The ROSACE Case Study: From Simulink Specification to Multi/Many-Core Execution

Claire Pagetti*, David Saussié†, Romain Gratia*, Eric Noulard*, Pierre Siron*
*ONERA - Toulouse, France † Polytechnique Montréal - Canada

***Abstract*—This paper presents a complete case study - named ROSACE for Research Open-Source Avionics and Control Engineering - that goes from a baseline flight controller, developed in MATLAB/SIMULINK, to a multi-periodic controller executing on a multi/many-core target. The interactions between control and computer engineers are highlighted during the development steps, in particular by investigating several multi-periodic configurations. We deduced ways to improve the discussion between engineers in order to ease the integration on the target. The whole case study is made available to the community under an open-source license.**

I. INTRODUCTION

The purpose of the paper is twofold: first, to provide an open-source avionic control engineering case study[1] that can be used as a benchmark, and second, to illustrate a way of translating such a high level SIMULINK [1] specification down to a multi-threaded code executing on a multi/many-core target that is compliant with the high level requirements This case study is analyzed with respect to real-time implementation and ways to reduce as much as possible the effort on the integration while preserving the correct behaviour.

A. Design of a parallel flight controller

We rely on a standard avionic development process but use recent languages and tools to design a parallel flight controller on a challenging to embed target. It is of paramount importance to prepare the embedding of multi/many-core COTS [2], [3] because they will be the only available processors on the market and because they dramatically lack of predictability. The design of a flight controller works as follows:
Step 1: Production of a multi-periodic controller. A multi-periodic flight controller is developed in SIMULINK around a given operating point [4]. The methodology to obtain such a controller is described in Section II-A. Controllers are usually verified and validated against several properties (i.e. stability, performance, robustness). Since our objective is to validate the real-time aspects, we mainly focus on *time-domain performance specifications* on both the transient response and the steady-state response. Four types of properties are analyzed on the system response to a step input:

P1 : settling time, that is the time required to settle within 5% (resp. 1% or 2%) of the steady-state value;
P2 : overshoot, that is the maximum value attained minus the steady-state value;
P3 : rise time, that is the time it takes to rise from 10% to 90% of the steady-state value;
P4 : steady-state error, that is the difference between the input and the output for a prescribed test input as $t \to \infty$.

The time-domain performance properties are illustrated in the figure 1 for a step input. At this stage, these properties are analyzed through SIMULINK simulations.

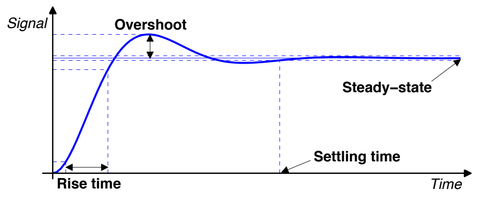

Figure 1. Performance properties

Step 2: Coding. The discrete SIMULINK specification is then translated within the PRELUDE/SCHEDMCORE framework. To do so, each block, executing at a given rate, is translated as a sequential C code and the multi-periodic assembly is translated into a PRELUDE program [5]. Currently, those translations are manual but future work could consider automatic translation using tools detailed in Section V.

The designer can then simulate the code with the SCHEDMCORE toolbox [6]. The code has been instrumented in order to generate SIMULINK-compliant traces, so that the designer can compare the tracings obtained by the simulation of the implementation with those of the high level design. Several assembly versions can be constructed by varying the periods and the precedence constraints in order to ease the integration. This stage is described in Section III.
Step 3: Validation on the target. Finally, the designer can integrate the implementation on the real target. To do so, the multi/many-core must be used in a predictable way by relying for instance on an appropriate *execution model* [7]. Such a model is a set of rules to be followed by the designer in order to avoid, or at least reduce, unpredictable behaviours. In this work, we reuse some ideas from the literature: off-line non preemptive partitioned schedule, static storage of code and variables in the caches, explicit communication using the network on chip (NoC). The experiments have been made on the TILERA TILEMPOWERGX-36 platform [8].

To validate the performances with regard to the environment dynamics, there are mainly three approaches: (1) hardware-in-the-loop validation; (2) connecting the controller executing on

[1]The complete case study can be found on the svn repository https://svn.onera.fr/schedmcore/branches/schedmcore-RTAS2014/Case_Study_RTAS.

the multi/many-core with the SIMULINK dynamic model; (3) implementing the dynamics model on the multi/many-core as well with a sufficiently high frequency to represent a *continuous dynamics*. We have chosen the last solution because the timings to interconnect the controller and the aircraft dynamics on the many-core are small and bounded, while interfacing the target with SIMULINK would not be easy to prove correct. Again tracings obtained during the real execution are compared with the high level requirements. This stage is depicted in Section IV.

B. Lessons learned

The described experiments helped us improving our understanding of the difficulties to (1) discuss between control engineers and computer scientists and (2) to highlight the link between the high level design and the low level real-time choices.

a) Where do antagonist requirements come from?: From the control engineers point of view, the more close to the real dynamics the controller is, the more confidence he will have in the result. Being close to the real dynamics means executing controller sub-functions in sequence as fast as possible (generating precedences) and as often as possible (generating high frequencies). This results in very strong real-time constraints for the integration. From the integrators point of view, the less severe the real-time requirements are, the safer the integration will be. Indeed, in practice, reducing frequencies decreases the CPU usage, freeing time for other applications. Reducing precedences among tasks increases the schedulability. Therefore, a compromise between both sides must be found.

b) Ease the discussion: The taxonomies and concerns differ in the two worlds. Control engineers consider (1) no resource limitation in general. They are however aware that delays will be introduced by the communication network (between sensors/actuators and CPUs) and that restrictions on the frequencies will be imposed by the CPUs; (2) properties such as stability, robustness and performances; (3) validation and verification on the closed loop. Computer engineers consider (1) mainly the provisioning of the finite resources; (2) local properties such as WCET computation, schedulability and response time; (3) real-time analyses on the controller solely. In particular, they do not handle the high level properties, such as the properties **P1**-**P4** of our flight controller.

There are two ways to ease the discussion. The first consists in providing tools and methods to the control engineers to precisely determine the real-time behaviours of the controller (WCET or schedulability). Such an approach does not exist yet, but there are good practices. For instance, standard controllers avoid jitter because they are supposed to degrade the control performance. The second approach consists in maintaining a common view during the development by taking into account the high-level properties at each development step. We follow the second way by offering a first common information in the form of tracings. This allows to quickly analyse the behaviour of several low-level designs to check if the performance properties are still valid.

c) Where compromise can be found: A civil flight controller is quite robust and can support a relaxed implementation as illustrated in the paper. In the future, flight controllers will be more reactive due to the use of composite structure, the reduction of fuel consumption and the intensification of the traffic. On the other hand, achieving a predictable implementation on next-generation processors will be rather difficult. These developments will increase the role and the complexity of finding a compromise. Therefore, the design of flight controllers will require more automatic methods and tools.

II. CASE STUDY: LONGITUDINAL FLIGHT CONTROLLER

We consider the longitudinal motion of a medium-range civil aircraft in *en-route* phase, specifically the *cruise* and *change of cruise level* subphases [9]. During the *cruise* subphase, the autopilot maintains a constant altitude h (actually a specific flight level FLxxx[2]) while the autothrottle (A/T) maintains the airspeed V_a. During a *change of cruise level* subphase (i.e. a step climb), the autopilot commands a constant vertical speed V_z (rate of climb), till capturing the new flight level. These changes of flight level are mainly for fuel economy reasons; the flight management system (FMS) executes step climbs of 2000 ft, or even 4000 ft, when appropriate (e.g. FL300 $\rightarrow$ FL320 $\rightarrow$ FL340 $\rightarrow$ FL360, figure 2).

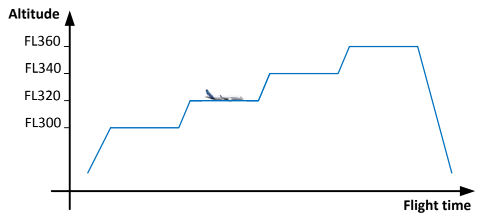

Figure 2. Step climbs in *en-route* phase

A. Recap on flight control system design

The electronic flight control system remains a challenging part of an aircraft design. As the aircraft dynamics vary significantly within its flight envelope[3], a single static controller is generally insufficient to ensure stability and performance on the whole operating domain. To this end, the controller must somehow "evolve" with the flight condition [9]. For decades now, the engineers have resorted to gain-scheduling techniques to design electronic flight control systems [10], [11]. Essentially, the gain-scheduling approach consists of choosing a finite set of operating points (i.e., flight conditions) distributed throughout the flight envelope and designing a corresponding set of linear controllers to locally achieve stability and performance. Afterwards, to fully cover the operating domain, these linear controllers are interpolated with scheduling variables representative of the flight condition. The overall stability and performance are finally assessed

[2]For example, FL300 denotes a pressure altitude of 30000 ft.

[3]The operating domain where the aircraft can be flown, generally defined in terms of altitude and Mach number.

by different mathematical methods and extensive time and frequency validations (e.g., Monte-Carlo method). This is nevertheless out of the scope of the paper.

As the common practice in automatic control is to design continuous-time (*analog*) controllers, the flight control laws must be digitalized in order to be implemented on the on-board computers. This implies the choice of adequate sampling periods. First the sampling period must be lower than the system delay margin, that is the maximum pure delay that the system can withstand before destabilizing. Moreover, to preserve performance, one should ideally choose a sufficiently low sampling period to reproduce as much as possible the behaviour of the continuous-time controller.

B. *Description of the case study*

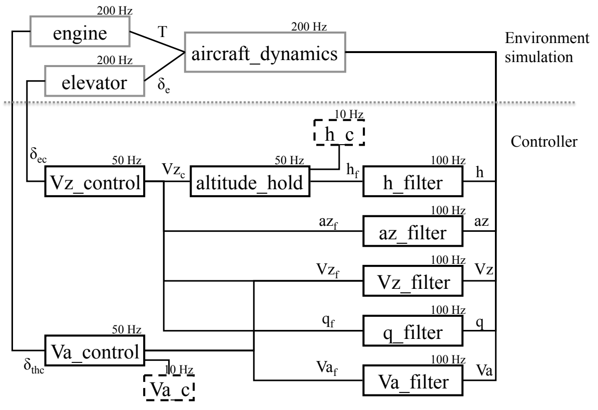

Figure 3. Controller design

The case study is a multi-periodic extension of the mono-periodic longitudinal control of Gervais and al [12]. A simple yet representative longitudinal flight controller has been designed in the MATLAB/SIMULINK environment for the flight condition ($h = 10000\,\mathrm{m}$, $V_a = 230\,\mathrm{m/s}$) which corresponds to an average cruise condition. The controller was verified in the continuous-time domain by studying the behaviour of the aircraft in the neighbourhood of this flight condition. However, the controller is not scheduled, meaning that it is likely to perform poorly far from this flight condition.

The SIMULINK scheme in Figure 3 is actually the discretization of our original SIMULINK scheme. It is divided into two parts: on the one hand, the *Environment Simulation* part represents the real system that is to be controlled, that is the aircraft as well as the engines and elevators, and, on the other hand, the *Controller* part gathers the control loops (`altitude_hold`, `Vz_control`, `Va_control`) as well as filters. The goal of the longitudinal flight controller is to track accurately altitude, vertical speed and airspeed commands (resp. h_c, V_{z_c} and V_{a_c}). The airspeed control is handled by the `Va_control` loop that maintains or tracks the desired airspeed V_{a_c}. The altitude control is split in two stages; an altitude command h_c is first translated into a vertical speed command V_{z_c} by the `altitude_hold` loop and the `Vz_control` loop then tracks V_{z_c}. During a step climb, the controller logic is as follows: a constant vertical speed command ($V_{z_c} = 2.5\,\mathrm{m/s}$ is first imposed so the aircraft gains altitude, then, within $50\,\mathrm{m}$ of the target flight level, the controller switches back to the altitude hold function to capture the commanded altitude and to travel the last meters. This ensures a climb at a low constant flight path angle, so the passengers will not experience any discomfort. Without this logic, a very steep climb could result from a direct altitude demand.

The considered outputs are listed in Tab. I and are measured by dedicated sensors. They are modelled as low-pass filters with bandwidth reflecting the nature of the measured signals.

Table I
VARIABLES

Outputs	V_z	vertical speed
	V_a	true airspeed
	h	altitude
	a_z	vertical acceleration
	q	pitch rate
Filtered outputs	V_{z_f}	vertical speed
	V_{a_f}	true airspeed
	h_f	altitude
	a_{z_f}	vertical acceleration
	q_f	pitch rate
Reference inputs	h_c	altitude command
	V_{a_c}	airspeed command
Commanded inputs	V_{z_c}	vertical speed command
	δ_{e_c}	elevator deflection command
	δ_{th_c}	throttle command
Aircraft inputs	δ_{e_c}	elevator deflection
	T	engine thrust

Discretization of the components Each component of the *Environment Simulation* part is modelled by Ordinary Differential Equations (ODEs), usually as continuous-time nonlinear state equations of the form $\dot{x} = f(x, u, t)$ with state vector x, input vector u and time t. They are approximately solved by numerical methods like Euler or Runge-Kutta integration methods with fixed or variable time steps. Usually, the smaller the time step is, the more precise but time-consuming the solution is.

Control engineers do not resort to the same approach to digitalize their controllers. Indeed, from a programming perspective, it is inconvenient to implement a controller with a numerical integration routine such as Runge-Kutta method. Moreover the discretization must preserve frequency-domain characteristics as much as possible, so the performance and stability requirements are still met. Therefore, dedicated techniques [13] other than numerical integration are used to convert a continuous-time controller $K(s)$ to its discrete-time version $K(z)$; these techniques all lead to difference equations. Among these techniques, the bilinear transformation (also known as Tustin's method) and the zero-order holder method are the most popular ones. Moreover filters with specific properties (e.g., bandwidth) can be designed directly in the digital domain.

Rate choices The closed-loop system with the continuous-time controller can roughly tolerate a pure time delay of $1\,\mathrm{s}$ before destabilizing. The sampling period must then be chosen lower than $1\,\mathrm{s}$ (1 Hz rate). Nevertheless, as we are not only interested in preserving stability but performance as well, the sampling period should be much lower, for instance $100\,\mathrm{ms}$ (10 Hz rate). Considering realistic rates used in industry, the three controller blocks are first digitalized with a $20\,\mathrm{ms}$ sampling period (50 Hz rate) whereas the filters work at a rate of 100 Hz to feed the data. Finally, as the environment (aircraft+elevator+engine) is supposed to model a continuous-time phenomenon, a greater rate of 200 Hz is used.

C. Validation objective

The design process first focuses on the internal V_a and V_z loops (resp. `Va_control` and `Vz_control` blocks). We analyse the properties **P1** to **P4** for separate step demands in V_a and V_z. Moreover the two outputs should be decoupled, that is a demand in V_a should slightly affect V_z, and vice versa. Figure 4 illustrates time-responses for separate step inputs obtained with SIMULINK. On the top figures, an airspeed variation of $5\,\mathrm{m/s}$ from the initial airspeed $V_a = 230\,\mathrm{m/s}$ is first commanded, while V_z lies around 0. On the bottom figures, a vertical speed demand of $V_{z_c} = 2.5\,\mathrm{m/s}$ leaves airspeed V_a almost invariant. The demand $V_{z_c} = 2.5\,\mathrm{m/s}$ will be the maximum vertical speed commanded by the altitude controller. The quantified objectives are summarized in Tab. II.

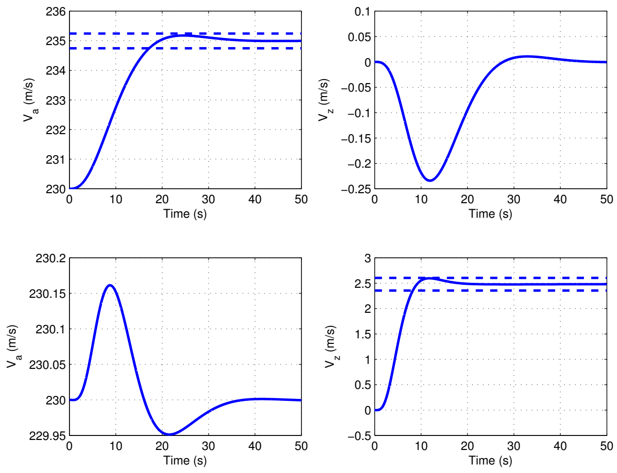

Figure 4. Time-responses and decoupling for separate solicitations in $V_{a_c} = 235\,\mathrm{m/s}$ and $V_{z_c} = 2.5\,\mathrm{m/s}$

Table II
REQUIREMENTS AND VERIFICATION IN SIMULINK

Property	Objective		Results in SIMULINK
P1 5% settling time	V_z	$\leq 10\,\mathrm{s}$	$8.22\,\mathrm{s}$
	V_a	$\leq 20\,\mathrm{s}$	$17.22\,\mathrm{s}$
P2 Overshoot	V_z	$\leq 10\%$	4.72%
	V_a	$\leq 10\%$	3.65%
P3 Rise time	V_z	$\leq 6\,\mathrm{s}$	$5.09\,\mathrm{s}$
	V_a	$\leq 12\,\mathrm{s}$	$11.6\,\mathrm{s}$
P4 Steady-state error	V_z	$\leq 5\%$	0.83%
	V_a	$\leq 5\%$	0.11%

The steady-state error (**P4**) for the decoupled approach is considered as correct. This property is however analysed on a step climb. Figure 5 illustrates a step climb of $1000\,\mathrm{m}$ asked at $t = 50\,\mathrm{s}$. During the first 50 seconds, the aircraft maintains an altitude of $10\,\mathrm{km}$ and an airspeed of $230\,\mathrm{m/s}$. As the new commanded altitude ($11\,\mathrm{km}$) is too high, the autopilot first commands a constant vertical speed of $V_{z_c} = 2.5\,\mathrm{m/s}$ (top right). The aircraft begins its ascent (top left) at constant vertical speed (bottom right). At $10950\,\mathrm{m}$ ($t = 437\,\mathrm{s}$), the controller logic switches back to altitude hold mode and smoothly brings the aircraft to $11000\,\mathrm{m}$ with very slight overshoot. During the whole maneuver, the airspeed V_a stays around $230\,\mathrm{m/s}$ as desired.

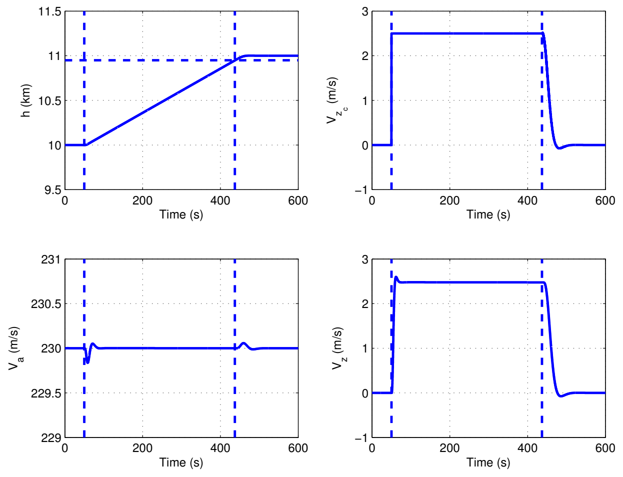

Figure 5. Step climb of $1000\,\mathrm{m}$

III. IMPLEMENTATION

This section describes the coding in C+PRELUDE. We illustrate how the control and computer engineers can interact in order to simplify the integration. This can be reached by investigating several multi-periodic configurations where variations are made on the frequencies and the precedence constraints.

A. Coding of the basic blocks

Each basic block is manually translated as C code in order to obtain a simple coding and a complete traceability. However, any automatic translation could work as long as the code can be parametrized by the sampling period T_s. Note that we use the same discretization methods as those selected in the SIMULINK model.

The components of the *Environment Simulation* are discretized with *Forward Euler* method, as it has a much simpler form than any other integration method. Moreover, the results are sound with this approach. The sampling period T_s is explicitely represented with a fixed integration step $\Delta = 0.005$ ms (for 200Hz).

The discretization of the three controllers is simple as the only dynamic element is an integrator $1/s$, which is usually discretized with forward difference $T_s/(z-1)$. Therefore, the sampling period T_s is explicit.

The discretization method that we used for the filters is the zero-order hold approximation. As before, the discrete models are implemented as difference equations, the coefficients of which depend on the sampling period T_s. The relationship between coefficient and T_s is complex, which means that the discretized filter must be computed again for a new choice of sampling period.

B. Coding of the assembly

The multi-periodic code generated by the SIMULINK toolbox contains too many implicit choices. A better suited approach should provide (1) explicit description (e.g. of the communication) and (2) independence with regards to the real execution, in the sense that the functional results must always be the same for a given input whatever the real-time execution applied by the executive layer.

PRELUDE [5][4] is a formal language designed for this purpose. It belongs to the category of synchronous data-flow languages [14] and focuses on the real-time aspects of multi-periodic systems. From a PRELUDE program the compiler generates a set of dependent periodic tasks that preserves the semantics of the original program. The preservation of the semantics is warranted so that consuming task instances always use data produced by the correct producing task instance. This property is ensured thanks to two mechanisms: first, precedence encoding enforces that a consuming task cannot execute before the end of the producer, and second, a communication buffer-based protocol similar to [15] is implemented.

PRELUDE reuses many concepts from the synchronous data-flow language LUSTRE [16]. The variables and expressions of a program denote infinite sequences of values called *flows*. Each flow is accompanied with a *clock*, which defines the instant during which each value of the flow must be computed. PRELUDE follows a *relaxed synchronous hypothesis* (introduced by [17]), which states that computations must end before their next *activation*. A program consists of a set of equations, structured into *nodes*. The equations of a node define its output flows from its input flows. It is possible to define a node that includes several subnode calls executed at different rates.

PRELUDE aims at integrating functions that have been programmed in another language. These imported functions must first be declared in the program. All the basic blocks of the case study are in particular declared as imported nodes. The syntax is the following: `imported node Va_filter(Va:real) returns(Va_f:real) wcet X` It consists of the signature of the node (type and number of inputs and outputs) with a WCET. At this stage we may not know this value, so we keep it undetermined. Imported node calls follow the usual data-flow semantics: an imported node cannot start its execution before all its inputs are available and produces all its outputs simultaneously at the end of its execution. Those imported nodes become the tasks populating the task set.

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

PRELUDE adds real-time primitives to the synchronous data-flow model. Those operators can decelerate, accelerate or offset flows. Real-time operators are formally defined using *strictly periodic clocks*. A *strictly periodic clock* is a sequence of instants that can be defined as a pair *(period, offset)*. The basic clock, defined as $(1, 0)$, is the fastest clock and all *strictly periodic clocks* are derived relatively from it. We choose the basic clock with a period of $100\mu s$. This choice is left to the integrator: it must take into account the real-time attributes of the tasks and the performance of the multi/many-core target. In our case, the WCETs are very low (couples of μs), communication on the NoC takes less than $35\mu s$ (if the mapping respects some rules detailed in Section IV), and the periods must be a multiple of the basic clock. Therefore the tightest basic clock is $50\mu s$. Any other basic clock must be greater than $50\mu s$ (imposed by the platform) and must be a divisor of $5000\mu s$ (imposed by the application). It can be useful to reduce the basic clock if the schedule fails since WCETs are multiple of the basic clock.

The reference inputs h_c and V_{a_c} become inputs flows of the node. Those inputs are assumed at 10Hz (cf Figure 3) and are therefore associated with the clock $(1000, 0)$. The assembly expressed below is equivalent to the SIMULINK design.

```
node assemblage (h_c,  Va_c: real rate (1000,0))
returns (delta_e_c, delta_th_c : real)
var Va, Vz, q, az, h: real;
  Va_f, Vz_f, q_f, az_f, h_f : real;
  Vz_c, delta_e, T : real;
let
  h_f = h_filter(h/^2);
  Va_f = Va_filter(Va/^2);
  Vz_f = Vz_filter(Vz/^2);
  q_f = q_filter(q/^2);
  az_f = az_filter(az/^2);
  Vz_c = altitude_hold (h_f/^2,  h_c*^5);
  delta_th_c = Va_control
        (Va_f/^2, Vz_f/^2, q_f/^2,  Va_c*^5);
  delta_e_c = Vz_control
        (Vz_f/^2, Vz_c, q_f/^2, az_f/^2);
  T = engine (delta_th_c*^4);
  delta_e = elevator (delta_e_c*^4);
  (Va, Vz, q, az, h)= aircraft_dynamics
        ((0.018645918123716 fby delta_e), (43219.8575 fby T));
tel
```

The commands produced by the controller, δ_{e_c} and δ_{th_c}, become outputs of the node. It is mandatory to express the rate of the input while the rate of outputs is inferred by the compiler. All other variables are declared as intermediate flows. Then, after keyword *let* all the equations are written. The first h_f = *h_filter(*$h/\hat{}2$*)* states that the node *h_filter* produces the variable h_f and consumes the flow $h/\hat{}2$ which is the deceleration by 2 of the flow h. Since h is produced by the node *aircraft_dynamics*, we can deduce that *h_filter* runs twice slower than *aircraft_dynamics*. This respects the proportionality of the frequencies in Figure 3 where *aircraft_dynamics* is at 200Hz and *h_filter* at 100Hz. We can compute the clock of *h_filter* from the sixth equation: V_{z_c} = *altitude_hold* ($h_f/\hat{}2$, $h_c*\hat{}5$). The node *altitude_hold* consumes $h_c*\hat{}5$; thus it runs 5 times faster than the input h_c. Since the node *altitude_hold* consumes also $h_f/\hat{}2$, we deduce that *altitude_hold* executes twice slower than *h_filter*. We deduce from the equations the following relationship (compliant with Figure 3):

Node	Frequency	Clock	Node	Frequency	Clock
altitude_hold	50Hz	$(200, 0)$	*h_filter*	100Hz	$(100, 0)$
Va_filter	100Hz	$(100, 0)$	*q_filter*	100Hz	$(100, 0)$
az_filter	100Hz	$(100, 0)$	*Va_control*	50Hz	$(200, 0)$
Vz_control	50Hz	$(200, 0)$	*engine*	200Hz	$(50, 0)$
elevator	200Hz	$(50, 0)$	*ac_dynamics*	200Hz	$(50, 0)$

C. *Variations in the assembly to relax real-time constraints*

This first specification is very constrained in term of real-time and the effort of integration will be stringent. Indeed, we have reduced the frequencies by implementing a multi-periodic controller instead of designing a mono-periodic that runs at 200 Hz. This decreases the CPU usage but the precedence constraints imposed by the communication force several tasks to execute in a very short interval. For instance, the functional chain $h \rightarrow h_f \rightarrow V_{z_c} \rightarrow \delta_{e_c} \rightarrow \delta_e$ produces the execution shown in Figure 6. The sequence of tasks *aircraft_dynamics*, *h_filter*, *altitude_hold*, *Vz_control*, *elevator* must execute in less than 5ms every 20 ms.

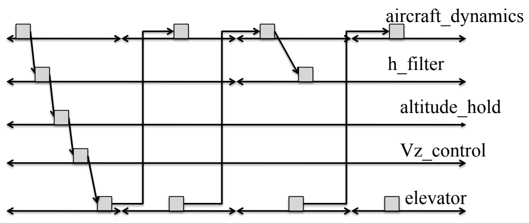

Figure 6. Functional chain in the specification

Relaxing precedences. To relax the effort of integration, the second precedence patterns in Figure 7 are better suited. Indeed, the system becomes more parallelisable and/or WCET of tasks with lower frequency can be increased. The unique difference for coding the assembly in PRELUDE (named in that case *assemblage_v2*) stands in the two equations:

```
T = engine ((1.640222296162316 fby delta_th_c)*^4);
delta_e = elevator ((0.018645918123716  fby delta_e_c)*^4);
```

Note that producing the same behaviour in SIMULINK is rather complex. Note also that breaking the dependencies is done with the `fby` operator, borrowed from LUCID SYNCHRONE [18], that delays a flow by one period.

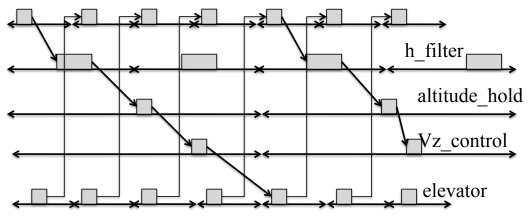

Figure 7. Relaxed functional chain

Relaxing periods. As illustrated previously, modifying the assembly is very easy in PRELUDE. Since it is also simple to generate the tracings, the control engineer can easily verify if the controller performances are fulfilled. This opens the room for testing many solutions.

In this paper, we will try three more assemblies that reduce periods and precedences. Next assembly (named *assemblage_v3*) reduces some frequencies: *h_filter*, *Va_filter*, *Vz_filter* are at 50Hz and *altitude_hold* is at 10Hz. For the three filters, we have to change the rate of the input flow as follows:

```
h_f = h_filter(h/^4);
```

For *altitude_hold*, the equation becomes

```
Vz_c = altitude_hold (h_f/^5,  h_c);
```

The equation of *Vz_control* has also to be modified as follow:

```
delta_e_c = Vz_control(Vz_f, Vz_c*^5, q_f/^2, az_f/^2);
```

In this version, dependencies are direct as in Figure 6, while *assemblage_v4* reuses those of Figure 7. In *assemblage_v41*, we changed the dependencies while keeping the frequencies. In *assemblage_v5*, we put *Vz_control* and *Va_control* at 25 Hz.

D. *Validation of the coding*

All the previous assemblies have been simulated on a PC using SCHEDMCORE framework. To do so, we rely on the simulator *lscm_run-nort* provided with SCHEDMCORE[5]. To assess the correctness of the various assemblies, we compared the time-responses to the same step climb simulated. Figure 8 superimposes the results of SIMULINK (Figure 5) and those of all the assemblies for the same autopilot instructions. Overall, there is no significant degradation and only the settling time is slightly impacted by the modifications. Indeed, we play on the rates and precedences that impact the response-times but not the functional values (no significant impact on properties **P2** and **P3**). Finally, we observe that the controller is robust to the relaxations. Since all results are satisfying, the different assemblies can be integrated on the target and assessed at this level.

IV. EXPERIMENTS

This section illustrates the feasibility of implementing the parallel flight controller on a multi/many-core. This shows that the abstract hypotheses made at the SIMULINK and the PRELUDE levels are reasonable. In particular, the basic clock, the relaxed synchronous hypothesis and the communication protocol can be implemented. We therefore describe in this section the porting of the code on the TILERA TILEMPOWERGX-36. We impose an *execution model* to enforce predictability on the platform in order to respect as much as possible the safety avionic standards.

A. TILERA *description*

The platform is equipped with a tiled microprocessor composed of 36 tiles. Each tile can communicate with other tiles, the peripherals and the external memory through a network on chip (NoC). The external memory is composed of 32 GBits

[5]SCHEDMCORE is available on the svn repository https://svn.onera.fr/schedmcore/branches/schedmcore-RTAS2014

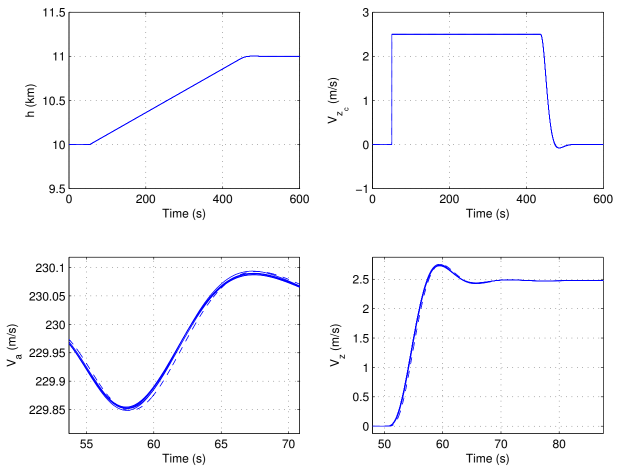

Figure 8. Step climb of $1000\,\mathrm{m}$ - Time-responses for the various assemblies and original SIMULINK (--)

DDR3 accessible through memory controllers. The grid is a 6x6 matrix of tiles as shown in the figure below (extracted from TILERA documentation [19]). Each tile is composed of the

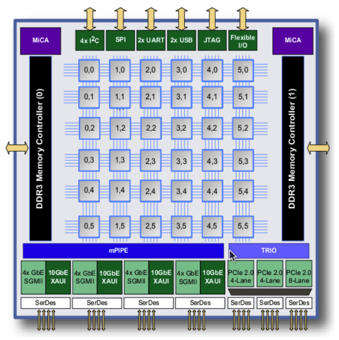

Figure 9. Scheme of the tile grids

following elements [19]: (1) a single core clocked at 1.2 GHz that owns two levels of cache (L1I-32KB, L1D-32KB, unified L2-256KB), (2) a switch that manages the communication over the network on chip, (3) a local clock accessible through the TILERA API *get_cycle_count*. The NoC is composed of five full-duplex sub-networks, each devoted to a particular type of exchange. The network Shared Dynamic Network (SDN) is the one used for exchanging data between tiles. All communications with the external memory go through the reQuest Dynamic Network (QDN) for write requests and through the Response Dynamic Network for read requests (RDN).

Three execution environments are provided with the platform: (1) standard SMP LINUX environment, (2) Zero Overhead LINUX (ZOL) or (3) bare-metal. The TILERA platform is not built for real-time systems but for high performance or networking applications. However, even if the most suitable environment is bare-metal, ZOL offers rather promising real-time predictability. The main features of ZOL [20, Chapter 7] are:

1) processors are isolated from interrupts, like the shielding approach promoted by [21]. The operating system does no longer interfere in the execution unless the application itself makes a system call. There is no interrupt handler;
2) there is a unique thread per processor. This ensures an applicative isolation of CPU and local caches resources.
3) a complete TILERA configuration can mix tiles in ZOL, in LINUX and bare-metal. In our experiments, a unique core is under LINUX to boot the chip and all other tiles are in ZOL, in particular those hosting the application.

Memory management. All environments support shared memory with builtin hardware coherency which may be disabled. For our experiments, we keep the shared memory active and we use one of the policies offered by TILERA for storing shared variables. The *cache homing* policy permits to alleviate the workload on the DDR by allocating each shared variable to a *home tile*. When tile t reads the variable, either the variable is in its own caches, otherwise instead of fetching the variable directly from the RAM memory, it asks the variable home tile. If the home tile has the data in its caches, it sends it directly to the requesting tile t. Otherwise, it is the home tile duty to fetch the variable from the RAM and then to send it to t. The writing works also by interacting with the home tile and invalidating the local caches containing the old value. Such pattern of exchanges is illustrated in Figure 10, extracted from TILERA documentation [20, §6].

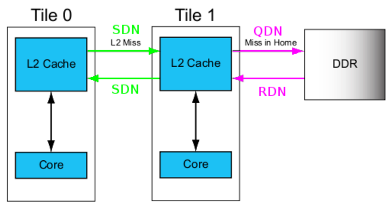

Figure 10. Cache homing policy

Stressing benchmarks. We made several benchmarks to assess the *predictability* of the TILERA platform. We first analysed the impact on the execution times when several tiles access concurrently the shared resources, such as DDR, local caches and NoC. We particularly focused on the time to access the local clock, to read/write data with the shared memory policies.

From the experimental observations, the mean time to access the local clock is 60ns and the maximum time is 400ns. The maximal value is rarely observed, around once every 10000 accesses. But we need to consider this value as the worst case. There is a real impact on the read and write access times when the number of concurrent tiles exceeds some bounds. Below these bounds, the times are low and repeatable. Above the bounds, a memory access can be delayed more than a second. To determine the bounds, we apply quite the same stressing benchmarks as for the Intel SCC [22]. An example of a benchmark is: several tiles (from 2 to 36) modify a shared variable hosted by a home tile and we measure the write access times for each writer, including the home tile.

Finally, we can deduce some rules on the mapping to avoid performance and predictability drops: (1) no more than 10 tiles must simultaneously access in writing the same [shared] cached memory location, (2) no more than 5 tiles must simultaneously access in writing the DDR, (3) no more than 30 tiles must simultaneously access in reading the same [shared] cached memory location.

B. Real-time implementation

To start with, the integrator must first assess the WCET of each task. Then, he/she must define an adequate execution model. Finally, a dispatcher compliant with the execution model must be developed for the TILERA TILEMPOWERGX-36.

1) WCET assessment: No static WCET analysis tool, such as ABSINT [23] or OTAWA [24], is available for the TILERA platform. Therefore, we used a measure-based approach, which is not safe in general but we could hardly do better at this stage. The evaluation was done on each task which was running in sequence and in isolation. We measured the execution time by surrounding the task call between two local clock reads. Because of the variability of the local clock access and to improve the reliability, we added some margin to the observed execution times.

It was decided in Section III-B that the basic clock should run at $100\mu s$. Therefore, WCET must be expressed as multiple of $100\mu s$. Moreover, inputs and outputs in PRELUDE are treated as sensors and actuators. This entails that they must be associated with a WCET. In our case, it could correspond to the delays generated by the bus between sensors, calculator and actuators. We imposed those values. WCETs are given in Tab. III.

Table III
WCET

Task	WCET	Task	WCET
aircraft_dynamics	200 μs	*elevator*	100 μs
altitude_hold	100 μs	*engine*	100 μs
h_filter	100 μs	*q_filter*	100 μs
Vz_filter	100 μs	*az_filter*	100 μs
Va_filter	100 μs	*Va_control*	100 μs
Vz_control	100 μs	*delta_e_c*, *Va_c*, *delta_th_c* *h_c*	500 μs

2) Execution model: Since we measured task execution times as if they were a sequential code running in isolation, we must use an execution model that fulfills those hypotheses. First, the schedule must be non preemptive to respect the sequential execution. It is best suited to avoid migration to reduce unexpected interrupt. Partitioning also improves predictability since it permits to uses a MIMD ("multiple instruction, multiple data") approach where the created binaries are specific to particular cores.

We tested two mappings, excluding the core 0 dedicated for initialization. Those mappings have been chosen manually. Any valid mapping can be computed using a constraint programming approach or a dedicated heuristic.

- map1: exactly one task on a tile;
- map2: grouping tasks with the same period on the same tile. For *assemblage* with the strongest precedences, we obtain the schedule shown Figure 11.

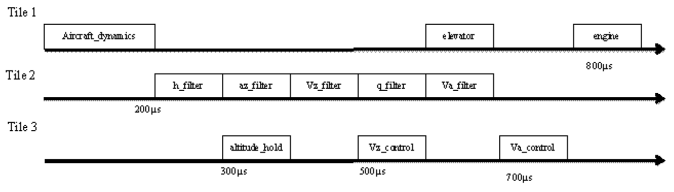

Figure 11. Off-line schedule - map2

Ensuring isolation is much more challenging. We promote the storage of code and data in the local caches as in [7], [25]. This prevents from unexpected applicative interactions. The tasks of the case study are small enough to fit in the caches. If this was not satisfiable, the designer must decompose, if possible, the code into smaller size pieces. Otherwise, a task that overflows the caches can run concurrently with locally stored tasks but not with other tasks that overflow the caches. The communication is done via shared memory and this contradicts the isolation hypothesis. However, since the mapping respects the bounds highlighted in Section IV-A, we can assume that the effect is negligible on the execution times. Note that, data could be exchanged with a message passing approach by relying on the User Dynamic Network (UDN), the performances of which are good for small size data. The home tile associated to a data produced by task t is the tile where task t executes.

The PRELUDE semantics imposes also to ensure precedence constraints between tasks. To fulfill this constraint, we choose a tick-based approach, that is scheduling decisions are taken only at discrete instants of a chosen granularity. We reuse the tick gap introduced in [22], in order to cope with the imperfect synchronization of local clocks and the communication delays. The idea consists in leaving a gap between the end of a job's termination and the beginning of the next tick. To do so, we add to each WCET a gap and in our case, the gap is $550ns$ where

$$35\ ns\ \textit{(communication delay)} + 500\ ns\ \textit{(clock precision)}$$

WCET given in Tab. III already contain the gaps.

3) Dispatcher implementation: The local clocks of the TILERA TILEMPOWERGX-36 are synchronous (i.e. no clock drift between the local clocks) but they are not perfectly synchronized because they do not boot at the same time. The offsets between the cores are not handled by the hardware and it is up to the user to manage a synchronization if needed. We have encountered the same problem on the Intel Single-chip Cloud Computer (SCC). The SCC bare-metal library we developed provides means to synchronize the core local clocks with a precision of 4 μs [22]. For the TILERA, we used another synchronization algorithm based on barrier that leads to a precision of 0.5 μs. It can hardly be reduced because of the worst time required to read the local clock. But in general, the observed precision is 50ns. The synchronization algorithm works as follows:

- N shared variables are homed on tile 0.
- 1 shared variable is homed on each tile.
- When tile i starts, it sets the N variables to 0. Then, the tile makes an active wait: as long as it did not receive any value on its own variable, it continuously sets to 1 the i-th of core 0.
- Tile 0 works differently. It sets all its homed variables to 0 and waits actively until all tiles are awaken. When this occurs, it sets the variables hosted by the other tiles to 1.
- When a tile detects that its local variable is set to 1, it starts a waiting of 1 s using its local clock. After this second, it reads the current time. This value becomes its local offset.
- Then the shared global time = local time − local offset.

Since no migration is allowed and since the task sequence is known in advance by every processor, we do not need timer interrupts for implementing the sequence. Each processor knows the static task sequence it has to run. When a processor needs to wait the next cycle, it does a busy wait (spin).

C. Results

We obtain almost the same results as those observed at PRELUDE level. They are also superimposed in Figure 8.

V. Related work

Control command and real-time: [26], [27] proposed a flexible real-time control system where the scheduler uses feedback from execution-time measures to adjust the periods in order to optimize the performances. Such a solution is not possible on non predictable target such as multi/many-core. The authors of [28] accept that WCETs are not computable, due to the new processors technologies. They analyse the end-to-end latencies of the control from sensors to actuators and show that some variability in execution times is acceptable.

Multi-periodic specification: Using a formal language for the description of dependent multi-rate task sets has been advocated by Baruah [29]. As already mentioned, SIMULINK [1] allows the description of multi-periodic systems in terms of blocks communicating through data-flows or events. However expressing complex communication patterns is difficult. Modelica also offers the possibility to describe multi-periodic assemblies [30]. The semantics relies on the use of strictly periodic clocks as in PRELUDE. The authors of [31] described a possible way to write multi-periodic SCADE systems but the toolbox has not been extended to consider such extension. The authors of [32] also consider SCADE extension with finite state automata in order to express multi-periodic systems.

Compilation of multi-periodic specifications: Each basic component can be automatically translated into C code: with SIMULINK toolbox; with GENEAUTO toolbox[6]; with a certified compiler as proposed in [33], [34]; or with an automatic SIMULINK to SCADE translator (as proposed in [35]), combined with the SCADE suite compiler. The multi-periodic configuration can be translated with SIMULINK, leading to a rate monotonic schedule. We can also mention the work of [36] that provides a sound semantics of SIMULINK operators. The authors of [37] particularly focus on the modular aspects which is a complementary aspect for an automatic translation.

The authors of [38] have implemented a multi-threaded compilation scheme for affine data-flow graphs, which also allow to specify multi-periodic assembly. Authors of [39] have developed a translation of SCADE programs to OASIS implementation.

Concerning the predictable use of multi/many-core for hard real-time, there is a large literature and we can mention the surveys [40] and [41].

[6]http://geneauto.gforge.enseeiht.fr/

VI. Conclusion

We have experimented the design of a parallel avionic longitudinal controller on a multi/many-core target. We have illustrated throughout a series of experiments what kind of discussion between control engineers and integrators can be leveraged, in order to find a compromise between both sides constraints.

The case study will be extended with a comprehensive flight control system that will operate on the whole flight envelope. Additional flight control laws will be integrated to cover the different flight modes. Validation will also be expanded to allow the observation of other criteria and possibly to allow Monte-Carlo simulations.

Future work will consider more automatic translation from the SIMULINK model, for instance by reusing and extending an existing tool [34]. PRELUDE language could also be improved by providing new features such as: (1) retrieving the strictly periodic clocks computed by the compiler in the imported node (indeed, the frequency of a discretized block has an impact on the code); (2) introducing the notion of "don't care" [42] in order to generate many assemblies. The authors of [43] use a MILP approach to determine where to introduce `fby` to break precedence constraints.

VII. Acknowledgement

The authors would like to thank the RTAS reviewers and Daniel Lohmann for their constructive comments and valuable suggestions that greatly improved the quality of the paper, as well as Tobias Klaus and Florian Franzmann for their comments and suggestions to improve on the usability of the actual case study. They are also grateful to Frédéric Boniol for his careful reading and helpful remarks.

References

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

[2] M. Gatti, "Development and certification of avionics platforms on multi-core processors," in *Tutorial Mixed-Criticality Systems: Design and Certification Challenges, Embedded Systems Week*, Montreal, Canada, 2013.

[3] J.-C. Laperche, "Multi/many-core in avionics systems," in *4th Workshop TORRENTS*, Toulouse, France, 2013.

[4] C. Bérard, J. M. Biannic, and D. Saussié, *Commande multivariable - Application au pilotage d'un avion*. Dunod, 2012.

[5] C. Pagetti, J. Forget, F. Boniol, M. Cordovilla, and D. Lesens, "Multi-task implementation of multi-periodic synchronous programs," *Discrete Event Dynamic Systems*, vol. 21, no. 3, pp. 307–338, 2011.

[6] M. Cordovilla, F. Boniol, J. Forget, E. Noulard, and C. Pagetti, “Developing critical embedded systems on multicore architectures: the prelude-schedmcore toolset,” in *19th International Conference on Real-Time and Network Systems (RTNS’11)*, 2011, pp. 107–116.

[7] E. Betti, S. Bak, R. Pellizzoni, M. Caccamo, and L. Sha, “Real-time i/o management system with COTS peripherals,” *IEEE Trans. Computers*, vol. 62, no. 1, pp. 45–58, 2013.

[8] Tilera Corp., “Tile processor architecture - Overview for the TILEPro Series,” Tech. Rep. UG120, 2013.

[9] B. L. Stevens and F. L. Lewis, *Aircraft control and simulation*, 2nd ed. Hoboken, NJ : Wiley, 2003.

[10] W. J. Rugh and J. S. Shamma, “Research on gain-scheduling,” *Automatica*, vol. 36, no. 10, pp. 1401–1425, 2000.

[11] D. Saussié, L. Saydy, O. Akhrif, and C. Bérard, “Gain scheduling with guardian maps for longitudinal flight control,” *AIAA Journal of Guidance, Control and Dynamics*, vol. 34, no. 4, pp. 1045–1059, 2011.

[12] C. Gervais, J.-B. Chaudron, P. Siron, R. Leconte, and D. Saussié, “Real-time distributed aircraft simulation through HLA,” in *16th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (DS-RT 2012)*, 2012.

[13] K. Ogata, *Discrete-Time Control Systems*, 2^{nd} ed. Prentice Hall, 1994.

[14] 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*, vol. 91, no. 1, pp. 64–83, 2003.

[15] C. Sofronis, S. Tripakis, and P. Caspi, “A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling,” in *Proc. of the 6th International Conference on Embedded Software (EMSOFT’06)*, Seoul, South Korea, Oct. 2006, pp. 21–33.

[16] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, “The synchronous data-flow programming language LUSTRE,” *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[17] A. Curic, “Implementing Lustre programs on distributed platforms with real-time constraints,” Ph.D. dissertation, Université Joseph Fourier, Grenoble, 2005.

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

[19] Tilera Corp., “Architecture Overview TILE-Gx,” Tech. Rep. UG130, 2013.

[20] ——, “Tilera Documentation: Gx MDE Programming Overview,” Tech. Rep. UG 505, 2013.

[21] S. Brosky and S. Rotolo, “Shielded processors: guaranteeing sub-millisecond response in standard linux,” in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, 2003.

[22] W. Puffitsch, E. Noulard, and C. 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)*, 2013, pp. 293–302.

[23] R. Wilhelm and al., “The worst-case execution-time problem - overview of methods and survey of tools,” *ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 3, 2008.

[24] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, “Otawa: An open toolbox for adaptive wcet analysis,” in *8th IFIP WG 10.2 International Workshop Software Technologies for Embedded and Ubiquitous Systems (SEUS 2010)*, 2010, pp. 35–46.

[25] F. Boniol, H. Cassé, E. Noulard, and C. Pagetti, “Deterministic execution model on cots hardware,” in *25th International Conference Architecture of Computing Systems (ARCS’12)*, ser. Lecture Notes in Computer Science, vol. 7179. Springer, 2012, pp. 98–110.

[26] A. Cervin, “Integrated control and real-time scheduling,” Ph.D. dissertation, Dept. of Automatic Control, Lund University, Sweden, Apr. 2003.

[27] O. Sename, D. Simon, and M. E. M. Ben Gaïd, “A LPV approach to control and real-time scheduling codesign: application to a robot-arm control,” in *Proceedings of the 47th IEEE Conference on Decision and Control (CDC’08)*, Cancun Mexique, 2008, pp. 4891–4897.

[28] P. J. Andrianiaina, D. Simon, A. Seuret, J.-M. Crayssac, and J.-C. Laperche, “Weakening Real-time Constraints for Embedded Control Systems,” INRIA, Rapport de recherche RR-7831, 2011.

[29] S. Baruah, “Semantics-preserving implementation of multirate mixed-criticality synchronous programs,” in *20th International Conference on Real-Time and Network Systems (RTNS’12)*, 2012, pp. 11–19.

[30] M. Otter, B. Thiele, and H. Elmqvist, “A library for synchronous control systems in modelica,” in *9th International Modelica Conference*, 2012.

[31] S. P. Jean-Louis Camus, Olivier Graff, “A verifiable architecture for multitask, multi-rate synchronous software,” in *4th Embedded Real-Time Software Congress (ERTS’08)*, 2008.

[32] M. D. Natale and H. Zeng, “Task implementation of synchronous finite state machines,” in *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE 2012)*, 2012, pp. 206–211.

[33] N. Izerrouken, O. S. Y. Kai, M. Pantel, and X. Thirioux, “Use of formal methods for building qualified code generator for safer automotive systems,” in *1st Workshop on Critical Automotive Applications: Robustness & Safety (EDCC-CARS’10)*. ACM, 2010, pp. 53–56.

[34] T. Wang, A. Dieumegard, E. Feron, R. Jobredeaux, M. Pantel, and P.-L. Garoche, “Autocoding of Computer-controlled Systems with Control Semantics for Formal Verification (regular paper),” in *Safe and Secure Systems and Software Symposium (S5)*, 2012.

[35] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic, “Translating discrete-time Simulink to Lustre,” *ACM Trans. Embedded Comput. Syst.*, vol. 4, no. 4, pp. 779–818, 2005.

[36] N. Marian and Y. Ma, *Translation of Simulink Models to Component-based Software Models*. <Forlag uden navn>, 2007, pp. 274–280.

[37] R. Lublinerman and S. Tripakis, “Modular code generation from triggered and timed block diagrams,” in *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 02008)*. IEEE Computer Society, 2008, pp. 147–158.

[38] A. Bouakaz and J.-P. Talpin, “Buffer minimization in earliest-deadline first scheduling of dataflow graphs,” in *SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES’13)*, 2013, pp. 133–142.

[39] S. Bliudze, M. Jan, and X. Fornari, “From model-based to real-time execution of safety-critical applications: Coupling scade with oasis,” in *Embedded Real Time Software and Systems (ERTS’012)*, 2012.

[40] O. Kotaba, M. Paulitsch, S. Petters, H. Theiling, and J. Nowotsch, “Multicore in real-time systems – temporal isolation challenges due to shared resources,” in *Workshop on Industry-Driven Approaches for Cost-effective Certification of Safety-Critical, Mixed-Criticality Systems (WICERT’13)*, 2013.

[41] A. Abel, F. Benz, J. Doerfert, B. Dörr, S. Hahn, F. Haupenthal, M. Jacobs, A. H. Moin, J. Reineke, B. Schommer, and R. Wilhelm, “Impact of resource sharing on performance and performance prediction: A survey,” in *24th International Conference on Concurrency Theory (CONCUR’13)*, 2013, pp. 25–43.

[42] R. Wyss, F. Boniol, J. Forget, and C. Pagetti, “A synchronous language with partial delay specification for real-time systems programming,” in *10th Asian Symposium Programming Languages and Systems (APLAS 2012)*, 2012, pp. 223–238.

[43] Z. Al-bayati, H. Zeng, M. D. Natale, and Z. Gu, “Multitask implementation of synchronous reactive models with earliest deadline first scheduling,” in *8th IEEE International Symposium on Industrial Embedded Systems (SIES 2013)*, 2013, pp. 168–177.