Multisource Software on Multicore Automotive ECUs—Combining Runnable Sequencing With Task Scheduling

Aurélien Monot, Nicolas Navet, Bernard Bavoux, and Françoise Simonot-Lion

***Abstract*—As the demand for computing power is quickly increasing in the automotive domain, car manufacturers and tier-one suppliers are gradually introducing multicore electronic control units (ECUs) in their electronic architectures. In addition, these multicore ECUs offer new features such as higher levels of parallelism, which ease the compliance with safety requirements such as the International Organization for Standardization (ISO) 26262 and the implementation of other automotive use cases. These new features involve greater complexity in the design, development, and verification of the software applications. Hence, car manufacturers and suppliers will require new tools and methodologies for deployment and validation. In this paper, we address the problem of sequencing numerous elementary software modules, called runnables, on a limited set of identical cores. We show how this problem can be addressed as the following two subproblems, which cannot optimally be solved due to their algorithmic complexity: 1) partitioning the set of runnables and 2) building the sequencing of the runnables on each core. We then present low-complexity heuristics to partition and build sequencer tasks that execute the runnable set on each core. Finally, we globally address the scheduling problem, at the ECU level, by discussing how we can extend this approach in cases where other OS tasks are scheduled on the same cores as the sequencer tasks.**

***Index Terms*—Automotive, autosar, load balancing, multicore, runnable, scheduling, static cyclic scheduling.**

I. Introduction

MULTISOURCE software running on the same electronic control unit (ECU) is becoming increasingly widespread in the automotive industry. This case is one of the main reasons that car manufacturers want to reduce the number of ECUs, which grew up above 70 for high-end cars. One major outcome of the Automotive Open System Architecture (AUTOSAR) initiative and, more specifically, its operating system (OS) is to help car manufacturers shift from the "one function per ECU" paradigm to more centralized architecture designs by providing appropriate protection mechanisms.

Another crucial evolution in the automotive industry is that chip manufacturers are reaching the point where they can no longer cost effectively meet the increasing performance requirements through frequency scaling alone. This condition is one reason that multicore ECUs are gradually introduced in the automotive domain. The higher level of performance provided by multicore architectures may help simplify in-vehicle architectures by executing on multiple cores that the software previously run on multiple ECUs. This possible evolution toward more centralized architectures is also an opportunity for car manufacturers to decrease the number of network connections and buses. As a result, parts of the complexity will be transferred from the electrical/electronic architecture to the hardware and software architecture of the ECUs. However, static cyclic scheduling makes it easy to add functions to an existing ECU. In practice, important architectural shifts are hindered by the carryover of ECUs and existing subnetworks, which are widely used by generalist car manufacturers. The extent to which more centralized architectures will be adopted thus remains unsure.

Multicore ECUs are also helpful for other use cases. For example, they bring major improvements for some applications that require high performance such as high-end engine controllers, electric and hybrid powertrains [1], [2], and advanced driver assistance systems [3], which sometimes involve real-time image processing [4]. These multicore platforms also offer additional benefits such as higher level of parallelism, allowing for more segregation, which may help meet the requirements of the International Organization for Standardization (ISO) 26262, which concerns functional safety for road vehicles. Furthermore, in multicore architectures, some core can be dedicated to a specialized usage such as handling low-level services. Now, the challenge is to adapt existing design methods to the new multicore constraints. The scheduling of the software components is one of the key issues in that regard, and it has to be revamped.

The introduction of multisource and multicore will induce drastic changes in the software architecture of automotive ECUs. Section II introduces the most likely scheduling choices and the literature relevant to the task scheduling in

Manuscript received December 31, 2010; revised July 2, 2011 and September 26, 2011; accepted November 28, 2011. Date of publication January 27, 2012; date of current version April 27, 2012.

A. Monot was with the Lorraine Research Laboratory in Computer Science and Its Applications/National Polytechnic Institute of Lorraine (LORIA/INPL), 54506 Vandoeuvre, France. He is now with the National Institute for Research in Computer Science and Control, 78153 Le Chesnay, France (e-mail: aurelien.monot@inria.fr).

N. Navet is with the Lorraine Research Laboratory in Computer Science and Its Applications (LORIA), National Institute for Research in Computer Science and Control (INRIA), Lorraine, 54506 Vandoeuvre, France, and also with RealTime-at-Work (RTaW), 54506 Vandoeuvre, France (e-mail: nicolas.navet@inria.fr).

B. Bavoux is with PSA Peugeot Citroën, 78 943 Vélizy-Villacoublay, France (e-mail: bernard.bavoux@mpsa.com).

F. Simonot-Lion is with the Lorraine Research Laboratory in Computer Science and Its Applications/National Polytechnic Institute of Lorraine (LORIA/INPL), 54506 Vandoeuvre, France (e-mail: simonot@loria.fr).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIE.2012.2185913

multiprocessor automotive ECUs. Then, Section III presents solutions for the scheduling of numerous software modules when only a few OS tasks are allowed. This paper builds on the study published in [5], where it was assumed that only one sequencer task was running on each core of the ECU to schedule the runnables. In Section V, we consider how we can build several sequencer tasks while possibly scheduling other tasks on the same core, and we discuss how we can globally analyze the schedulability of such systems.

For clarity, "sequencing" refers to the scheduling of runnables, whereas "scheduling" is solely used for tasks.

II. Scheduling in the Automotive Domain

A. Scheduling Design Choices for Multicore ECUs

In this section, we explain and justify, particularly in light of predictability requirements, the multicore scheduling approach, which is, to the best of our knowledge, the most widely considered method in the automotive industry.

1) Partitioning Scheduling Scheme: In a multicore system, either the tasks are statically allocated to the cores or they can dynamically be distributed at runtime to balance the workload or migrate functions to increase availability. The latter approach involves complex tasks and resource interactions that are difficult to predict and validate. Thus, approaches that rely on static allocation (i.e., partitioning) and deterministic mechanisms such as periodic cyclic scheduling are more likely to be used in the automotive context, and this is the option taken within the AUTOSAR consortium. Scheduling tasks on a multiprocessor systems under the static partitioning approach has been well studied; for example, see [6]–[9]. However, the works we are aware of deal with online algorithms such as fixed priority preemptive (FPP) or earliest deadline first (EDF) and do not consider the static cyclic scheduling of tasks.

2) Static Cyclic Scheduling: The static cyclic scheduling of elementary software modules or runnables is common, because there are usually many more runnables than the maximum number of tasks allowed by automotive operating systems such as OSEK/VDX or AUTOSAR OS. Thus, runnables must be grouped together and scheduled within a sequencer task (also called a dispatcher task). In this paper, we focus on how we can sequence large runnable sets on multicore platforms using a static partitioning approach. Indeed, the static task partitioning scheme is very likely to be adopted, at least, in a first step, because it is conceptually simple and provides better predictability for ECU designers compared with a global scheduling approach. We aim at developing practical algorithms whose performances can be guaranteed to build the dispatcher tasks on each core and to schedule the runnables within these dispatcher tasks to comply with sampling constraints and, as long as possible, uniformize the CPU load over time. This latter objective is, of course, important to minimize the hardware cost and to facilitate the addition of new functions, as typically done in the incremental design process of car manufacturers. This objective is achieved by desynchronizing the runnable release dates. Precisely, the first release date of each runnable, called its offset, is determined to uniformly spread the CPU demand over time. The configuration algorithms developed in this paper

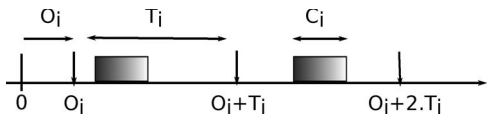

Fig. 1. Model of the runnables. After its release, an instance of a runnable has to be executed before the next instance is released (i.e., the deadline is set to the period).

are closely related to [10] (monoprocessor scheduling of tasks with offsets) and [11] (scheduling of frames with offsets), but it is applied to multicore and goes beyond as we provide lower bounds on the performances. Because the problem is of practical interest in the industry, there are in-house tools at the car manufacturers and commercial tools that have been developed for configuring the scheduling, such as RealTime-at-Work RTaW-ECU [12]. However, the proprietary algorithms used in these tools can usually not be disclosed, and they are sometimes specialized for some specific usage.

B. Model Description

In this paper, we consider a large set of n periodic elementary software modules, also called runnables, that will be allocated on an ECU that consists of m identical cores. In practice, a runnable can be implemented as a function that is called, whenever appropriate, within the body of an OS task.

1) Runnable Characteristics: The ith runnable is denoted by $\mathcal{R}_i = (C_i, T_i, O_i, \{R\}, P_i)$. Quantities C_i, T_i, and O_i correspond, respectively, to the worst case execution time (WCET), the period (i.e., the exact time between two successive releases), and the offset of $\mathcal{R}_i$. As shown in Fig. 1, the offset of a runnable is the release date of the first instance of that runnable, and subsequent instances are then periodically released. The choice made for the offset values has a direct influence on the repartition of the workload over time.

A set of interrunnable dependencies is denoted by $\{R\}$. Indeed, due to specific design requirements, such as shared variables, some runnables may have to be allocated on the same core, and the set $\{R\}$ is used to capture these constraints. In addition, some specific features, such as input/output (I/O) ports located on a given core, may require a runnable to be allocated onto a specific core. This locality constraint is expressed by P_i.

2) Dispatcher Task: Runnables are scheduled on their designated core using a dispatcher task or a "sequencer task," which stores the runnable activation times in a table and releases them at the right points in time. A dispatcher task is characterized by the duration of the dispatch table T_{cycle}, which is executed in a cyclic manner, and by a quantum T_{tic}, which is the duration of a slot in the table. Typically, we may have, for example, $T_{cycle} = 1000$ ms and $T_{tic} = 5$ ms. Note that T_{cycle} must be a multiple of the greatest common divisor of the runnable periods and the least common multiple (LCM) of these periods must be a multiple of T_{tic}. As a result, a dispatch table holds T_{cycle}/T_{tic} slots.

3) Assumptions: In this paper, we place a set of working assumptions, which, in our experience, can most often be met in current automotive applications, as follows.

- Each runnable is periodically executed strictly. As a result, the whole trajectory of the system is defined by the first activation times of the runnables (i.e., their offsets).

- The runnables are assumed to be offset-free, because the initial offset of a runnable can freely be chosen in the limit of its period (see [10]). These offsets will be assigned during the construction of the dispatch table, with the objective of uniformizing the central processing unit (CPU) load over a scheduling cycle.
- The WCETs of the runnables are assumed to be small compared to T_{tic}. Typical values for the case that we consider would be 5 ms for T_{tic} and $C_i \leq 300\ \mu$s.
- All cores are identical with regard to their processing speed.
- There are no dependencies between runnables allocated on different cores. Therefore, all cores can independently be scheduled. This assumption is in line with the choices made by AUTOSAR with regard to multicore architecture [13].

This last assumption allows us to divide the overall problem into two independent subproblems. The first part of the problem consists of allocating all of the n runnables onto the m cores with respect to their constraints, with the aim of balancing the CPU load of the m resulting partitions (see Section III-A). The second part of the problem consists of building the dispatch table for each core (see Section III-B).

4) Schedulability Condition: Assuming that we only consider runnable scheduling, the system is schedulable and, thus, can safely be deployed if and only if the following conditions are satisfied on each core.

1) The runnables are strictly periodically executed.
2) The initial offset of each runnable is smaller than its period.
3) The sum of the WCET of the runnables allocated in each slot does not exceed a given threshold, which is typically chosen as the duration of the slot, i.e., T_{tic}.

III. Runnable Sequencing Algorithms for Multicore ECUs

In this section, we present algorithms and, when possible, derive lower bounds on their efficiency to schedule large numbers of runnables on multicore ECUs.

Because automotive OSs can only handle a limited amount of OS tasks, the sequencing of runnables has to be done within dispatcher tasks. The first step of the approach is to partition the runnable sets onto different cores. The next and last step is to determine the offsets between the runnables allocated on each core to balance the load over time.

A. Building Tasks as a Bin-Packing Problem

It is assumed that the number of cores is fixed. We first distribute all the runnables on the cores. Assigning n tasks to m cores is similar to subdividing a set of n elements into m nonempty subsets. By definition, the number of possibilities for this problem is given by the Stirling number of the second kind (see [14]): $(1/m!)\sum_{i=0}^{m}(-1)^{(m-i)}\binom{m}{i}i^n$. Considering that the runnables may have core allocation constraints, the cores should be distinguished. Thus, the $m!$ combinations of cores must be considered. As a result, we have at most $\sum_{i=0}^{m}(-1)^{(m-i)}\binom{m}{i}i^n$ different possibilities for the partitioning problem alone. Such a complexity prevents us from an exhaustive search. For example, with $n = 30$ and $m = 2$, the search space holds more than one billion possibilities.

Considering this complexity, to balance as evenly as possible the utilization of processor cores, we propose a heuristic based on the bin-packing decreasing worst-fit scheme for a fixed number of bins (where "bins" are processor cores). The heuristic is given in Algorithm 1.

Algorithm 1: Partitioning of the runnable set.

Input: Runnable set $\{\mathcal{R}_i\}$, number of cores m
(1) Group interdependent runnables into runnable clusters. Independent runnables become clusters of size 1.
(2) Allocate the runnable clusters that have a locality constraint to the corresponding cores.
(3) Sort the runnables clusters by decreasing order of CPU utilization rate $\rho = \sum_i (C_i/T_i)$.
(4) Iterate over the sorted clusters.
  (a) Find the least loaded (LL) core.
  (b) Assign the current cluster to this core.

Step 1 runs in $\mathcal{O}(n)$. Step 2 runs in $\mathcal{O}(n)$, but all the runnables allocated in step 2 will not have to go through steps 3 and 4, which are algorithmically more complex. Step 3 runs in $\mathcal{O}(n \cdot \log n)$. Finally, step 4 runs in $\mathcal{O}(n \cdot m)$. As a conclusion, Algorithm 1 runs in $\mathcal{O}(n(m + \log n))$, which does not raise any issue in practical cases.

Note that $m \geq \lceil \sum_{i=1}^{m}(C_i/T_i) \rceil$ is a necessary schedulability condition that can be used to rule out configurations with too few processor cores.

B. Strategies for Sequencing Runnables

The next stage consists of building the dispatch table for the set of runnables. In the first step, it is assumed that there are no precedence constraints between the runnables and that a single sequencer table is needed per core. This latter assumption can easily be relaxed, as done in Section V.

1) Least Loaded Algorithm: Considering a runnable R_i of period T_i, there are (T_i/T_{tic}) possibilities for allocating this runnable (see schedulability condition 2 in Section II-B4). As a result there are $\prod_{i=1}^{n}(T_i/T_{tic})$ alternative schedules for the n runnables, and given the cost function, we are not aware of any ways of finding the optimal solution with an algorithm that does not have an exponential complexity. Considering a realistic case of 50 runnables whose period is as least twice as large as T_{tic}, we would need to evaluate a minimum of 2^{50} possible solutions. Once again, given the complexity, we have to resort to a heuristic. Here, we adapt to the problem of sequencing runnables the LL algorithm proposed by Grenier *et al.* in [11] for the frame offset allocation on a controller area network.

The intuition behind the heuristic is simple. At each step, we assign the next runnable to the LL slot, as described in Algorithm 2. The load of a slot is the sum of the C_i of the runnables $\{\mathcal{R}_i\}$ assigned to this slot. This algorithm is further referred to as the LL algorithm.

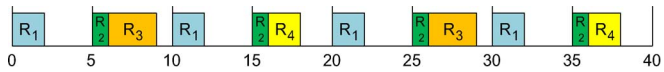

Fig. 2. Example of dispatch table.

TABLE I
LOAD REPARTITION CORRESPONDING TO THE DISPATCH TABLE IN FIG. 2

Slot	1	2	3	4	5	6	7	8
Load	2	4	2	3	2	4	2	3

Algorithm 2: Assigning runnables to slots—LL heuristic.

Input: Runnable set $\{\mathcal{R}_i\}$, T_{tic}, T_{cycle}
(1) Sort runnables $\mathcal{R}_i$ such that $T_{tic} \leq T_1 \leq \cdots \leq T_n \leq T_{cycle}$.
(2) For $i = 1 \ldots n$
  (a) Look for the LL slot in the (T_i/T_{tic}) first slots.
  (b) Allocate $\mathcal{R}_i$ in every (T_i/T_{tic}) slot, starting from this slot.

Step 1 runs in $\mathcal{O}(n \cdot \log n)$. Step 2 iterates n times over steps 2a and 2b, which run, respectively, in $(T_i/T_{tic}) \leq (T_{cycle}/T_{tic})$ and $(T_{cycle}/T_i) \leq (T_{cycle}/T_{tic})$. As a result, this algorithm runs in $\mathcal{O}(n(\log n + (\max_i\{T_i\}/T_{tic}) + (T_{cycle}/\min_i\{T_i\})) \leq \mathcal{O}(n(\log n + 2(T_{cycle}/T_{tic}))$.

For practical applications, ties at step 1 are broken using the highest WCET first, and ties at step 2a are broken by choosing the central slot of the longest sequence of consecutive slots having the minimum load. Although the latter rule for breaking ties does not have any impact on the theoretical results, which will be derived next, it helps separate load peaks, which is important from the ECU designer point of view. As an illustration, applying the LL heuristic to the set of runnables $\mathcal{R}_i(T_i, C_i) : \mathcal{R}_1(10, 2)$, $\mathcal{R}_2(10, 1)$, $\mathcal{R}_3(20, 4)$, $\mathcal{R}_4(20, 2)$ leads to the dispatch table shown in Fig. 2. The resulting distribution of the load is shown in Table I.

We consider two main metrics for evaluating the quality of a dispatch table. The first important criterion is to have the lowest maximum load in the cycle, because this will determine the feasibility of the schedule and the possibility of adding further functions later in the lifetime of the system. The maximum load over all slots is also referred to as the *peak load*. In the second step, a more fine-grained assessment of the uniformity of the load balancing can be given by the standard deviation of the load distribution over all the slots.

2) Upper Bound on the Peak Load: Here, we derive an upper bound on the peak load, which holds for runnable sets with harmonic periods (i.e., each period is a multiple of all smaller periods). Based on this value, we consequently derive a closed-form sufficient schedulability condition. In this perspective, we first point out that the slots in which a runnable $\mathcal{R}_i$ will periodically be assigned are of equal load, which is the rationale behind step 2a in Algorithm 2.

Lemma 1: Before the allocation of a runnable $\mathcal{R}_i$, the slot allocation induced by the previously allocated runnables repeats with a period (T_i/T_{tic}).

Proof: This lemma is proved by induction. The property holds for $\mathcal{R}_0$, because all slots are empty. Assuming that the property holds for $\mathcal{R}_i$, this runnable will periodically be allocated in every (T_i/T_{tic}) slots. Therefore, the slot allocation will still repeat with a period (T_i/T_{tic}) after its allocation in the LL slot. Because runnables are sorted by increasing periods and their periods are harmonic, $T_{i+1} = k \cdot T_i$ with $k \in \mathbb{N}^*$, and the slot allocation also repeats with a period $k \cdot (T_i/T_{tic}) = (T_{i+1}/T_{tic})$ before the allocation of $\mathcal{R}_{i+1}$. ■

Therefore, the LL slot in the first (T_i/T_{tic}) slots is the LL over the whole dispatch table, and we need not need look farther. As the second step, we show that the highest load in the slot where a runnable will be allocated arises when the load is equal in every slot.

Lemma 2: The maximum load in the LL slot is obtained for perfect load balancing, which corresponds to a constant load throughout the cycle.

Proof: Reasoning with a constant allocated load, anything else than a perfect load balancing will result in a load below the average load per slot in some slot that will eventually be chosen to allocate the runnable under consideration. ■

As a result, the highest peak that a runnable can create happens in the case of perfect load balancing. Now, let us define $\rho_k = \sum_{i \in \{\mathcal{R}\}_k} (C_i/T_i)$, the total utilization of core k, where $\{\mathcal{R}\}_k$ is the set of runnables allocated to core k, and card$\{R\}_k$ is the cardinality of $\{R\}_k$.

Theorem 3: On processor k, an upper bound on the peak load of a slot allocation is

$$PL_k = \max_{i \in \{\mathcal{R}\}_k} \left\{ C_i + \rho_k T_{tic} - \sum_{j=i}^{\text{card}\{\mathcal{R}\}_k} \frac{C_j}{T_j} T_{tic} \right\}. \quad (1)$$

Proof: In the case of perfect load balancing, before the allocation of $\mathcal{R}_i$, the load of a slot is given by $\sum_{\text{allocated runnables}} WCET \cdot$ (number of allocation slots/total number of slots), i.e.,

$$\sum_{j \in \{\mathcal{R}\}_k}^{i-1} C_j \cdot \frac{T_{cycle}}{T_j} \cdot \frac{T_{tic}}{T_{cycle}}. \quad (2)$$

After the allocation of $\mathcal{R}_i$, the load in the corresponding slot is

$$C_i + \sum_{j \in \{\mathcal{R}\}_k}^{i-1} C_j \cdot \frac{T_{tic}}{T_j}. \quad (3)$$

Moreover, $\sum_{j \in \{\mathcal{R}\}_k}^{i-1} (C_j/T_j) = \rho_k - \sum_{j=i}^{\text{card}\{\mathcal{R}\}_k} (C_j/T_j)$.

Consequently, the worst case peak load on processor core k resulting from the allocation of $\mathcal{R}_i$ in a slot is

$$PL_k^i = C_i + \rho_k T_{tic} - \sum_{j=i}^{\text{card}\{\mathcal{R}\}_k} \frac{C_j}{T_j} T_{tic}. \quad (4)$$

Taking the maximum for all the runnables gives (1). ■

If the worst case peak load is below T_{tic} for all runnables, then the solution given by the algorithm is schedulable. Hence, we have the following corollary.

Corollary 1: Based on Theorem 1, we derive the following sufficient schedulability condition:

$$\rho_k \leq 1 + \frac{C_{\min}}{T_{\max}} - \frac{C_{\max}}{T_{tic}} \quad (5)$$

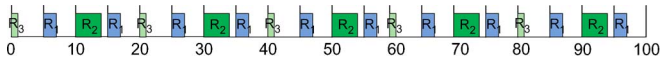

Fig. 3. Dispatch table before the insertion of $\mathcal{R}_4$.

with $C_{\max} = \max_{i\in\{\mathcal{R}\}_k}\{C_i\}$, $T_{\max} = \max_{i\in\{\mathcal{R}\}_k}\{T_i\}$ and $T_{\min} = \min_{i\in\{\mathcal{R}\}_k}\{T_i\}$.

Proof: $\forall i,\ C_i \le C_{\max}$ and $\forall i,\ \sum_{j=i}^{\mathrm{card}\{\mathcal{R}\}_k}(C_j/T_j) \ge (C_{\min}/T_{\max})$ gives

$$\forall i, PL_k^i \le C_{\max} + \rho_k T_{tic} - \frac{C_{\min}}{T_{\max}} T_{tic}. \tag{6}$$

Scheduling condition 3 in Section II-B4 (i.e., $PL_k^i \le T_{tic}$) leads to the result. ■

This bound is achievable for $n \cdot k$ identical runnables with a period equal to $k \cdot T_{tic}$ and load equal to C and the last runnable $\mathcal{R}_{n\cdot k+1}$ of period $T_{\max}$ and load C. With this setup, $C_{\min} = C_{\max} = C$ and the allocation of the $n \cdot k$ first runnable results in perfect load balancing of constant load $\rho_k \cdot T_{tic} - C \cdot T_{tic}/T_{\max}$. As a result, allocating the last runnables induces the load $\rho_k \cdot T_{tic} - C_{\min} \cdot T_{tic}/T_{\max} + C_{\max}$ in some slots.

3) Lower Bound on the Allocatable Load: We introduce here a lower bound on the capacity of the core that Algorithm 2 guarantees to use, given a harmonic runnable set. This lower bound is referred to as the harmonic schedulability bound.

Theorem 5: The harmonic schedulability bound is equal to $(1 - (C_{\max}/T_{tic}))$ of the capacity of each core.

Proof: Reasoning as done for Corollary 1, the worst case peak load is given by allocating a runnable $\mathcal{R} = (C_{\max}, T_{\max} = T_{tic})$ in a slot allocation with perfect balance load. In the worst case, the system is still schedulable when this average slot load is equal to $T_{tic} - C_{\max}$. In other words, when the system becomes no longer schedulable, every slot has an allocated load greater than or equal to $T_{tic} - C_{\max}$. As a consequence, at least $(1 - (C_{\max}/T_{tic}))$ of the capacity of the considered core can be used by our algorithm. ■

For example, with $T_{tic} = 5$ ms and $C_{\max} = 300\ \mu$s, at least 94% of the CPU is guaranteed to be usable. In practice, when $C_{\max}$ is small, this bound is very useful.

Considering the problem of scheduling a given harmonic runnable set on a multicore ECU with an infinite number of cores using as few cores as possible, the following corollary gives a bound of the maximum number required by this algorithm.

Corollary 2: Defining $\mathrm{P} = \sum_i (C_i/T_i)$, the total load to allocate for a runnable set with harmonic periods, and $m_{\min}$, the number of core required by the LL algorithm to schedule it, Theorem 2 gives

$$m_{\min} \le \left\lceil \frac{\mathrm{P}}{1 - C_{\max}/T_{tic}} \right\rceil. \tag{7}$$

4) Dealing With Nonharmonic Runnable Sets: Usually, in practice, runnable sets do not have strictly harmonic periods. As a consequence, Lemmas 1 and 2 no longer hold anymore, and (1) and (5) cannot be applied to provide bounds. In particular, placing a runnable in the LL slot of the dispatch table could induce peaks because of the runnable periodicity. Take the following runnable set for example: $\mathcal{R}_1(10, 2)$, $\mathcal{R}_2(20, 3)$, $\mathcal{R}_3(20, 1)$, $\mathcal{R}_4(50, 2)$ with $T_{tic} = 5$ and $T_{cycle} = 100$. Fig. 3 shows the dispatch table before the allocation of $\mathcal{R}_4$. The resulting distribution of the load is shown in Table II.

TABLE II
LOAD REPARTITION CORRESPONDING TO THE DISPATCH TABLE IN FIG. 3

Slot	1	2	3	4	5	6	7	8	9	10	12	12	...
Load	1	2	4	2	1	2	4	2	1	2	4	2	...

At that point, choosing one of the LL slots in the dispatch table will make the schedule fail, because $\mathcal{R}_4$ will also have to be allocated in the slot with the highest load because of its periodicity. For example, if the first instance of $\mathcal{R}_4$ is allocated in slot 1, the next instance will be placed in slot 11, and this condition will make the system unschedulable. However, allocating $\mathcal{R}_4$ in any even slot is safe.

To deal with nonharmonic runnable sets, we need to go through a larger window of slots for the choice of the offsets. In the following discussion, variable T_{window} is equal to the LCM of the periods of the runnables already scheduled at the current state of the algorithm. Instead of looking for the LL slot in the first T_i/T_{tic} slots, we try to create the lowest peak (LP) over T_{window}, knowing that the schedule repeats in cycle afterward. This algorithm is further referred to as LP.

Algorithm 3: LP heuristic.

Input: Runnable set $\{\mathcal{R}_i\}, T_{tic}, T_{cycle}$
(1) Sort runnables $\mathcal{R}_i$ such that $T_{tic} \le T_1 \le \cdots \le T_n \le T_{cycle}$.
(2) $T_{window} = T_{tic}$.
(3) For $i = 1 \ldots n$
  (a) $T_{window} = LCM(T_{window}, T_i)$.
  (b) In the first (T_i/T_{tic}) slots, look for the slot such that the highest load in the slots where $\mathcal{R}_i$ is periodically allocated in the (T_{window}/T_{tic}) first slots is the lowest.
  (c) Allocate $\mathcal{R}_i$ in every (T_i/T_{tic}) slot, starting from this slot.

Step 1 of Algorithm 3 runs in $\mathcal{O}(n \cdot \log n)$. Step 3a runs in $\mathcal{O}(\log T_{cycle})$. Steps 3b and 3c, respectively, run in $\mathcal{O}(n(T_{window}/T_{tic})) \le \mathcal{O}(n(T_{cycle}/T_{tic}))$ and $\mathcal{O}(n(T_{cycle}/T_i)) \le \mathcal{O}(n(T_{cycle}/T_{tic}))$. As a result, the whole algorithm runs in $\mathcal{O}(n(\log n + 2(T_{cycle}/T_i) + \log T_{cycle}))$.

5) Improvement: Placing Outliers First: The algorithms described in Sections III-A and B construct the sequencing of runnables with arbitrary periods and, possibly, with locality and interrunnable constraints. Experiments show that these algorithms sometimes do not always perform well with runnable sets where a few runnables with low frequency have a very large WCET compared to other runnables.

In practice, runnables with a large WCET tend to have a large period. As a result, runnables with a large WCET are usually processed late in the runnable allocation, which explains the load peaks. To reduce these peaks, the scheduling algorithm is improved by first processing some runnables with a large WCET.[1]

[1] Allocating the runnables by decreasing order of WCET proves not to be an efficient approach in our experiments.

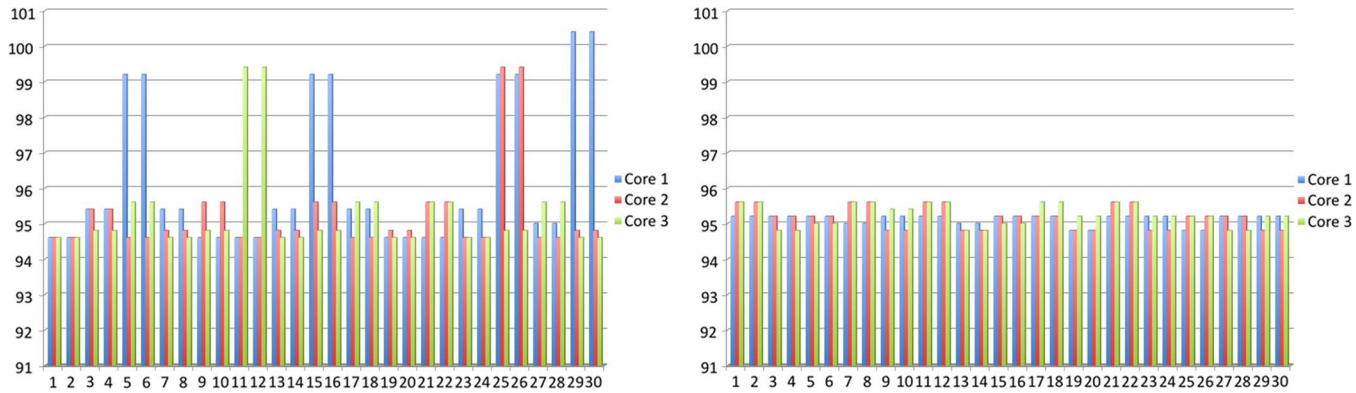

Fig. 4. Distribution of the load percentage over time. The left-hand graphic shows the result of the LP algorithm, whereas the right-hand graphic shows the result of the $LP_{k\sigma}$ algorithm. Only the first 30 slots are shown in the graphics, but they are representative of the whole 200 slots of the sequencer tasks. The algorithms in this paper have been implemented as plug-ins of the RealTime-at-Work RTaW-ECU [12] software.

We define the WCET threshold $C_{critic} = \mu + k \cdot \sigma$ with μ and σ denoting, respectively, the average and the standard deviation of the distribution of $\{C_i\}$ and k an integer value. The runnables with C_i larger than C_{critic} are allocated first. Then, the rest of the runnables are processed as done in Algorithm 3. This new version of the load-balancing algorithm is referred to as $LP_{k\sigma}$.

IV. Experimentations

Here, we evaluate the ability of the algorithms to uniformize the CPU load over time and to keep on providing feasible solutions at very high load levels. For this purpose, the algorithms LL, LP, and $LP_{k\sigma}$, described, respectively, in Sections III-B1, B4, and B5, have been implemented in the freely available software RTaW-ECU [12].

A. Balancing Performance

We applied the algorithms to sets of runnables that are realistic in the sense that their characteristics (i.e., period and WCET) are drawn at random from distributions derived from an existing body gateway ECU with about 200 runnables whose periods are close to harmonic (only about 5% of the runnables have nonharmonic periods).

In the experiments, the duration of the slot T_{tic} is set to 5 ms, the largest WCET is 30 times the smallest WCET, and the periods are nonharmonic, chosen in {10, 20, 25, 40, 50, 100, 200, 250, 500, 1000} ms. Random dependencies between runnables are also introduced through the following three parameters.

- Interdependency ratio, i.e., the percentage of runnables that are dependent and must thus be executed on the same core, is chosen equal to 30%.
- Maximum size of the clusters of dependent runnables is equal to 4.
- Core locality constraint ratio is the percentage of runnables that are preallocated to a given core, chosen to be equal to 30%.

The following additional parameters are used for this experimentation: 1) $C_{\max} = 300$ μs; 2) $T_{tic} = 5$ ms; 3) $T_{cycle} = 1$ s. In addition, there are more than 4000 runnables to schedule on three cores, inducing an average load of 95% of the capacity of the ECU. The parameters have been set so that the problem is challenging, because we are above the harmonic schedulability bound, which would be 94% here.

The distribution of the load obtained with the LP and $LP_{1\sigma}$ algorithms are shown in Fig. 4. In the two subgraphics, the x-axis is the time line, and the x-axis is the load of the slots (in percentage). The left-hand graphic shows that LP fails to provide a feasible schedule, because the load is slightly above 100% in some slots (e.g., slots 29 and 30 of core 1). The few load peaks (corresponding to around 5% of the core capacity) are due to runnables with a large WCET with a large period and, thus, have been placed late in the allocation. On the other hand, $LP_{1\sigma}$ can successfully schedule the runnable sets on the three cores with a well balanced distribution of load. In addition, around 5% of the capacity of each core remains available most of the time, which means that some more runnables can be added in future evolutions of the ECU.

B. Schedulability Performances and Robustness on Automotive ECUs

The goal is to assess the extent to which the schedulability bound, even if it has been derived in the harmonic case, can provide guidelines for the nonharmonic case. Precisely, we measure the success rate of the algorithms in the nonharmonic case at load levels such that feasibility would be ensured in the harmonic case. In the existing body gateway ECU, the set of task periods is close to be harmonic, because withdrawing only a few runnables ensures the harmonic property. To test the algorithms in a more difficult context, we build a "hard" nonharmonic case with more departure from the harmonic property. Precisely, the periods are now chosen in the set {10, 20, 25, 40, 50, 100, 125, 200, 125, 500, 1000} ms.

As shown in Table III, when the load is close to the harmonic schedulability bound, the algorithms remain efficient, in particular the LP, which successfully scheduled the 1000 random configurations of the test. This result suggests that the harmonic schedulability bound is also a good dimensioning criterion in the nonharmonic case.

Table IV presents the results obtained at higher loads, i.e., above the harmonic schedulability bound. Precisely, sets of runnables with the maximum WCET equal to 300 and 900 μs

TABLE III
PERFORMANCES OF THE SCHEDULING ALGORITHMS IN THE NONHARMONIC CASE WHEN THE LOAD IS CLOSE TO THE HARMONIC SCHEDULABILITY BOUND: STATISTICS COLLECTED ON 1000 RANDOM CONFIGURATIONS FOR EACH MAXIMUM WCET VALUE. THE SCHEDULABILITY BOUND IS DERIVED FROM THEOREM 2

max WCET (μs)	150	300	900
Schedulability bound in the harmonic case	97%	94%	82%
Success % of LL in the "hard" non-harmonic case	96%	96%	92%
Success % of LP in the "hard" non-harmonic case	100%	100%	100%

TABLE IV
PERFORMANCES OF THE SCHEDULING ALGORITHMS IN THE NONHARMONIC CASE WHEN THE LOAD IS GREATER THAN THE HARMONIC SCHEDULABILITY BOUND: STATISTICS COLLECTED ON 1000 RANDOM CONFIGURATIONS FOR EACH MAXIMUM WCET VALUE

CPU load	95%	97%	95%	97%
Schedulability bound in the harmonic case	94% WCET=300μs		82% WCET=900μs	
Success % of LL	64%	18%	12%	1%
Success % of LP	94%	94%	30%	5%
Success % of $LP_{1\sigma}$	100%	100%	97%	76%

and CPU loads equal to 95% and 97% are scheduled with LL, LP, and $LP_{1\sigma}$.

As expected, the lower the schedulability bound is, the harder it becomes to schedule the runnables (compare, for example, the 97% columns). The second lesson is that $LP_{1\sigma}$ clearly outperforms all the other contenders, particularly when the WCETs are large.

V. COMBINING RUNNABLE SEQUENCING WITH TASK SCHEDULING

In this section, we address the global scheduling at the OS level in an approach that combines sequencer tasks with other OS tasks. We assume that the different tasks are scheduled by a fixed-priority preemptive scheduler, as it is the case in AUTOSAR ECUs. In the next sections, the following two cases are distinguished: 1) synchronized tasks and 2) nonsynchronized tasks. In the first case, the initial offsets between tasks are known, and the tasks are scheduled using a single clock. In the second case, the different sequencer tasks may be driven by different clocks. This latter case arises, for example, in engine controllers in which some runnables are driven by the microcontroller clock whereas other runnables are driven based on the engine speed (in revolutions per minute), which varies over time. For each case, we discuss how we can build the slot allocation of the sequencer tasks to maximize the schedulability of the task set and then, we address how we can verify the schedulability of the resulting solution.

A. Problem Description

In this section, the focus is set at the core level, and the assumption that only one sequencer task is scheduled on each core is relaxed. Now, it is assumed that we can have several sequencer tasks on the same core. This case arises when memory protection across runnables is needed. Indeed, memory protection, such as provided by AUTOSAR OS, cannot be ensured at the runnable level but at the task (or interrupt service routine and OS application) level. Then, we are now given an extra set of periodic tasks that needs to be scheduled on the same core and described, as usual, by $\mathcal{T}_i = (C_i, T_i, D_i, J_i)$. Quantities C_i, T_i, D_i, and J_i correspond to the WCET, the period, the relative deadline, and the release jitter of the task $\mathcal{T}_i$, respectively.

In the context of this problem, the runnables have been allocated onto a set of sequencer tasks $\mathcal{S}_j$, as described in Section II-B2, according to their source and memory protection requirements, but the slot allocation remains to be done. Schedulability condition 3 in Section II-B4 is also relaxed: it is too stringent for the case where we have multiple sequencer tasks per core, because higher priority tasks may preempt the sequencer tasks during the whole slot duration. For now, we only require runnables to have a single instance active at each point in time, which corresponds to the classical case where deadlines are equal to periods. Finally, although it can be handled in our framework, it is assumed that the release jitter of a sequencer task is negligible, because its activations are usually driven by a high-priority OS service. Given these hypotheses, the problem is to find a strategy for building the slot allocation of all the sequencer tasks to increase the schedulability of the system consisting of a task set and some runnable sets that are scheduled on the same core.

B. Synchronized Tasks

Here, synchronized means that the initial offsets of the different tasks are known and all the tasks are scheduled using the same time basis. As a consequence, the tasks' phasings are known and can be used when building the slot allocation of the sequencer tasks. As in the case where we have a single sequencer, this latter problem cannot optimally be solved (see Section III-B1), and a heuristic approach is required.

We propose a similar approach as done for the LP algorithm, but here, we have to take into account the interferences of higher priority tasks. When placing a runnable, we look for the slot that minimizes the highest response time of the runnable, throughout all the slots where it is allocated (until the schedule repeats itself). For that purpose, each slot is transformed in a task that captures the execution requirements of all the runnables allocated in the slot.

The response time needs to be calculated for each slot in the hyperperiod of the task set (possibly longer than T_{cycle} of the sequencer task). This method can be done by using the response time analysis for static priority tasks with offsets and jitters, as introduced in [15], and applying it for each slot to whom the runnable belongs. Integrating sequencer tasks into Redell's approach is done by transforming the slot allocations into a task set. Each of the slots is translated into a single task with the proper offset and a period equal to T_{cycle}. This step does not further complexify Redell's analysis, because the number of task instances remains the same.

Although this approach is extremely computation intensive,[2] usual automotive applications lead to small hyperperiods,

[2]The problem of computing response times with offsets in the synchronized case is conjectured to be NP-hard; however, to the best of our knowledge, there is no proof in the literature.

because the task sets have almost harmonic periods. Furthermore, the schedulability analysis is conducted at the same time as the sequencer tasks are built.

C. Nonsynchronized Tasks

The previous approach cannot be applied to nonsynchronized tasks, because their offset and time bases are not known. If the tasks are scheduled on different and varying time bases, e.g., CPU clock and engine speed (in revolutions per minute), regardless of how a sequencer task is constructed, every higher priority task can interfere with any of the sequencer task's slot, because all offset configurations between them are possible at runtime. This condition means that, contrary to the synchronized case in Section V-B, we cannot take advantage of the characteristics of the higher priority tasks when building the slot allocation of a sequencer task. The maximum robustness to all possible asynchronisms between sequencer tasks is achieved by individually balancing the load of each of the tasks, as done in the basic use case in the algorithms in Section III-B.

Because of the possibly varying time bases, the schedulability of the slot allocation cannot be checked as in the previous cases. However, if it is possible to bound the clock speed of each sequencer task, and the multiframe task model (i.e., periodic activations but varying execution times between instances [16]) can be used to check the feasibility of the schedule. The transformation of a sequencer task into a multiframe task is lossless: the slots of a sequencer task become the task instances, with their execution times depending on the runnables actually scheduled in each slot. Then, assuming the maximum clock speeds, which lead to the worst case workload arrival, a multiframe schedulability test that integrates release jitters in the schedulability analysis can be applied; for example, see [17], [18], or [19].

VI. Conclusion

Multisource software and multicore ECUs will drastically change the electrical/electronic architectures and should enable more cost effective and more flexible automotive embedded systems. In our view, the OS protection mechanisms specified by AUTOSAR provide a sound basis for developing appropriate safety mechanisms and policies, despite the growing complexity and criticality of software functions. However, current design methodologies need to be adapted to this new context, and there is a wide range of technical problems to be solved. Among these issues are the design of the software architectures and the scheduling of the software components, which have been considered in this paper.

The set of runnable sequencing algorithms proposed in this paper aims at uniformizing the load over time and thus increases the maximum workload schedulable on the CPU. The algorithms also provide guaranteed performance levels in some specific contexts. Experimentations on realistic case studies have confirmed that the algorithms are versatile and efficient in terms of CPU usage optimization.

We have presented practical solutions for scheduling activities according to both the static cyclic and priority-driven paradigms, as it is becoming a need in automotive multicore ECUs and other complex embedded systems with dependability requirements such as in the aerospace domain. Our ongoing work aims at extending this study to handle the constraints originating in the communication between runnables located on distinct ECUs. This approach first requires the precise modeling of the data exchanges and capturing their timing constraints, for example, using the TIMMO-2-USE methodology, and then extending the scheduling algorithms.

References

[1] A. Emadi, Y. Lee, and K. Rajashekara, "Power electronics and motor drives in electric, hybrid electric, and plug-in hybrid electric vehicles," *IEEE Trans. Ind. Electron.*, vol. 55, no. 6, pp. 2237–2245, Jun. 2008.

[2] F. Mapelli, D. Tarsitano, and M. Mauri, "Plug-in hybrid electric vehicle: Modeling, prototype realization, and inverter losses reduction analysis," *IEEE Trans. Ind. Electron.*, vol. 57, no. 2, pp. 598–607, Feb. 2010.

[3] D.-J. Kim, K.-H. Park, and Z. Bien, "Hierarchical longitudinal controller for rear-end collision avoidance," *IEEE Trans. Ind. Electron.*, vol. 54, no. 2, pp. 805–817, Apr. 2007.

[4] T. Bucher, C. Curio, J. Edelbrunner, C. Igel, D. Kastrup, I. Leefken, G. Lorenz, A. Steinhage, and W. von Seelen, "Image processing and behavior planning for intelligent vehicles," *IEEE Trans. Ind. Electron.*, vol. 50, no. 1, pp. 62–75, Feb. 2003.

[5] N. Navet, A. Monot, B. Bavoux, and F. Simonot-Lion, "Multisource and multicore automotive ECUs—OS protection mechanisms and scheduling," in *Proc. IEEE ISIE*, 2010, pp. 3734–3741.

[6] A. Burchard, J. Liebeherr, Y. Oh, and S. H. Son, "New strategies for assigning real-time tasks to multiprocessor systems," *IEEE Trans. Comput.*, vol. 44, no. 12, pp. 1429–1442, Dec. 1995.

[7] Y. Oh and S. Son, "Fixed-priority scheduling of periodic tasks on multiprocessor systems," Dept. Comput. Sci., Univ. Virginia, Charlottesville, VA, Tech. Rep. CS-95-16, 1995.

[8] S. Lauzac, R. Melhem, and D. Mossé, "An improved rate-monotonic admission control and its applications," *IEEE Trans. Comput.*, vol. 52, no. 3, pp. 337–350, Mar. 2003.

[9] A. Karrenbauer and T. Rothvoss, "An average-case analysis for rate-monotonic multiprocessor real-time scheduling," in *Proc. 17th Annu. ESA*, 2009, pp. 432–443.

[10] J. Goossens, "Scheduling of offset free systems," *Real-Time Syst.*, vol. 24, no. 2, pp. 239–258, Mar. 2003.

[11] M. Grenier, L. Havet, and N. Navet, "Pushing the limits of CAN—Scheduling frames with offsets provides a major performance boost," in *Proc. Eur. Congr. ERTS*, 2008.

[12] RealTime-at-Work, *RTaW-ECU: Static cyclic scheduling of tasks*, 2011. [Online]. Available: http://www.realtimeatwork.com

[13] AUTOSAR Consortium, AUTOSAR Release 4.0, *Specification of multicore OS architecture v1.0*, 2009.

[14] M. Abramowitz and I. Stegun, *Handbook of Mathematical Functions*. New York: Dover, 1970.

[15] O. Redell and M. Törngren, "Calculating exact worst case response times for static priority scheduled tasks with offsets and jitter," in *Proc. RTAS*, 2002, pp. 164–172.

[16] A. Mok and D. Chen, "A multiframe model for real-time tasks," *IEEE Trans. Softw. Eng.*, vol. 23, no. 10, pp. 635–645, Oct. 1997.

[17] S. Baruah, D. Chen, and A. Mok, "Static-priority scheduling of multiframe tasks," in *Proc. 11th Euromicro Conf. Real-Time Syst.*, 1999, pp. 38–45.

[18] A. Zuhily and A. Burns, "Exact response time scheduling analysis of accumulatively monotonic multiframe real time tasks," in *Proc. ICTAC*, 2008, pp. 410–424.

[19] A. Zuhily and A. Burns, "Exact scheduling analysis of nonaccumulatively monotonic multiframe tasks," *Real-Time Syst.*, vol. 43, no. 2, pp. 119–146, Oct. 2009.

[20] ITEA2 TIMMO-2-USE Consortium, *Mastering timing tools, algorithms, and languages*, 2011. [Online]. Available: http://www.timmo-2-use.org/overview.htm

Aurélien Monot received the B.Eng. degree in computer science from the Ecole des Mines de Nancy, Nancy, France, in 2008 and the Ph.D. degree on "End-to-End Timing Constraints in the AutoSar Context" from the PSA Peugeot Citroën, Nancy, cosupervised by Lorraine Research Laboratory in Computer Science and Its Applications/National Polytechnic Institute of Lorraine (LORIA/INPL), Vandoeuvre, France.

For six months, he was with the IBM Research Laboratory, Zürich, Switzerland, He is currently a Research Engineer with the National Institute for Research in Computer Science and Control (INRIA), Le Chesnay, France, working on the ITEA2 Timmo-2-Use Project. His research interest include real-time embedded systems and model-based design approaches.

Nicolas Navet received the B.S. degree in computer science from the University of Berlin, Berlin, Germany, in 1993 and the Ph.D. degree in computer science from the National Polytechnic Institute of Lorraine (INPL), Nancy, France, in 1999.

Since 2000, he has been a Researcher with Lorraine Research Laboratory in Computer Science and Its Applications (LORIA), National Institute for Research in Computer Science and Control (INRIA), Lorraine, France, where he is also the Head of the INRIA Real-Time and Interoperability (TRIO) Project. He is the Founder of RealTime-at-Work, a company that helps system designers build truly safe and optimized critical systems. His research interests include real-time and embedded systems, communication protocols, fault tolerance, and dependability assessment. For the last 17 years, he has worked on numerous projects with OEMs and suppliers in the automotive and avionics domains.

Bernard Bavoux received the B.Eng. degree from the École Supérieure d'Électricité (Supélec) in 1984.

He began his career developing embedded microcontrollers and printed circuits boards for spatial projects with Thales (formerly Thomson) for five years. Then, he moved to the aeronautic domain with TEAM, a world leader equipment supplier, where for nine years, he was in charge of the electronic design office, including research and development, for analog and digital audio intercommunication systems. For 14 years, he has been working in the automotive industry. For five years he was with Valeo, where he managed an electric and electronic architecture innovation team and an embedded electronic control units research team. He is currently with PSA Peugeot Citroën, Vélizy-Villacoublay, France, leading an advanced research team in electricity, electronics and optoelectronics in connection with the best research institutes in the world. For ten years, he has been recognized as a senior expert in this field.

Françoise Simonot-Lion was a Professor of computer science with the University of Lorraine, Nancy, France. Between 1997 and 2010, she was the Scientific Leader of the Real-Time and Interoperability (TRIO) Project Research Team, Lorraine Research Laboratory in Computer Science and Its Applications/National Polytechnic Institute of Lorraine (LORIA/INPL). Since 2010, she has been the Head of LORIA. Her research interests include the modeling and analysis of real-time distributed systems. She is involved in several research collaborations with the automotive industry. She coauthored more than 100 technical papers in the area of real-time systems modeling and analysis.

Prof. Simonot-Lion is an Associate Editor for the IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS. She was a Cochair of the Automotive Electronic and Embedded Systems Subcommittee of the IEEE Industrial Electronic Society Technical Committee on Factory Automation (IES TCFA).