	$(9, 6)$	$(8, 5)$	

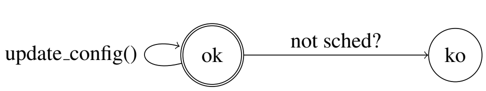
m_1 represents the periodic extended precedence constraints for τ_1. For instance, $\tau_{1.0}$ can only start executing at date 1, which is the date of completion of its predecessor $\tau_{0.0}$. Note that at time 5, there are two overlapping configurations: one for the completion of $\tau_{0.0}$ ($(0, 0)$) and one for the release of $\tau_{0.1}$ ($(5, 1)$). This holds at any release.

4.3 Analysis for existing sub-optimal online policies

There are mainly two approaches for scheduling a task set on a multiprocessor platform: a *global strategy*, where there is a single queue from which the tasks with the highest priority are selected to execute on any of the available processors, or a *(semi) partitioning strategy* where an algorithm allocates a distinct subset of tasks (or jobs) to a (subset of) processor(s) (leading to different queues for different processors and local scheduling).

The current implementation of SCHEDMCORE schedulability analyzer is targeted for global strategies but could easily be extended to (semi) partitioned policies. Currently, four policies are supported: (1) FP (fixed priority, tasks are assigned a specified priority off-line), (2) gEDF (multiprocessor extension of Earliest Deadline First, priority is given according to the next absolute deadline of each job), (3) gLLF (multiprocessor extension of Least Laxity First, priority is assigned regularly according to the remaining execution time), and (4) LLREF (Largest Local Remaining Execution First, mixing fair execution and local laxity).

The `lsmc_converter` takes a task set (in our case the model generated by PRELUDE), verifies several necessary schedulability conditions, such as checking that the load does not exceed the maximum load and models its execution with a given scheduling policy as a finite automaton [10] depicted below:



The automaton works as follows:

- Variable *list_prio(t)* contains the list of tasks ordered by decreasing priority at time t. Priorities are assigned according to the chosen policy, this is the only part that depends on the scheduling policy. The task τ_i executes on a processor at time t if it is active, is not blocked by a precedence constraint and its position in *list_prio(t)* is less than the number of processors;
- The precedence constraints are stored in a bidimensional array *tabCurrent*;
- The loop transition labelled *update_tab* updates the configurations $(T_\tau^c(t), C_\tau^c(t), O_\tau^c(t), D_\tau^c(t))$, the array *tabCurrent* and the list *list_prio(t)* after one unit of time elapses;
- The transition labelled *not sched?* can be fired only if the variable *sched* is equal to false, which only occurs when the task set is not schedulable (i.e. when some of the constraints of Definition 3 are violated).

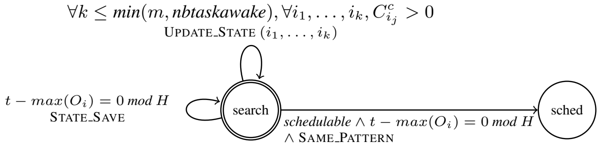
The schedulability analysis then simply consists in proving that the transition *not sched?* is never fired. `lsmc_converter` generates a model to be verified either by the UPPAAL model checker or by an ad hoc exploration algorithm programmed in C. Since the execution is deterministic (when two tasks have the same priority, the scheduler always choose one task in a deterministic way to have the highest priority), it is sufficient to explore the execution until a sufficient point or a violation of the temporal constraints. The window is known to be $[0, H]$ for synchronous constrained deadline task sets and for FP [12]. Otherwise, since the pattern of repetition of a schedule is a multiple of H, it is sufficient to explore until finding a repetitive configuration. The performance of this approach is relatively good compared to others, see §3.3 of [10] for detailed figures.

4.4 Optimal off-line schedule

A task set $\langle \mathcal{S}, \mathcal{R} \rangle$ is *feasible* for a given architecture if there exists a schedule that respects the temporal and the precedence constraints of every task (regardless of any specific policy). A scheduling algorithm is *optimal* [26], with respect to an architecture and a class of policies (e.g. preemptive/non-preemptive, static/dynamic priority, etc.), if it can schedule all the feasible task sets. Being optimal on multiprocessor requires to be *clairvoyant* [26] meaning that the scheduler must know a priori the future events, in particular, release times. Thus, on-line policies that only know of currently active tasks (which is the case of most of the on-line policies) cannot be optimal.

A widespread approach in critical embedded scheduling is to compute a feasible schedule off-line (also called sequencing). This provides several interesting properties: (1) all the costs are assessable before run-time, for instance the number of preemptions, migrations and all context switches (2) no semaphores or synchronisation mechanisms are required since the dispatcher knows the predefined instants where tasks are resumed or suspended (3) off-line schedules are tractable and provide a very simple and efficient scheduler implementation (4) off-line scheduling is optimal.

The `lsmc_converter` is able to construct off-line schedules that can be run by the SCHEDMCORE runner. The construction also relies on the use of an automaton. This time, the objective is to find a trace reaching the state *sched*.


$\forall k \leq min(m, nbtaskawake), \forall i_1, \ldots, i_k, C^c_{i_j} > 0$
UPDATE_STATE $(i_1, \ldots, i_k)$
$t - max(O_i) = 0 \; mod \; H$
STATE_SAVE
search
sched
$schedulable \wedge t - max(O_i) = 0 \; mod \; H$
$\wedge$ SAME_PATTERN

The automaton works as follows:

- The transition UPDATE_STATE updates the values of the tuples $(T^c_\tau(t), C^c_\tau(t), O^c_\tau(t), D^c_\tau(t))$ after one unit of time. It is a non deterministic transition since we consider any combination of k active tasks, where k is the minimum between the number of processors m and the number of active tasks. These k tasks are assumed to access the processors at time t;

- The loop transition STATE_SAVE may be fired non-deterministically at time $t = max_{i \leq n}(O_i) \; mod \; (H)$ (remember that H is the hyperperiod). If it is fired, the current configuration is saved;

- The condition on the transition SAME_PATTERN states that the task set must be schedulable and the backup is exactly equal to the current configuration. This means that we have found a repetitive schedule.

Optimal fixed priority assignment In [11], Cucu and Goossens note that it is possible to determine an optimal fixed priority assignment by enumerating all the possible priority assignments and checking the feasibility of the resulting schedule on the schedulability interval defined in the paper. This can easily be encoded in the model, with minor modifications.

5 SCHEDMCORE execution environment

Once the program generated by PRELUDE has been proved schedulable, it can be executed on the multicore architecture target using the SCHEDMCORE execution environment.

5.1 SCHEDMCORE philosophy and objectives

The SCHEDMCORE runner, a.k.a. `lsmc_run`, requires several inputs:

- The specification of a task set: this can be done in two ways, either using a descriptive text file (SCHEDMCORE task file) or a dynamic library produced by the compilation of a PRELUDE-generated C file;

- The number of cores to be used;

- The scheduling policy: SCHEDMCORE currently offers four on-line policies (FP, GEDF, gLLF and LLREF) and an off-line sequencer, which sequences the task set from a static trace. The user can also easily create his own policy and plug it into the platform. A SCHEDMCORE scheduling policy is a dynamic library, which follows a specified API that allows it to be plugged into the runtime system. Adding a new policy simply consists in writting a C program that implements the API, compiling it as a dynamic library and specifying that the runner should be linked to this particular library. Thanks to the high level primitives provided by SCHEDMCORE, writting the policy itself is fairly simple. As an example, our implementations of the EDF and RM policies consist in a single C file, less than 50 code lines long.

We opted for a time-triggered platform, the pluggable scheduler wakes up at fixed time periods and forces tasks to consume all their WCET. This design makes the SCHEDMCORE runtime system simple, deterministic and impervious to scheduling anomalies [22].

The main target platform is Linux but it should be portable to any system that provides: a preemptive fixed-priority scheduler with at least 5 priority levels, some synchronization mechanisms (semaphores and/or barrier) and the ability to bind a thread to a particular processor (a.k.a. processor affinity). In this paper, we use a standard Linux system, with POSIX thread, `SCHED_FIFO` real-time scheduler and processor affinity support. The runtime is implemented as a user space library, which means that the Linux kernel does not have to be patched (as it can be with other solutions [17]). Thus, the maintenance of SCHEDMCORE does not need to follow the evolution of the underlying operating system and it can be ported to another OS that offers the same primitives.

5.2 SCHEDMCORE architecture

The SCHEDMCORE runtime architecture is shown in figure 3. It consists of 3 management threads, which control the user real-time tasks that execute in separate threads.

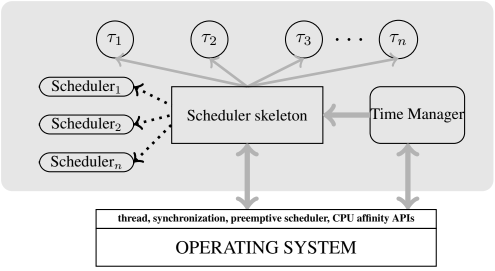


Figure 3. SCHEDMCORE **runtime architecture**

The *scheduler skeleton* is a POSIX thread, which manages the communication with the operating system and offers all necessary functions to manage task accesses to system resources. The execution of this entity is controlled by a semaphore. If the semaphore is unlocked the scheduler skeleton can preempt other tasks and performs all necessary actions for the task scheduling.

The *schedule* function of the pluggable scheduler implements one scheduling policy and takes scheduling decisions. It consists of a dynamic library that must provide 3 mandatory functions: `initialize`, `schedule` and `finalize`. The generic scheduler skeleton calls those functions at appropriate instant. Thus, systems instructions and scheduling instructions are separated which makes writing a new pluggable scheduler easier.

The *time manager* entity is woken up periodically at a fixed time interval by the operating system and it unlocks the scheduler skeleton semaphores to trigger the scheduling of the tasks. We chose to separate the *time manager* from the *scheduler skeleton* because we may decide (in further work) to release the scheduler at some other instants, or to use the usual event-based triggering.

The design of SCHEDMCORE requires 5 operating system preemptive priority levels, that define how each thread can preempt the others. The first (highest) is assigned to the time manager task, the second is assigned to generic scheduler task, the third is the priority corresponding to Execution task state, the fourth to Idle task state and the last one is the one used by the main program. The task states will be explained in the following section.

5.3 Execution with an on-line scheduler

The task model implemented in the SCHEDMCORE runner can be represented by the automaton of Figure 4.

A task is *Active* when it is not blocked by a precedence constraint and ready to run (released). This corresponds to the condition for task τ_i:

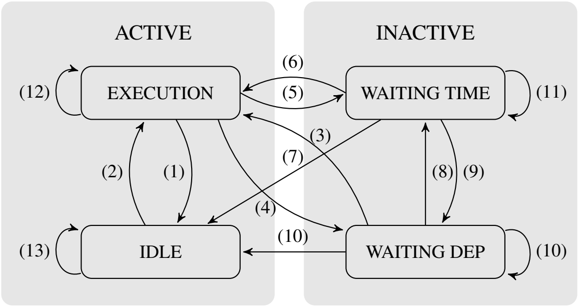


Figure 4. On-line task

$$active(\tau_i, t) = \begin{cases} \textit{false if } O^c_{\tau_i}(t) > 0 \textit{ or } C^c_{\tau_i}(t) = 0 \textit{ or } prec(\tau_i, t) \\ \textit{true} \quad \textit{otherwise} \end{cases}$$

$$\textit{where } prec(\tau_i, t) = true \Leftrightarrow \begin{cases} \exists j, k, \tau_{j.k} \rightarrow \tau_{i.(t-O_i)/T_i} \\ \wedge\big((C^c_j > 0 \wedge k = (t - O_j)/T_j) \\ \qquad \vee k > (t - O_j)/T_j\big) \end{cases}$$

The condition *prec* derives from Definition 1 and the remark that at time t, task τ_i executes the job $\tau_{i.(t-O_i)/T_i}$. An active task can either be in state *execution*, when it executes on one of the m CPU cores, or in *idle* state, when it is ready to run but all cores are already occupied.

Otherwise, the task is *Inactive*. An inactive task can either be in state *waiting_time*, if the task has not started yet (not released) or has already completed its current job execution, or in state *waiting_dep*, if the task is blocked by some precedence constraint.

The main work of a pluggable scheduler is to decide when to change the state of the concerned tasks, that it to say to define when the transitions of the task automata are fired. This is defined in the `schedule` function of the pluggable scheduler, which is called by the skeleton scheduler.

Therefore, `schedule` function computes the ordered queue which corresponds to *list_prio*. We use an additional function *pos_prio* which gives the position of an active task τ in *list_prio*. The conditions associated with the different states are:

- *Execution*: $active(\tau, t) \wedge pos\_prio(\tau, t) < m$;
- *Idle*: $active(\tau, t) \wedge pos\_prio(\tau, t) \geq m$;
- *Waiting Time*: *not* $active(\tau, t) \wedge$ *not* $prec(\tau, t)$;
- *Waiting Dep*: $prec(\tau, t)$

Two operating system primitives are used to manage the task executions, *semaphores* and *operating system priority*. Semaphores are used to distinguish between active tasks (semaphore unlocked) and inactive tasks (semaphore locked). The m tasks with highest priority (according to scheduling policy) are assigned the highest (user task) priority of the OS and can go to state *execution*. The remaining tasks will go to or remain in *idle* state.

5.4 Execution with an off-line scheduler (dispatcher)

The schedule of the system can also be computed by an off-line scheduler (called a dispatcher). This mode of execution uses a trace table to decide at each instant which task must execute on which CPU core. The execution trace is not computed by the SCHEDMCORE runtime, it is instead computed off-line using the methodology described Section 4. The `lsmc_tracer` tool takes an UPPAAL counter-example trace as input and generates a C file which contains the corresponding sequencing information. This file is compiled with a generic dispatcher C file to produce a dynamic library that conforms to the pluggable scheduler interface (as for on-line schedulers, a dispatcher is designed as a pluggable scheduler).

The automaton representing the execution of a task with such an off-line schedule is described Figure 5. A task can only be in two different states, *execution* or *waiting*. The execution table simply describes the transitions of the tasks at each instant. Transition (4) unlocks the task semaphore and the system priority of the task becomes that of executing tasks. Transition (2) locks the semaphore and the system priority of the task becomes that of waiting tasks. Transitions (1) and (3) basically have no effect.

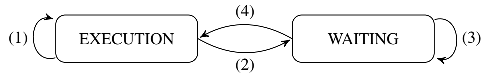


Figure 5. Off-line task

```
imported node Gyro_Acq(gyro, tc: int)
      returns (o: int) wcet 10;
imported node GPS_Acq(gps, tc: int)
      returns (o: int) wcet 10;
imported node FDIR(gyr, gps, str, gnc: int)
      returns (to_pde, to_gnc, to_tm: int) wcet 20;
imported node PDE(fdir, gnc: int)
      returns (pde_order: int) wcet 10;
imported node GNC_US(fdir, gyr, gps, str: int)
      returns (o: int) wcet 210;
imported node GNC_DS(us: int)
       returns (pde, sgs, pws: int) wcet 300;
imported node PWS(gnc: int) returns (pws_order: int) wcet 20;
imported node SGS(gnc: int) returns (sgs_order: int) wcet 20;

imported node Str_Acq(str, tc: int) returns (o: int) wcet 200;
imported node TM_TC(from_gr, fdir: int)
       returns (cmd: int) wcet 1000;


sensor gyro wcet 0; sensor gps wcet 0;
sensor str wcet 0; sensor tc wcet 0;
actuator pde wcet 0; actuator sgs wcet 0; actuator gnc wcet 0;
actuator pws wcet 0; actuator tm wcet 0;

node FAS(gyro: rate (100, 0); gps: rate (1000, 0);
         str:  rate (10000, 0); tc: rate (10000, 0))
returns (pde, sgs;gnc: due 300; pws, tm)
var gyro_acq, gps_acq, str_acq, fdir_pde,
    fdir_gnc, fdir_tm, gnc_pde, gnc_sgs, gnc_pws;
let
  gyro_acq = Gyro_Acq(gyro, (0 fby tm)*^100);
  gps_acq = GPS_Acq(gps, (0 fby tm)*^10);
  str_acq = Str_Acq(str, 0 fby tm);
  (fdir_pde, fdir_gnc, fdir_tm) =
     FDIR(gyro_acq, gps_acq*^10, (0 fby str_acq)*^100, (0 fby gnc)*^10);
  gnc=GNC_US(fdir_gnc/^10, gyro_acq/^10, gps_acq, str_acq*^10);
  (gnc_pde, gnc_sgs, gnc_pws)=GNC_DS(gnc);
  pde = PDE(fdir_pde, (0 fby gnc_pde)*^10);
  sgs = SGS(gnc_sgs);
  pws=PWS(gnc_pws~>1/2);
  tm = TM_TC(tc, fdir_tm/^100);
tel
```

Figure 6. The FAS in PRELUDE

policy	schedulable	time in UPPAAL	time in C
FP (default)	no	0.033s	0.000s
gEDF	yes	45.734s	0.010s
gLLF	yes	6.231s	0.029s
LLREF	no	0.071s	0.000s
OPT-FP	yes	out of memory	0.013s
OPT	-	out of memory	out of memory

Figure 7. Schedulability analysis of the FAS

6 Case study

We illustrate our framework on the FAS presented in Introduction. It can be programmed with PRELUDE as described Figure 6. The compiler generates a task set made up of of 19 tasks and 26 precedence constraints (the inputs and outputs are translated into tasks).

The results of the schedulability analysis of the task set, performed by SCHEDMCORE, are provided Figure 7. LLREF fails to schedule the task set because it tries to balance task execution without taking into account the urgency of a task that is due to its successors.

7 Conclusion

We described the design of an open end-to-end framework for the development of multi-periodic real-time systems targeting multicore architectures. We think that bridging the gaps between formal high-level view of a system and its real-time execution facilitates rapid prototyping and experimentation in the domain of multicore systems for time-critical applications. SCHEDMCORE aims at filling this gap using COTS operating system.

In the future, we will extend the PRELUDE language to integrate mode automata [9]. We will also provide automatic ways for aggregating nodes when there are too many. We will improve the performance of the off-line schedule generator and we will enrich the scheduling policy libraries.

References

[1] P. Aubry, P. Le Guernic, and S. Machard. Synchronous distribution of Signal programs. In *Proceedings of the 29th Hawaii International Conference on System Sciences (HICSS'96) Volume 1: Software Technology and Architecture*, pages 656–665, 1996.

[2] T. P. Baker and S. K. Baruah. Schedulability analysis of multiprocessor sporadic task systems. In *Handbook of Realtime and Embedded Systems*. CRC Press, 2007.

[3] G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In *4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[4] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[5] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the Sig-

nal language and its semantics. *Sci. of Compu. Prog.*, 16(2), 1991.

[6] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From simulink to scade/lustre to tta: a layered approach for distributed embedded applications. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, pages 153–162, 2003.

[7] H. Chetto, M. Silly, and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems*, 2, 1990.

[8] H. Cho, B. Ravindran, and E. D. Jensen. An optimal real-time scheduling algorithm for multiprocessors. *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS 2006)*, pages 101–110, 2006.

[9] J.-L. Colaço, B. Pagano, and M. Pouzet. A conservative extension of synchronous data-flow with state machines. In *Proceedings of the 5th ACM International Conference On Embedded Software (EMSOFT'05)*, pages 173–182, 2005.

[10] M. Cordovilla, F. Boniol, E. Noulard, and C. Pagetti. Multiprocessor schedulability analyser. In *Proceedings of the 26th ACM Symposium on Applied Computing (SAC'11)*, 2011.

[11] L. Cucu and J. Goossens. Feasibility intervals for multiprocessor fixed-priority scheduling of arbitrary deadline periodic systems. In *Proceedings of the conference on Design, automation and test in Europe (DATE'07)*, pages 1635–1640, San Jose, CA, USA, 2007. EDA Consortium.

[12] L. Cucu-Grosjean and J. Goossens. Exact schedulability tests for real-time scheduling of periodic tasks on unrelated multiprocessor platforms. *Journal of Systems Architecture - Embedded Systems Design*, 57(5):561–569, 2011.

[13] A. Curic. *Implementing Lustre programs on distributed platforms with real-time constraints*. PhD thesis, Université Joseph Fourier, Grenoble, 2005.

[14] A. David, J. Illum, K. G. Larsen, and A. Skou. *Model-Based Design for Embedded Systems*, chapter Model-Based Framework for Schedulability Analysis Using UPPAAL 4.1, pages 93–119. CRC Press, 2010.

[15] R. I. Davis and A. Burns. A survey of hard real-time scheduling algorithms and schedulability analysis techniques for multiprocessor systems. techreport YCS-2009-443, University of York, Department of Computer Science, 2009.

[16] F.-X. Dormoy. Scade 6 a model based solution for safety critical software development. In *Embedded Real-Time Systems Conference (2008)*, 2008.

[17] D. Faggioli, F. Checconi, M. Trimarchi, and C. Scordino. An edf scheduling class for the linux kernel. In *Proceedings of 2009 Real Time Linux Workshop*, 2011. Revised version.

[18] J. Forget. *A Synchronous Language for Critical Embedded Systems with Multiple Real-Time Constraints*. PhD thesis, Université de Toulouse - ISAE/ONERA, Toulouse, France, Nov. 2009.

[19] J. Forget, F. Boniol, E. Grolleau, D. Lesens, and C. Pagetti. Scheduling dependent periodic tasks without synchronization mechanisms. *In 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'10)*, April 2010.

[20] J. Forget, F. Boniol, D. Lesens, and C. Pagetti. A multi-periodic synchronous data-flow language. In *11th IEEE High Assurance Systems Engineering Symposium (HASE'08)*, Nanjing, China, Dec. 2008.

[21] A. Girault, X. Nicollin, and M. Pouzet. Automatic rate desynchronization of embedded reactive programs. *ACM Trans. Embedded Comput. Syst.*, 5(3):687–717, 2006.

[22] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17:416–429, 1969.

[23] N. Guan, Z. Gu, M. Lv, Q. Deng, and G. Yu. Schedulability analysis of global fixed-priority or edf multiprocessor scheduling with symbolic model-checking. In *Proceeding of the 11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'08)*, pages 556–560, 2008.

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

[25] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming (PLILP '91)*, Passau, Germany, 1991.

[26] K. S. Hong and J. Y. T. Leung. On-line scheduling of real-time tasks. *IEEE Transactions on Computers*, 41:1326–1331, 1988.

[27] P. Li, B. Ravindran, S. Suhaib, and S. Feizabadi. A formally verified application-level framework for real-time scheduling on posix real-time operating systems. *IEEE Trans. Softw. Eng.*, 30:613–629, September 2004.

[28] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environmet. *Journal of the ACM JACM*, 20(1):46–61, 1973.

[29] B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, USA, 2002.

[30] M. Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, 2006.

[31] M. A. Rivas and M. G. Harbour. A POSIX-Ada Interface for Application-Defined Scheduling. In *International Conference on Reliable Software Technologies, Ada-Europe 2002*, pages 136–150, 2002.

[32] C. Sofronis, S. Tripakis, and P. Caspi. A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling. In *Proceedings of the 6th International Conference on Embedded Software (EMSOFT'06)*, pages 21–33, Seoul, South Korea, Oct. 2006.

[33] The Mathworks. *Simulink: User's Guide*. The Mathworks, 2009.

[34] S. Tripakis, C. Pinello, A. Benveniste, Albert Sangiovanni-Vincentelli, P. Caspi, and M. Di Natale. Implementing synchronous models on loosely time-triggered architectures. *IEEE Transactions on Computers*, 57(10):1300–1314, Oct. 2008.

[35] R. Urunuela, A.-M. Déplanche, and Y. Trinquet. STORM, simulation tool for real-time multiprocessor scheduling. Technical report, Institut de Recherche en Communications et Cybernétique de Nantes, september 2009.