Implementing mixed-criticality synchronous reactive systems upon multiprocessor platforms

Sanjoy Baruah
The University of North Carolina at Chapel Hill

Abstract— **In designing large complex safety-critical systems that are subject to certification, current industrial practice is centered on the use of high-level abstract design tools. Using such tools greatly facilitates the process of coming up with certifiably correct system designs; however it is a challenge to obtain resource-efficient implementations of the designs thus produced upon actual execution platforms. This paper explores the use of some recent approaches to the scheduling of mixed-criticality systems, to obtain efficient multiprocessor implementations of systems that were designed using tools based on the popular synchronous reactive paradigm of computation.**

I. Introduction

In designing and implementing safety-critical real-time embedded application systems, the system builders are motivated by two significant concerns. On the one hand, the safety-critical nature of the applications makes it imperative that the systems be implemented in a correct – often, certifiably correct – manner. On the other hand, since many embedded systems are implemented upon severely resource-constrained platforms it is desirable that the implementations make very efficient use of platform resources.

In the early years of the discipline of real-time computing, these twin goals of *correctness* and *efficiency* were achieved by keeping things very simple. Safety-critical real-time systems were restricted to being responsible only for simple, highly repetitive, functionalities. Such systems were often implemented as carefully hand-crafted programs executing upon very simple (and hence highly predictable) processors. Gradually, however, things grew more complicated. The requirements placed upon real-time systems became far more complex. More powerful platforms were needed that were typically less deterministic in that their run-time behavior was not predictable at design time. It was no longer possible to reason about an entire system in all its detail; instead, *abstractions* needed to be introduced that highlighted certain features of a system while concealing less important ones.

In designing such abstractions, the real-time systems research community appears to have split into two and diverged down two different paths.

- Some researchers focused primarily on the issue of *efficiency*, and came up with abstractions that enable resource-efficient implementation. (Many prominent researchers engaged in this form of research are very active in the ECRTS community, and a lot of outstanding results of their research efforts have been presented at prior editions of ECRTS.) Such abstractions include recurrent –periodic and sporadic– task models; priority-based scheduling algorithms; formal models for preemptive and non-preemptive uni- and multi-processor computing platforms; models, algorithms, and protocols for real-time networking; etc. Systems specified according to these abstractions can be implemented in a highly-resource-efficient manner, but it is arguably true that these abstractions are at too "low" a level to allow a designer to specify large complex systems in a manner that inspires great confidence regarding the correctness of these specifications.
- Other researchers focused on *provable correctness*, and came up with a different set of abstractions. These abstractions have resulted in some very powerful model-based design (MBD) techniques that find widespread use in industry. The abstractions underlying MBD techniques (including, e.g., the synchrony assumption [7], the actors abstraction [19], etc.) tend to have a strong focus on formal methodologies and proof techniques; for various reasons, research into this kind of work has been under-represented in ECRTS. Large complex systems may be specified according to these abstractions and non-trivial correctness (such as safety, liveness, progress, etc.) properties of these systems proved. But we do not know how to obtain efficient implementations of systems specified in this manner, upon current advanced platforms.

The research described in this paper is part of a larger project that asks whether these two currently distinct paths of real-time systems research can be re-integrated. That is, can real-time scheduling theory techniques that were developed to enable efficiency of implementation, be applied to obtain more resource-efficient implementations of systems that are specified according to higher-level abstractions focusing on provable correctness? In this paper, we focus on the following specific question: *Can recent advances in real-time scheduling theory be extended to allow for more efficient* ***multiprocessor*** *implementations of* ***mixed-criticality*** *real-time systems that are designed using the* ***synchronous reactive*** *model*? We now explain the various component phrases in this statement in greater detail.

Mixed criticality systems. There is an increasing trend in embedded systems towards integrating multiple functionalities on a common platform. Such platforms support functionalities of different degrees of importance or criticalities, with some of the more safety-critical functionalities subject to mandatory

certification by statutory *Certification Authorities* (CAs).

Current approaches to achieving certification in such integrated platforms are centered on ensuring complete isolation between applications of different criticalities. However, it has been observed that such separation makes sub-optimal use of platform resources, since *(i)* the pessimism typically needed for obtaining certification for the safety-critical functionalities requires severe over-provisioning of platform resources to these functionalities, and *(ii)* the very principle of separation rules out the "reclaiming" of these over-provisioned resources for executing non-critical functionalities. The problem of meeting rigorous certification requirements in such systems while simultaneously making efficient use of platform resources has recently attracted widespread attention in the real-time scheduling theory community (see, e.g., [29], [10], [2], [14], [5]; in addition, this topic has been the subject of multiple papers presented recently at ECRTS [4], [30], [24], [11], [25], [1], [20].

Synchronous reactive (SR) systems. Software modeling and development methodologies and commercial tools based on the *synchronous reactive* [7], [15] model of computation, such as Simulink from MathWorks(`www.mathworks.com`) and SCADE from Esterel (`www.esterel-technologies.com`), are widely used in the design and implementation of embedded control systems, particularly in the automotive and the aeronautics industries. In the SR approach, the semantics – the behavioral aspects – of reactive systems are specified or formulated under an assumption called the synchrony hypothesis. The synchrony hypothesis asserts that *the underlying platform is infinitely fast and, hence, the reaction of the system to an input event is instantaneous*. The reaction intervals are thus reduced to reaction instants and do not overlap with each other. The behavior of a system can be thought of as going through a potentially infinite series of steps, one occurring at each "logical" time-instant (often called a *tick* or a *round*): the system reads in its inputs at the time-instant corresponding to the t'th round and, based on its current state and these inputs, instantaneously computes the resulting outputs and updates its current state, and then does nothing until the time-instant corresponding to the $(t+1)$'th round. At this time-instant the system again reads in its inputs and instantaneously computes the resulting outputs and updates its state, and then waits until the time-instant corresponding to the $(t+2)$'th round, and so on.

The synchronous abstraction makes reasoning about concurrency a lot easier by eliminating the non-determinism resulting from interleaving of concurrent behaviors. This allows deterministic semantics, therefore making synchronous systems amenable to formal analysis and verification and thereby facilitating the process of obtaining statutory certification. These features help explain the immense popularity of software development methodologies based on the SR model of computation.

However, the undoubted benefits of SR models of computation do come at a price. The physical platforms on which systems are to be implemented do not satisfy the synchrony hypothesis: actions take time to execute. Semantics-preserving implementations of an SR model must therefore choose a time-unit large enough so that all the actions assumed to occur atomically at one instant complete execution upon the underlying platform strictly prior to the next instant[1]. Due to this and related factors, techniques for obtaining actual implementations (on real hardware) from SR models tend to make poor use of the platform resources. This is particularly true when compared to implementations of models that are specified using *task-based models* (see, e.g., [21], [22]), and scheduled using priority-based scheduling strategies such as RM or EDF [21].

This, then, is the trade-off involved in using SR-based design methodologies and tools in preference to earlier task-based ones: one gets an *easier to use and formally verify* methodology that, however, tends to make *less efficient use of resources*. As embedded systems have become increasingly more complex and difficult to design, this is a tradeoff that system designers have generally been willing to make; even more so since computational capabilities of computing platforms have, in keeping with the predictions of Moore's law, continued to increase at an exponential rate thereby making the efficiency issue less important. However, *energy* considerations and *thermal* issues are bringing efficiency considerations back to the forefront: even if plenty of computing capacity can be made available on a platform, providing the energy needed to enable all this computing capacity is fast becoming a bottleneck. This problem is further exacerbated in mobile platforms that are not tethered to the power-grid. The related problem of heat-dissipation in order to prevent inadmissible increases in the temperature of the platform is also often a major concern.

Mixed criticality SR systems. As stated above, one of the most significant benefits of the SR model is that it is usually far easier to *formally verify* or validate the correctness of a design. Sophisticated analysis tools based both on theorem-proving [17], [6], [16] and on model-checking [28] have been developed and shown effective. Verification tools associated with specific tool-suites (e.g., with Esterel) have even been certified for use in safety-critical system design.

However, these tools merely demonstrate the correctness of the *model*, not the implementation. In order to retain the correctness properties –and the certification– in moving from a model to an implementation of that model, the implementations are required to make extremely conservative (and hence pessimistic) assumptions, which typically result in severe under-utilization of platform resources. Consider, for instance, the *worst-case execution time (WCET)* of a piece of code. The exact WCET of any non-trivial piece of code is extremely difficult to determine on today's complex hardware architectures; system analysis therefore depends on obtaining

[1]This requirement has been stated [9, p. 101] as the *bounded delay property* of the implementation: there is a maximum delay in completing the execution of the actions representing the system reaction to any input, which is strictly less than the minimum time that elapses between successive rounds.

safe *upper bounds* on the actual WCET. A certification authority may require that the estimate of the WCET of a piece of code be made to a far higher level of assurance than a system designer would have chosen on their own. This may result in the same piece of code being characterized by two different WCET estimates: one by the system designer that is probably a reasonably safe over-estimation of the actual WCET of the code, and a much more conservative one that may be far larger (in some cases, orders of magnitude larger) than the actual WCET or the system designer's WCET estimate. When implementing a model that has been certified correct, in order to retain the certification it is necessary that adequate resources be provisioned that would allow the code to execute for up to its higher WCET estimate; since this is extremely unlikely to happen in practice, most of these provisioned resources go unused.

Under current practice, there is little that can be done about this under-utilization of resources during run-time. However the trend towards mixed-criticality platforms upon which functionalities subject to certification co-exist with functionalities that do not need to be certified, offers the possibility of using the over-provisioned resources in order to execute non-critical (i.e., not subject to certification) code. Such an approach to achieving better resource utilization in embedded systems that are subject to statutory certification requirements has recently been widely studied in the real-time scheduling theory community. The research reported in this manuscript explores the viability of extending some of this research in order to obtain more resource-efficient implementations of SR models on multiprocessor platforms.

Organization. We briefly describe a (simplified version of) the synchronous model of reactive computation in Section II. In Section III we present a formal model for representing the kinds of mixed-criticality SR systems we seek to implement; in Section IV we describe the kinds of scheduling strategies we consider appropriate, and provide arguments to justify our choice. We describe an algorithm for implementing such mixed-criticality SR systems on preemptive uniprocessors in Section V. Section VI contains the technical heart of this paper: an algorithm, accompanied by a correctness proof, for implementing mixed-criticality SR systems efficiently upon multiprocessor platforms. We conclude in Section VII with a discussion on some possible generalizations to the models considered in this paper.

II. Synchronous reactive systems: a brief introduction

This introductory section seeks to present SR systems in a framework that is familiar to researchers in real-time scheduling, and focuses only on those aspects that are relevant to the remainder of this paper. It is *not* intended to be a comprehensive introduction to the subject of SR modeling.

A basic unit of a SR system design is a *component* or a *block*. Each such component has input, output, and state variables; during each round the component reads its input variables and, based on these input values and the current values of its state variables, instantaneously updates its state variables and assigns values to its output variables. The SR model defines special forms of variables called *events*. In contrast to other (non-event) variables, an event variable may take on a special value "absent" (often notated as $\perp$). A component with an input variable that is an event is said to be *event-triggered*; an event-triggered component does not participate in any round in which one or more of its input variables is absent. This allows for the specification of components that are not expected to participate in every round: for instance a component with an input event variable that is only present (i.e., not equal to $\perp$) every T'th round executes periodically with a period equal to T rounds.

A SR system is designed by composing components, each of which may in turn be composed from smaller components in a hierarchical manner; the output variables of a component may become the input variables of a different component. Since the constituent components of a system are all assumed to execute instantaneously during every round (unless a component is event-triggered and its triggering event is absent during that round), care must be taken in composing the components to avoid inconsistent, impossible, or non-deterministic behaviors. Each specific SR-based methodology has its own rules for ensuring this; for the vast majority of such methodologies, these rules restrict the dependencies during any round to being acyclic[2]. Hence the actions (the "jobs," in the parlance of task models) that are to be executed during any round can be represented as a *directed acyclic graph (DAG)* with the directed edges denoting precedence dependencies. Implementing an SR model into code on a target execution platform requires that during each round, all the actions (the jobs) in the DAG that need to be executed for that round complete execution before the start of the next round.

Current commercial code generators (e.g., the ones for implementing Simulink models – these include Simulink Coder (formerly known as Real-Time Workshop) from Mathworks and TargetLink from dSPACE (`www.dspace.com`)) provide correct but often inefficient implementations on some specific target platforms. More sophisticated implementations have been proposed in an academic setting (see, e.g., [23] and the references therein). Most of these implementations are targeted at uniprocessor platforms; although some commercial code generators, e.g., Simulink Coder, allow for the specification of multi-core target platforms, it is not clear whether they are actually generating code for multiple processors, or simply adapting uniprocessor code to execute on multiple processors.

[2]Some SR languages such as Esterel do not restrict that dependencies be acyclic; they instead require that evaluation of all dependencies converge to a unique fixpoint (see [8, p. 65] for an instructive description of the different approaches taken by different languages). We note that if an a priori bound can be established on the number of iterations needed to converge on the fixpoint, then the fixpoint computation can be modeled as acyclic dependencies by "unrolling" the cyclic dependencies the appropriate number of times; if such an a priori bound cannot be established, then we cannot provide a guarantee on the amount of (real, not logical) time taken to achieve convergence.

III. System Model

An SR model has some inputs defined, and some outputs. In the *mixed-criticality* framework, we envision that some of the outputs are subject to certification and are designated as HI-criticality outputs, while the rest are not subject to certification (and designated as LO-criticality outputs)[3]. For instance, the value computed by an SR component and sent to an actuator may be considered a HI-criticality output; some additional data that are logged for subsequent off-line analysis may be LO-criticality.

Once an SR model of a system has been obtained, it is subject to extensive analysis in order to validate its correctness, and to obtain certification of the HI-criticality outputs. (Often, this analysis is conducted using model-checking which results in an exploration of reachable configurations of the system.) After the model is determined to be correct, it remains to obtain an implementation of the model upon a target execution platform. It is this step that is of interest to us in the remainder of this manuscript.

As stated in Section II above, most SR frameworks restrict that the dependencies amongst the actions (the jobs) that need to be executed during any given round are in the form of a directed acyclic graph (DAG). In the extreme case, this DAG would include all the actions in the SR model; more realistically, the analysis of the model would use information regarding the occurrence of external input events to identify maximal subsets of actions that are enabled during any round. (Such information is easily obtained as a side-effect of performing certain forms of model-checking; it may otherwise be obtained by analysis of, for example, the periodicity of external input events and the manner in which different components interact with one another.) Regardless of how we obtain it, we assume the following model for the set of actions that is to execute during any particular round:

1) The execution that needs to be performed on a mixed-criticality SR system during any given round is represented as a directed acyclic graph (DAG) with designated *input nodes* and *output nodes*. Each node in the DAG represents a *job* that needs to be executed, while edges represent dependencies – if (j_i, j_j) is an edge then job j_i must complete execution before job j_j can begin execution. Input (output, respectively) nodes are responsible for reading in the inputs to (producing the outputs from, resp.) the system during that round.
2) Each node j_i is characterized by a LO-criticality WCET $C_i(\text{LO})$ and a HI-criticality WCET $C_i(\text{HI})$, denoting the WCET for the job represented by the node as estimated by the system designer and the certification authorities (CAs) respectively. (We assume that the CA is more pessimistic than the system designer: $C_i(\text{LO}) \leq C_i(\text{HI})$ for all j_i.)
3) A ***deadline*** or *makespan bound* D is specified for the entire DAG – this is the maximum delay assumed for the SR model by the bounded delay property [9, p. 101] mentioned in Section I.[4]
4) Each output node is specified as being either HI-criticality or LO-criticality; the CA is only interested in ensuring that the HI-criticality output nodes complete by the deadline, whereas the system designer seeks to ensure that all output nodes complete by the deadline.

Example systems are depicted pictorially in Figures 1 and 3.

Given a system specified in this manner, the goal is to determine a certifiably correct scheduling strategy. We will discuss what constitutes an acceptable scheduling strategy more deeply in Section IV; for now, we define a certifiably correct scheduling strategy as follows:

Definition 1: A ***certifiably correct*** scheduling strategy is one which guarantees that

1) If no node executes beyond its LO-criticality WCET, then all the output nodes complete execution by time-instant D; and
2) If no node executes beyond its HI-criticality WCET then all the HI-criticality output nodes complete execution by time-instant D.

■

In informal terms, if the system designers' WCET assumptions were the correct ones then all the outputs are obtained within the specified deadline, whereas if the system designers' WCET assumptions turn out to be incorrect but the CA's WCET assumptions hold, then only the HI-criticality outputs are guaranteed to be obtained.

As a **preprocessing** step, we assign each non-output node in the DAG a criticality of LO or HI according to the following rules:

- If a node is a predecessor of a HI-criticality node, then it is assigned HI criticality.
- All nodes not assigned HI criticality by the above rule are assigned LO criticality.

In the example of Figure 1, the non-output nodes j_1 and j_2 are both assigned HI criticality since they are immediate predecessors of the HI-criticality output node j_4.

IV. Characterizing suitable scheduling strategies

Before deriving algorithms for obtaining suitable scheduling strategies for mixed-criticality SR systems, let us consider for

[3]Thus far in this paper, we have identified two criticality levels – needing certification, and not needing certification. However, in many safety-critical application domains more than two criticality levels (also called, e.g., Safety Integrity Levels (SILs) or Design Assurance Levels (DALs) in different standards documents) are specified, with functionalities at higher criticality levels subject to more rigorous validation requirements. For ease of presentation in this paper, we will initially make the simplifying assumption that there are just two criticality levels that we will call LO and HI. In Section VII we describe how our results extend in a straight-forward manner to systems with more than two criticality levels.

[4]An alternative possibility is that a deadline is *not* specified; instead, the objective is to complete all output nodes as soon as possible. This is achieved by modeling the scheduling problem as a *makespan minimization* problem, which is then solved using techniques essentially identical to the ones discussed in this paper. In this makespan minimization version, an implementation of the SR system is viable if the sum of the makespans of the DAGs representing a sequence of time-steps is no larger than a specified end-to-end deadline bound.

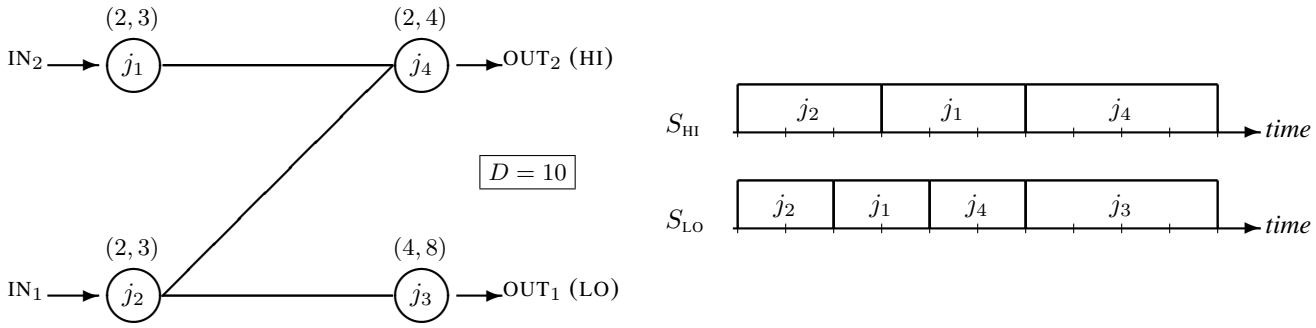

Fig. 1. An example task graph, and its scheduling tables. All edges in the task graph are assumed to be directed left to right. The ordered pair above node j_i denotes $\big(C_i(\text{LO}), C_i(\text{HI})\big)$ — the job's WCET estimates at LO-criticality and HI-criticality. This task graph has a specified deadline $D = 10$. (The scheduling tables are explained in Section IV. Each "tick" along the time axis in these tables represents one time-unit.)

a moment the *kinds* of properties we desire in a scheduling strategy in order to facilitate the certification process.

In the *time-triggered (TT)* paradigm [18] of real-time scheduling, activities in the system are triggered by the progression of time. A schedule for the entire duration of a system's execution is computed prior to run-time, and the scheduling decision that is made at each instant during run-time is completely determined by examining this pre-computed schedule, represented, e.g., in a scheduling table.

TT scheduling is characterized by complete determinism, and is hence particularly easy to verify and have certified. However, the TT paradigm offers limited flexibility: once the schedule is computed (prior to run-time), it is not possible to modify it in response to events that may have occurred during run-time. Extensions to the basic time-triggered scheduling paradigm have been proposed to remedy this shortcoming; in one such extension [12], multiple scheduling tables are pre-computed and the occurrence of certain run-time events triggers a "mode change" and a consequent transition from using one particular pre-computed schedule to using a different one.

A time-triggered strategy for mixed-criticality scheduling has recently been proposed [3], based upon such a mode change approach[5]. According to this strategy, multiple scheduling tables are constructed prior to run-time for any given mixed-criticality system. During run-time all scheduling decisions are initially made in accordance with one of these scheduling tables. Certain run-time events (which are identified and specified prior to run-time) may trigger a change as to which scheduling table to use.

Such a mode change approach to mixed-criticality scheduling maintains much of the determinism of TT scheduling, since the job selected for execution at each instant in time during run-time depends only on what mode-change actions (if any) have occurred prior to this time instant. The transition itself is considered during schedule construction as well. In this manner the appeal of time triggered execution for certification is for the most part maintained, while providing added efficiency with respect to the utilization of platform resources.

Under such an approach, a possible set of scheduling tables S_{LO} and S_{HI} for implementing the example task graph of Figure 1 on a uniprocessor platform is depicted to the right of the DAG. Initially, run-time dispatching decisions are taken in accordance to schedule S_{LO}. That is, jobs j_2, j_1, j_4 and j_3 are allowed to execute over the time-intervals $[0, 2), [2, 4), [4, 6)$, and $[6, 10)$ respectively. If any job completes execution before the end of its allocation, the processor is idled for the remainder of this allocation. If a job reaches the end of its allocation without completing execution, a mode-change is immediately triggered and subsequent scheduling decisions are made according to S_{HI}. For instance suppose that j_2 completes within two time units but j_1 does not complete within two time units. At time-instant 4 the dispatcher immediately switches to using the scheduling table S_{HI}; as a consequence, j_1 is allowed to execute until time-instant 6, and j_4 over $[6, 10)$.

The reader may verify that scheduling in this manner is a correct scheduling strategy, in the sense that if all jobs complete execution upon having executed for no more than their LO-criticality WCETs then both the output jobs j_3 and j_4 complete execution, whereas if any job executes for more than its LO-criticality WCET (but no job executes for more than its HI-criticality WCETs) then the HI-criticality output job j_4 completes execution, by the specified deadline of 10.

To reiterate what we have previously stated: the system designers, who are the ones responsible for generating these scheduling tables, do not expect to ever have a mode-change occur — their expectation is that their own WCET estimates are correct, and the LO-criticality mode scheduling table S_{LO} will always suffice. However, they need to construct the other schedule — the one that the system would transition to using in case their WCET assumptions turn out to be incorrect — in order to have their design pass certification by the CAs.

Preemptions and migrations. A schedule is said to include *preemptions* if an executing job may be preempted and have its execution resumed at a later point in time; such a schedule is called a *preemptive* schedule. A preemptive schedule upon a multiprocessor platform has *migrations* if some preempted job resumes its execution upon a processor that is different from the one it had been executing upon prior to being preempted.

Allowing preemptions and migrations in a scheduling strategy affords benefits (in terms of increased schedulability –

[5]We point out that while [3] only considered systems that could be modeled as *collections of independent jobs* implemented on *preemptive uniprocessors*; in this paper we explain how this approach can be extended to more general systems –those modeled as discussed in Section III– that are implemented on preemptive and non-preemptive uni- and multi-processors.

a greater likelihood that a viable schedule will be found) and drawbacks (primarily, increased run-time overhead, and a more difficult problem of WCET estimation). Although a detailed discussion of these benefits and drawbacks is beyond the scope of this document, we will briefly touch upon some of the issues in Section VII. For now, we merely point out here that our uniprocessor scheduling strategy (Section V) is non-preemptive whereas our multiprocessor scheduling strategy (Section VI) uses both preemptions and migrations. This multiprocessor strategy can easily be modified to disallow preemptions and migrations, at a cost of reduced schedulability – the precise loss of schedulability in doing so is quantified in Lemma 7.

V. Constructing uniprocessor scheduling tables

Given a system represented as a DAG G and a deadline D that is to be implemented on a single processor, we first obtain a topological ordering of the nodes in the DAG, with HI-criticality nodes listed first whenever possible.

Let us simulate this step on the example graph of Figure 1. Recall that we had determined that during the preprocessing step nodes j_1, j_2, and j_4 are all labeled as HI-criticality nodes, while node j_3 is labeled a LO-criticality node.

1) We can first list either j_1 or j_2, since neither has any unlisted predecessors. Let us suppose we choose to list j_2 first.
2) This results in j_3 also becoming eligible for listing: it has no unlisted predecessors. The set of nodes eligible for being listed next is therefore $\{j_1, j_3\}$. Since we are required to list HI-criticality nodes before LO-criticality ones, we must list j_1.
3) This results in j_4 now becoming eligible for listing; the set of nodes eligible for being listed next is therefore $\{j_3, j_4\}$. Since j_4 is HI-criticality while j_3 is LO-criticality, we list j_4 next, and j_3 last.

The final topological ordering of the nodes that we obtain is thus j_2, j_1, j_4 and j_3.

Lemma 1: All HI-criticality nodes appear before all LO-criticality nodes in this topological ordering.

Proof: Suppose for a contradiction that some HI-criticality node j_ℓ were to appear immediately after a LO-criticality node j_k. Since our ordering scheme favors HI-criticality jobs over LO-criticality ones, this implies that there is a directed edge from j_k to j_ℓ. But by our method for allocating criticalities to nodes, this would require that node j_k also be labeled as a HI-criticality node, thereby obtaining a contradiction to the assumption that j_k is a LO-criticality node. ■

Once a topological ordering of the vertices has been obtained, we obtain the two scheduling tables as follows:

- S_{LO} is obtained by scheduling each job according to this topological ordering, with each job j_i being assigned an execution amount equal to $C_i(\text{LO})$. By Lemma 1, this implies that all HI-criticality jobs all allocated execution before any LO-criticality job is allocated execution.
- S_{HI} is obtained by scheduling only the HI-criticality jobs, in the order in which they appear in this topological ordering, with each such job j_i being assigned an execution amount equal to $C_i(\text{HI})$.

Finally, we declare success if and only if the makespan (the duration) of both schedules does not exceed the specified bound D.

Lemma 2: This schedule-generation algorithm is *correct*, in the sense that scheduling a system according to the procedure described in Section IV by using the scheduling tables generated as described above, corresponds to a certifiably correct scheduling strategy (Definition 1).

Proof: According to the procedure described in Section IV, we start out during run-time executing jobs according to the scheduling table S_{LO}. If we never transition to S_{HI} then it is evident that the lemma holds: all jobs have completed within their allocated execution times.

Suppose now that we transition to using table S_{HI}, because some job j_i executes beyond its LO-criticality WCET without completing. Our obligation, to show certifiable correctness, is to demonstrate that each HI-criticality output job will complete execution (provided each such job executes for no more than its HI-criticality WCET). To show this, we observe that

- If j_i is a LO-criticality job, then all HI-criticality jobs must have already completed execution. This follows from Lemma 1 and the fact that S_{LO} executes jobs in the same order as they appear in the topological sorting.
- If j_i is a HI-criticality job, then no LO-criticality job has received any execution thus far. And, since both S_{LO} and S_{HI} execute the HI-criticality jobs in the *same* order, it must be the case that
  1) All HI-criticality jobs that were scheduled prior to j_i in S_{HI} have already completed execution (since they were also scheduled prior to j_i in S_{LO} and did not trigger a mode-change).
  2) By the time-instant that the transition from using S_{LO} to S_{HI} occurred, job j_i must have received at least as much execution in S_{LO} as it would have if we had been following S_{HI} from the very beginning. Hence there is sufficient computing capacity allocated in S_{HI} for all the remaining HI-criticality jobs (including j_i) to complete by the deadline D.

The lemma follows. ■

Lemma 3: This uniprocessor schedule-generation algorithm is *optimal*, in the sense that it can generate scheduling tables for any DAG that can be scheduled correctly by any algorithm (including clairvoyant ones).

Proof: We first derive a necessary condition for a system to be schedulable.

- First, consider the behavior of the system in which each job j_i needs exactly $C_i(\text{LO})$ units of execution – by definition, such a behavior has criticality level LO. It is therefore necessary for schedulability that this behavior be schedulable in such a manner that each job j_i receives at least $C_i(\text{LO})$ units of execution. Let F_{LO} denote the

completion time of the last job in any preemptive work-conserving schedule, in which each j_i receives exactly $C_i(\text{LO})$ units of execution.

- Consider next the behavior of the system in which each job j_i needs exactly $C_i(\text{HI})$ units of execution – by definition, such a behavior has criticality level HI. It is necessary that this behavior be schedulable in such a manner that each HI-criticality job j_i receives at least $C_i(\text{HI})$ units of execution (while the LO-criticality jobs need receive no execution). Let F_{HI} denote the completion time of the last job in any preemptive work-conserving schedule, in which each HI-criticality job j_i receives exactly $C_i(\text{HI})$ units of execution while each LO-criticality job receives no execution at all.

A necessary schedulability condition is hence as follows:

$$\max\{F_{\text{LO}}, F_{\text{HI}}\} \leq D$$

Notice that the makespan of the schedule S_{LO} is exactly equal to F_{LO}, and that of schedule S_{HI} exactly equal to F_{HI}. It therefore follows that both these schedules will complete by the deadline specified for the DAG if and only if this necessary schedulability condition is satisfied. ■

VI. Constructing multiprocessor scheduling tables

Designing an algorithm for generating scheduling tables for scheduling mixed-criticality SR systems on multiprocessor platforms turns out to be far more challenging than it was in the uniprocessor case. Before proceeding further, we briefly review (Section VI-A below) some well-known results concerning the scheduling of "regular" (i.e. not mixed-criticality) systems upon multiprocessor platforms.

A. Multiprocessor scheduling to minimize makespan

It is known that determining a schedule with minimum makespan for a given collection of regular precedence-constrained jobs is NP-hard in the strong sense [27]. (In fact if preemption is forbidden, this intractability result holds for collections of independent jobs; i.e., even if the precedence constraints are empty.) Fortunately, an efficient algorithm, known as ***List Scheduling (LS)*** [13] is known for solving this problem approximately. LS can be defined non-preemptively or preemptively. In either form LS accepts as input a collection of jobs with inter-job precedences represented as a DAG and a total *priority ordering* defined on the jobs[6].

- *Non-preemptive LS:* Whenever a processor is idled, LS chooses for execution the highest-priority ready job (that is, some unscheduled job whose predecessors have all completed execution), and executes this job non-preemptively to completion. (If there are no ready jobs, then the processor is idled until some job becomes ready due to the completion of the execution of its predecessors on other processors.)

[6]This total priority ordering can be represented as a *list* of the jobs – hence the name *List Scheduling*.

GENSCHEDMULTIPROC(G, D, m)

1) Build the m-processor schedule S_{HI} by applying non-preemptive LS to only the HI-criticality jobs in G, under the assumption that each job may need to execute for up to its HI-criticality WCET. Any priority ordering of these jobs may be used.
2) Define a priority ordering $\prec$ amongst the jobs in G. The priority assigned to HI-criticality jobs are based on the time they *begin* executing in the scheduling table S_{HI} obtained above: jobs with an earlier start time are assigned greater priority (ties broken arbitrarily). All HI-criticality jobs have priority over any LO-criticality job; the LO-criticality jobs may have any priority ordering defined amongst themselves.
3) Build the m-processor schedule S_{LO} by applying *preemptive* LS on all the jobs in G with these job priorities, under the assumption that each may execute for up to its LO-criticality WCET.
4) Declare ***failure*** if either S_{LO} or S_{HI} has makespan $> D$

Fig. 2. Pseudocode representation of the multiprocessor schedule-generation algorithm

- *Preemptive LS:* At each instant the m processors are executing the m highest-priority ready jobs that have not yet completed execution. Hence if a processor is executing some job j_i and some higher-priority job j_j becomes ready, the processor preempts the execution of j_i and begins executing j_j instead. (Job j_i will resume execution later when it becomes one of the m highest-priority jobs that have not yet completed execution.)

LS was shown [13] to have a worst-case approximation ratio of $\left(2 - \frac{1}{m}\right)$ when implemented on an m-processor identical platform. Specifically, it was shown that if an optimal algorithm can generate an m-processor schedule (preemptive or non-preemptive) of length L then either version of LS, provided with *any* priority ordering on the jobs, generates a schedule of length $\leq \left(2 - \frac{1}{m}\right) \times L$. It has recently been shown [26] that $(2 - \xi)$ is likely to be a lower bound on the worst-case approximation ratio of any polynomial-time approximation algorithm, for some positive constant ξ that tends to zero; hence from the perspective of this metric LS is probably close to the best polynomial time algorithm possible.

B. Algorithm GENSCHEDMULTIPROC: description

We now turn our attention back to our problem of building scheduling tables for mixed-criticality SR systems on multiprocessor platforms. Given a system represented as a DAG G and a deadline D that is to be implemented on a platform consisting of m identical processors, the procedure for generating the scheduling tables S_{LO} and S_{HI} is given in Figure 2. As can be seen from the pseudo-code listing, this algorithm

1) first generates the HI-criticality scheduling table S_{HI} using non-preemptive LS with any priority listing amongst the jobs;
2) uses the schedule S_{HI} so determined to define a priority ordering that will be the "list" used for list-scheduling when generating S_{LO}; and

3) generates S_{LO} using preemptive LS, with the list being the priority ordering determined in the step above.

Failure is declared if either S_{LO} of S_{HI} has makespan greater than the specified bound D.

We illustrate this procedure by constructing a two-processor schedule for the simple example SR system depicted in Figure 3 on the left (the generated scheduling tables are depicted on the right). Note that both output nodes of this SR system are labeled as HI-criticality outputs, that all the HI-criticality WCETs are 5, and that j_2's LO-criticality WCET is 4 while all other nodes' LO-criticality WCETs are 5.

- It may be validated that the preprocessing step assigns HI criticality to all the nodes in this DAG – this is only to be expected, since there are no LO-criticality outputs in the DAG and all the non-output nodes are predecessors to some output node.
- We obtain the HI-critcality scheduling table S_{HI} shown in the figure by applying LS to these jobs, under the assumption that each may execute for its HI-criticality WCET (i.e., for five time units). We use the lexicographic priority ordering to obtain this table; as stated in Figure 2, any priority ordering may be used.
- Based on this schedule we define the following priority ordering $\prec$ on the jobs, which respects the requirement that jobs that begin execution earlier in S_{HI} have greater priority (ties broken arbitrarily):

  $$j_1 \prec j_2 \prec j_3 \prec j_4 \prec j_5 \prec j_6$$

  (We note that since all the predecessors of a job j_i must complete execution before job j_i begins execution, these predecessors must also have begun execution prior to job j_i. Therefore, it follows that all predecessors of each job j_i occur prior to j_i in this total priority ordering.)
- Finally, we use preemptive LS with this defined priority ordering and under the assumption that each job may execute for its LO-criticality WCET (i.e., j_2 may execute for 4 time-units and the other jobs each for 5).
  - At time-instant 4, job j_2 completes execution. Since jobs j_3 and j_4 are not yet ready (since their predecessor job j_1 has not completed), j_5 is the highest-priority job that is eligible to execute. It therefore begins execution at time-instant 4.
  - At time-instant 5, j_1 completes execution; this causes j_3 and j_4 to become ready to execute. Since *preemptive* LS is being used, this results in the preemption of the execution of the lower-priority job j_5 at time-instant 5.
  - Job j_5 resumes execution at time-instant 10, after jobs j_3 and j_4 complete execution and vacate their processors.

  The resulting LO-critcality scheduling table S_{LO} shown in the figure. (Note that j_5 resumes on a different processor than the one it has been execution in prior to preemption, since we allow for inter-processor migration.)

C. *Algorithm* GENSCHEDMULTIPROC*: properties*

Since much of the work in Algorithm GENSCHEDMULTIPROC is done in the calls to the list scheduling algorithm LS, it should be evident that GENSCHEDMULTIPROC can be implemented about as efficiently as LS; i.e., with a run-time complexity that is a low-order polynomial in the number of nodes (jobs) in the DAG.

We now derive correctness and performance properties of Algorithm GENSCHEDMULTIPROC. We will formally show that Algorithm GENSCHEDMULTIPROC is correct (Lemma 5), and that although it is not optimal[7], its deviation from optimality is no more than that of LS when LS is scheduling regular (non mixed-criticality) DAGs (Lemma 6).

But first, we need an additional result – Lemma 4 below, which asserts that at all times each HI-criticality job makes at least as much progress towards completion in schedule S_{LO}, as it does in schedule S_{HI}.

Lemma 4: Consider schedules S_{LO} and S_{HI} for some DAG, constructed by Algorithm GENSCHEDMULTIPROC.

1) If a HI-criticality job has not completed execution in S_{LO} by some time-instant t, then it has executed for at least as much in S_{LO} as in S_{HI} over the interval $[0, t)$.
2) Each HI-criticality job completes execution no later in S_{LO} than in S_{HI}.

Proof: We observe first that since all predecessors of HI-criticality jobs are also HI-criticality jobs and since LO-criticality jobs are assigned lower priority than HI-criticality jobs, the presence of LO-criticality jobs has no influence whatsoever on the scheduling of the HI-criticality jobs in the *preemptive* schedule S_{LO}. This lemma only makes assertions about HI-criticality jobs; henceforth in this proof, therefore, we can safely ignore the LO-criticality jobs.

Let $j'_1, j'_2, \ldots,$ denote the priority ordering on the HI-criticality jobs determined during the second step of Algorithm GENSCHEDMULTIPROC: $j'_\ell \prec j'_{\ell+1}$ for all ℓ. Our proof is by induction on the jobs, according to this priority ordering.

- **[Basis]:** Since each job's LO-criticality WCET is $\leq$ its HI-criticality WCET, job j'_1 completes no later in S_{LO} than in S_{HI}. Furthermore, j'_1 begins executing at time-instant zero in S_{LO}, and executes non-preemptively to completion. The statement of the lemma therefore holds for job j'_1.
- **[Induction:]** Suppose all the jobs $j'_1, \ldots, j'_{\ell-1}$ complete no later in S_{LO} than in S_{HI}. Then j'_ℓ becomes ready to execute (in the sense that all its predecessors have completed) no later in S_{LO} than in S_{HI}. Hence if j'_ℓ is executing at some instant in S_{HI} but not in S_{LO}, it must be because it has already completed execution prior to that instant in S_{LO}. It immediately follows that j'_ℓ completes no later in S_{LO} than in S_{HI}; prior to the instant that it completes in S_{LO}, it executes in S_{LO} at all time-instants that j'_ℓ executes in S_{HI}. The statement of the lemma therefore holds for job j'_ℓ as well.

[7]This is not at all surprising, since the simpler problem of scheduling DAGs of non-mixed-criticality jobs to minimize makespan is strongly NP-hard.

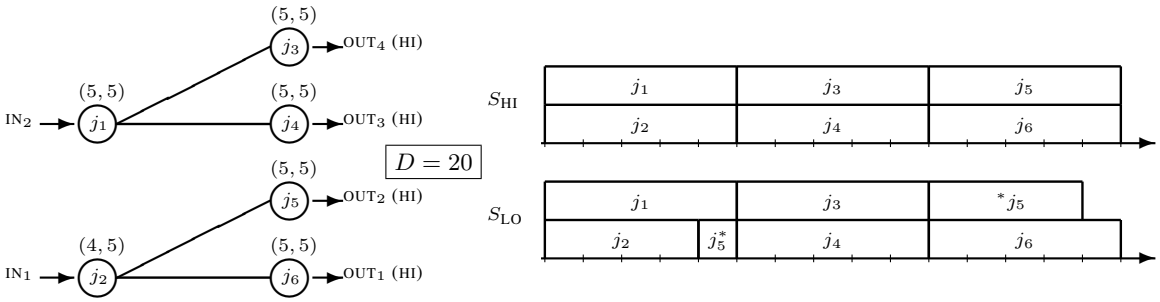

Fig. 3. Example DAG, and 2-processor scheduling tables. Each scheduling table is depicted visually by stacking the schedule on each of the two processors atop each other. The * denotes that the job j_5 is preempted/ resumed.

■

Lemma 4 above has established that at each time-instant t each HI-criticality job has either

1) Completed execution in S_{LO} by time-instant t, or
2) Executed for at least as much in S_{LO} as in S_{HI} over the interval $[0, t)$.

These facts allow us to demonstrate the correctness of our scheduling algorithm:

Lemma 5: Algorithm GENSCHEDMULTIPROC is correct, in the sense that scheduling a system according to the procedure described in Section IV by using the scheduling tables generated by GENSCHEDMULTIPROC, corresponds to a certifiably correct scheduling strategy[8].

Proof: According to the time-triggered run-time scheduling procedure that is described in Section IV, we start out executing according to the scheduling table S_{LO}. If we never transition to S_{HI} then it is evident that the lemma holds: all jobs have completed within their allocated execution times.

Suppose now that we transition to using table S_{HI}, because some job has executed beyond its LO-criticality WCET without signaling that it has completed execution. To prove certifiable correctness it suffices to demonstrate that each HI-criticality job will complete execution by the deadline D, provided each executes for no more than its HI-criticality WCET.

Let t denote the time-instant at which the transition from using S_{LO} to S_{HI} occurs. By Lemma 4 above, in S_{LO} each HI-criticality job j_i has either already completed execution or has received at least as much execution over $[0, t)$ as it would have in S_{HI} during the same interval. But since each such j_i is guaranteed to receive $C_i(\text{HI})$ units of execution in S_{HI} by the deadline D, it follows that j_i will therefore receive enough execution after the transition to complete by the deadline D.

■

Lemma 5 above asserts the correctness of Algorithm GENSCHEDMULTIPROC; Lemma 6 below characterizes its performance.

Lemma 6: If an optimal clairvoyant algorithm can schedule a given DAG on m processors by a deadline D, then Algorithm GENSCHEDMULTIPROC generates m-processor scheduling tables S_{LO} and S_{HI} for the same DAG with specified deadline $\left(2 - \frac{1}{m}\right)D$.

Proof: This is a direct consequence of the $\left(2 - \frac{1}{m}\right)$ approximation ratio of LS. It is evident that if an optimal clairvoyant algorithm can schedule a given DAG on m processors by a deadline D, an optimal schedule for only the HI-criticality jobs would be of duration $\leq D$. Hence the scheduling table S_{HI}, which is constructed using nonpreemptive LS, has duration $\leq \left(2 - \frac{1}{m}\right)D$. Similarly an optimal schedule for all the jobs in which each executes for its LO-criticality WCET would also be of duration $\leq D$. The scheduling table S_{LO}, which is constructed using preemptive LS, therefore also has duration $\leq \left(2 - \frac{1}{m}\right)D$. ■

VII. SUMMARY AND DISCUSSION

The research described in this document addresses the following question: can we obtain efficient multiprocessor implementations of mixed-criticality real-time systems that are modeled using the synchronous reactive model? We believe the answer to this question is "yes"; in this document, we have taken an initial step towards validating this belief by applying advanced (traditional) multiprocessor scheduling theory as well as recent results from real-time scheduling to design an algorithm for implementing mixed-criticality SR systems upon multiprocessor platforms. In order to be able to do so, we have needed to devise appropriate scheduling models for the kinds of SR systems we seek to schedule, and to formalize the kinds of scheduling strategies that may be considered appropriate for such systems. We conclude this paper with a discussion on some additional issues that merit mention, but that were not directly discussed in the earlier sections.

More than two criticality levels. Our framework and algorithms are easily extended to deal with mixed-criticality application domains that have more than two criticality levels specified. In scheduling an instance with k distinct criticality levels, we would pre-construct k different scheduling tables, one corresponding to each criticality level. We would start out making dispatching decisions according to the scheduling table that corresponds to the lowest criticality level; if the run-time behavior is revealed to be of a higher criticality level, we immediately switch to making dispatching decisions according to the scheduling table that corresponds to that criticality level.

[8]Recall that *certifiably correct* scheduling strategies are defined in Definition 1.

Thus a maximum of $(k-1)$ such switches may occur during any given execution of the system.

Preemptions and migrations. As we had mentioned in Section IV, there are benefits and drawbacks to allowing preemptions and migrations in the schedules we seek to construct. The benefit is enhanced schedulability; the most obvious drawback is increased overhead: depending upon the characteristics of the platform, performing each preemption and/ or each migration may require a significant amount of time. Additional drawbacks include the unpredictability that results from allowing preemptions or migrations (for instance, in determining WCETs).

Although our uniprocessor scheduling strategy generates non-preemptive scheduling tables, the scheduling tables for multiprocessor platforms that are generated by Algorithm GENSCHEDMULTIPROC may have preemptions and migrations. These can be gotten rid of, but at a cost of decreased schedulability: we state without proof the following result

Lemma 7: If an optimal clairvoyant algorithm can schedule a given DAG on m processors by a deadline D, then a modification of Algorithm GENSCHEDMULTIPROC generates m-processor scheduling tables *with no preemptions or migrations* for the same DAG, with specified deadline $2 \times \left(2 - \frac{1}{m}\right) D$.

■

Equivalently, disallowing preemptions and migrations increases the makespan of the resulting scheduling tables by a further factor of 2.

REFERENCES

[1] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems*, ECRTS '12, Pisa (Italy), 2012. IEEE Computer Society.

[2] S. Baruah, A. Burns, and R. Davis. Response-time analysis for mixed criticality systems. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, Vienna, Austria, 2011. IEEE Computer Society Press.

[3] S. Baruah and G. Fohler. Certification-cognizant time-triggered scheduling of mixed-criticality systems. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, Vienna, Austria, 2011. IEEE Computer Society Press.

[4] S. Baruah and S. Vestal. Schedulability analysis of sporadic tasks with multiple criticality specifications. In *Proceedings of the EuroMicro Conference on Real-Time Systems*, Prague, Czech Republic, July 2008. IEEE Computer Society Press.

[5] S. K. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie. Scheduling real-time mixed-criticality jobs. *IEEE Transactions on Computers*. To appear.

[6] S. Bensalem, P. Caspi, C. Parent-Vigouroux, and C. Dumas. A methodology for proving control systems with Lustre and PVS. *Dependable Computing for Critical Applications*, 1999.

[7] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270 –1282, sep 1991.

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

[9] G. Berry. *The Esterel v5 Language Primer: version v5_91*. Centre de Mathématiques Appliquées, Ecole des Mines and INRIA, 2000.

[10] F. Dorin, P. Richard, M. Richard, and J. Goossens. Schedulability and sensitivity analysis of multiple criticality tasks with fixed-priorities. *Real-Time Systems*, 2010.

[11] P. Ekberg and W. Yi. Bounding and shaping the demand of mixed-criticality sporadic tasks. In *Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems*, ECRTS '12, Pisa (Italy), 2012. IEEE Computer Society.

[12] G. Fohler. Changing operational modes in the context of pre run-time scheduling. *IEICE Transactions on Information and Systems (Special Issue on Responsive Computer Systems)*, E76–D(11):1333–1340, 1993.

[13] R. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[14] N. Guan, P. Ekberg, M. Stigge, and W. Yi. Effective and efficient scheduling for certifiable mixed criticality sporadic task systems. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, Vienna, Austria, 2011. IEEE Computer Society Press.

[15] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Publisherss, 1993.

[16] N. Halbwachs and P. Raymond. Validation of synchronous reactive systems:from formal verification to automatic testing. In *ASIAN'99, Asian Computing Science Conference*, Phuket (Thailand), Dec. 1999. LNCS 1742, Springer Verlag.

[17] L. J. Jagadeesan, C. Puchol, and J. E. Olnhausen. A formal approach to reactive systems software: A telecommunications application in Esterel. *Formal Methods in System Design*, 8:123–151, 1996.

[18] H. Kopetz. *Real-Time Systems - Design Principles for Distributed Embedded Applications*. Springer, 2011.

[19] E. A. Lee, S. Neuendorffer, and M. J. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers*, 2, 2003.

[20] H. Li and S. Baruah. Global mixed-criticality scheduling on multiprocessors. In *Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems*, ECRTS '12, Pisa (Italy), 2012. IEEE Computer Society.

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

[22] A. Mok. *Fundamental Design Problems of Distributed Systems for The Hard-Real-Time Environment*. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1983. Available as Technical Report No. MIT/LCS/TR-297.

[23] M. D. Natale and H. Zeng. Task implementation and schedulability analysis of synchronous finite state machines. In *Work in Progress (WiP) session of the IEEE Real-Time and Embedded Technology And Applications Symposium*, pages 21–24, Apr. 2011.

[24] R. Pathan. Schedulability analysis of mixed-criticality systems on multiprocessors. In *Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems*, ECRTS '12, Pisa (Italy), 2012. IEEE Computer Society.

[25] F. Santy, L. George, P. Thierry, and J. Goossens. Relaxing mixed-criticality scheduling strictness for task sets scheduled with FP. In *Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems*, ECRTS '12, Pisa (Italy), 2012. IEEE Computer Society.

[26] O. Svensson. Conditional hardness of precedence constrained scheduling on identical machines. In *Proceedings of the 42nd ACM symposium on Theory of computing*, STOC '10, pages 745–754, New York, NY, USA, 2010. ACM.

[27] J. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384 – 393, 1975.

[28] E. Vecchié and R. de Simone. Syntax-driven behavior partitioning for model-checking of Esterel programs. *Electron. Notes Theor. Comput. Sci.*, 153:19–35, June 2006.

[29] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings of the Real-Time Systems Symposium*, pages 239–243, Tucson, AZ, December 2007. IEEE Computer Society Press.

[30] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems*, ECRTS '12, Pisa (Italy), 2012. IEEE Computer Society.