Functionally Deterministic Scheduling

Frédéric Boniol*, Claire Pagetti*,**, François Revest**

*IRIT-ENSEEIHT, 2 rue C. Camichel. F31071 Toulouse, France
frederic.boniol@enseeiht.fr,
**ONERA-CERT, 2 av. E. Belin. F31055 Toulouse - France
claire.pagetti,francois.revest@cert.fr

Abstract. The aim of this paper is to prove that the question "*is a scheduling functionally deterministic for a given set of periodic real time tasks and a given conservative scheduling policy*" is decidable. For thar purpose, we encode the tasks and the scheduler by an acyclic stopwatch automaton. We show then that the previous question can be encoded by a decidable reachability problem.

1 Introduction

Embedded systems architecture is classically decomposed into three main parts. The *control software* is often designed by a set of communicating functional modules, also called tasks, usually encoded with a high level programming language (e.g. synchronous language) or a low level one (e.g. Ada or C). Each functional module is characterized by real-time attributes (e.g. period, deadline) and a set of precedence constraints. The *material architecture* organizes hardware resources such as processors or devices. The *scheduler* decides in which order functional modules will be executed so that both precedence and deadline constraints are satisfied.

Behavioral correctness is proved as the result of the logical correctness, demonstrated with the use of formal verification techniques (e.g. theorem proving or model-checking) on the functional part, and the real-time correctness which ensures that all the computations in the system complete within their deadlines. This is a non trivial problem due both to precedence constraints between tasks, and to resource sharing constraints. This problem is addressed by the real-time scheduling theory which proposes a set of dynamic scheduling policies and methods for guaranteeing/proving that a tasks configuration is schedulable.

However, in spite of their mutual dependencies, these two items (functional verification and schedulability) are seldom addressed at the same time: schedulability methods take into account only partial information on functional aspects, and conversely the verification problem of real-time preemptive modules has been shown undecidable (Ermont and Boniol, 2007). To overcome this difficulty, a third property is often required on critical systems, especially for systems under certification: *determinism*, i.e. all computations produce the same results and actions when dealing with the same environment input. The benefit of this property, if ensured, is to limit the combinatorial explosion, allowing an easier abstraction of real-time attributes in the functional view. For instance, preemptive modules may be abstracted by non preemptive ones characterized by fixed beginning and end dates. The interesting consequence

is to allow separated functional and real-time analyses. For ensuring determinism, two ways can be followed: either to force it, or to prove it.

Several approaches were proposed in order to guarantee determinism. One of the simplest manner is to remove all direct communications between tasks. This seems quite non realistic but it can be achieved by developing an adequate architecture, for instance, current computed data are stored in a different memory while consumed input are the ones produced in a precedent cycle. The execution order between tasks within each cycle does not impact the produced values. However, the main disadvantage is to lengthen the response time of the system. This solution is then not suitable for systems requiring short response time.

A second approach is based on off-line non preemptive strategies, such as cyclic scheduling. Provided that functional modules are deterministic, the global scheduled behavior will also be deterministic. This solution is frequently followed by aircraft manufacturer for implementing critical systems such as a flight control system. However this strategy has two main several drawbacks. Firstly this scheduling leads to a low use of resources because tasks are supposed to use their whole worst case execution time (WCET). To overcome this first problem, tasks are required to be as small as possible. Secondly, off-line scheduling strategies often need for over-dimensioning the system in order to guarantee acceptable response times to external events. For that purpose, tasks periods are often to be reduced (typically divided by 2) compared to the worse period of the polled external events.

The guaranty that WCET and BCET (resp. worst and best case execution times) coincide provides a third interesting context. Any off-line scheduling is then deterministic and it is possible to modify the task model to produce a deterministic on-line scheduling. Unfortunately, warranting that BCET is equal to WCET is hardly possible. This can limit the programming task (no use of conditional instruction, or on the contrary use of dead code to enforce equality).

Other more recent approaches are based on formal synhronous programming languages. Systems are specified as deterministic synchronous communicating processes, and are implemented either by a sequential low level code which enforces a static execution order, or by a set of tasks associated with static or dynamic scheduling policies (Scaife and Caspi, 2004; S. Tripakis and Caspi, 2005). Implementation is correct-by-construction, i.e., it preserves the functional semantics (and then determinism). These approaches are interesting, for they allow to by-pass the determinism verification problem. They are implemented by existing indutrial tools such as the Simulink Real Time Workshop code generator. However, they often are based on “ideal” zero-time semantics. If such a semantic model is suitable for homogenous and regular control systems (e.g., a flight control system), it is less appropriate for complex systems composed of non synchronous tasks developed by several teams and integrated into a same digital infrastructure (as in integrated modular avionics).

Previous solutions are not suitable for highly dynamic non synchronous systems with high workload. On-line preemptive scheduling strategies are often optimal, easy to implement, but deeply non deterministic when associated to asynchronous communication models. Problematic configuration appears when there are temporal undeterminism on execution time and preemption. Let us consider for instance the three tasks NA, NF and GF depicted in figure 1 and scheduled by an on-line preemptive scheduling strategy. Depending on the execution times of NF and GF, two runs depicted figure 1 are possible. Considering that task NA produces x_n at time n, there are two possibilities, either NF and GF consume x_n or NF cannot complete its execution before $n + 1$. In that case x_n is overriden by x_{n+1} and then GF consumes x_{n+1}.

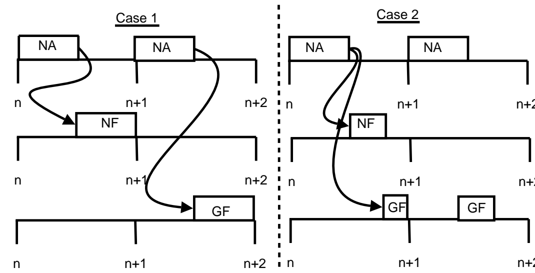

FIG. 1 – *Example of non deterministic configuration*

Consequently, if on-line preemptive scheduling policies are needed (for performance reasons for instance), it is the necessary to verify determinism. The aim of this article is to answer the question *is a scheduling deterministic for a particular multi-periodic tasks model and a given policy?* The result is that the determinism problem is decidable even in case of preemptive on-line scheduling policies. As stated previously, we consider multi-periodic tasks. We propose a formal definition of functionally deterministic scheduling, and we prove the decidability result. For that purpose, we encode the problem with a synchronized product of stopwatch automata and a particular extended timed automaton. The response can then be checked, in a decidable way, by a reachability analysis.

In the rest of the paper, we only consider single-processor systems.

2 Functional Determinism in Real-Time Scheduling

Real-time scheduling theory provides a formal framework for checking the schedulability of a tasks configuration and finding feasible, as well as optimal, scheduling. The aim of this section is to give a brief overview of this framework, and afterwards to introduce the notion of functional determinism.

2.1 Recall on Real-Time

Task Model. A task is an executable program implementing one and only one functional module. A task may be periodic or sporadic. In most cases, especially in the context of critical systems, tasks are supposed to be periodic. In the following, we only consider periodic tasks.

According to the Liu-Layland model (Liu and Layland, 1973), periodic task may be characterized by static parameters (T, r, D, B, W) where T is the task period, r is the release date (first activation), D is the (relative) deadline, B and W are the best and worst execution time (BCET and WCET). B and W depends on multiple elements: the processor, the compiler, the memories... Estimation of these parameters is a wide research area which is considered out of the scope of the paper.

Communication Model. Tasks are supposed to communicate by producing and consuming data in an asynchronous way: emissions and receptions are non blocking, emitter writes in a non blocking memory and receiver always consumes the last emitted value. Such a model is obviously non blocking and is often suitable for timed critical systems. Moreover, in order to

ensure determinism, we assume tasks always read input data at the beginning of their execution and produce output data at the end. We assume also that these input and output part can not be suspended, and that their execution time is negligible.

Real-Time Constraints. Execution of real-time tasks must satisfy three type of constraints. *Timing constraints* enforce each task instance to complete its execution before D after the date the task is released (D is a relative deadline); *precedence constraints* force partially task instance order and the read of current data values; *resource constraints* represent the exclusive access to shared resources. The scheduling problem is then to decide if a set of tasks can be executed on a given architecture while satisfying these three types of constraints.

Scheduling Policies. A scheduling strategy consists in organizing the execution of a tasks set under constraints. Usually, scheduling strategies are classified as preemptive versus non-preemptive, and off-line versus on-line policies. In non-preemptive case, each task instance, when started, completes its execution without interruptions. Conversely, in preemptive case, the scheduling unit can suspend a running task instance if a higher priority task asks for the processor. Off-line scheduling is based on a schedule which is computed before run-time and stored in a table executed by a dispatcher. One of the most popular off-line scheduling strategy is cyclic executive approach. With this method, tasks are executed in a predefined order, stored in a cyclic frame whose length is the least common multiple of the tasks periods. Each task can then be executed several times in the frame according to its period. Conversely, the idea of on-line scheduling is that scheduling decisions are taken at run-time whenever a running task instance terminates or a new task instance asks for the processor.

The three most popular on-line scheduling strategies are Rate Monotonic (RM), Deadline Monotonic (DM) and Earliest Deadline First (EDF) (Liu and Layland, 1973; J. Leung, 1980; Leung and Whitehead, 1982). RM is an on-line preemptive static priority scheduling strategy for periodic and independent tasks assuming that $T_t = D_t$ (period equals deadline) for each task t. The idea is to determine fixed priorities by task frequencies: tasks with higher rates (shorter periods) are assigned higher priority. DM is a generalization of RM with tasks such that $T_t \neq D_t$. In that case, tasks with shorter deadlines are assigned higher priority. EDF is a more powerful strategy. It is an on-line preemptive dynamic priority scheduling approach for periodic or aperiodic tasks. The idea is that, at any instant, the priority of a given task instance waiting for the processor depends on the time left until its deadline expires. Lower is this time, higher is the priority.

2.2 Towards a definition of deterministic scheduling

Definition 1 (Functionally Deterministic Scheduling) *Let $\mathcal{T} = \{t_i = (T_i, r_i, D_i, B_i, W_i)$, $i = 1 \dots n\}$ be a finite set of tasks and P a scheduling policy (EDF, RM $\dots$) such that $\mathcal{T}$ is schedulable. We say that the scheduling is functionally deterministic if for all possible executions, the output values are the same.*

A sufficient condition for determinism is that whenever two tasks t and t' are such that t produces data to t', then either t always completes before t', or t never completes before t'.

As said before, off-line non preemptive scheduling is always deterministic. This is not the case for the other scheduling techniques. As it was done for the schedulability property, the question is then: for a given tasks configuration and for a given on-line scheduling policy (pre-

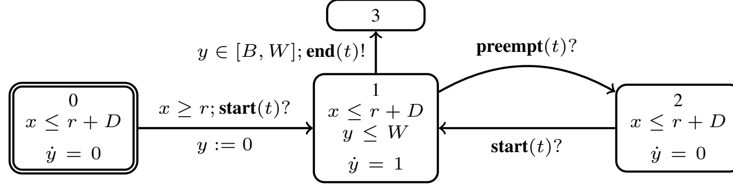

FIG. 2 – *Stopwatch Automaton Associated to a Task* t

emptive or non preemptive), is it possible to prove that the system is functionally deterministic? Answering this question is the purpose of the following section.

3 Decidability Result

The aim of this section is to prove that the question *is a scheduling functionally deterministic for a given policy* is decidable and is equivalent to a reachability analysis in a acyclic stopwatch automaton. For that purpose, we firstly show that a real-time task can be formalized with a stopwatch automaton (Alur et al., 1992) and a scheduler can be represented with a timed automaton extended with a data structure. We then prove that the synchronized product of these automata is loop free. We finally express the decision *is a scheduling functionally deterministic for a policy* by a temporal formula, and we show that checking this formula is equivalent to a reachability problem on acyclic stopwatch automata. Since this last question is decidable on loop less stopwatch automata, we conclude that the determinism question of a tasks configuration under a scheduling policy is decidable.

Let us note $\mathcal{T}$ a finite set of tasks. The first step of the decidability proof consists in encoding the tasks and a scheduler with respectively stopwatch and timed automata.

Task Unrolling on the Maximal Period. Since the tasks are periodic without direct dependencies and are under a conservative scheduling policy, it is sufficient to study their behavior in the feasibility interval $[0, \max_i(r_i) + 2LCM_i(T_i)]$, where $\max_i(r_i)$ is the the maximal release date and $LCM_i(T_i)$ is the least common multiple of all the task period (J. Leung, 1980; Pailler, 2006). Let us note $F = \max_i(r_i) + 2LCM_i(T_i)$. For each task $t = (T, r, D, B, W)$, we unroll it into n aperiodic tasks t_i, appearing only one time in the feasibility interval, such that n is the greatest integer satisfying $n \times T \leq F$ and for all $i \in [0, n-1]$, $t_i = (\infty, r+T \times i, D, B, W)$ (we note period of t_i equals ∞ to mean that t_i is an aperiodic unrolled task). We note $U^{-1}(t_i) = t$ to express that t_i is an unrolled instance of t. We note $\mathcal{T}^*$ the set of unrolled tasks of $\mathcal{T}$.

Task Real-Time Behavior. Let us consider an unrolled task $t = (\infty, r, D, B, W)$. Then, its behavior is depicted in the automaton given in figure 2. The task begins by waiting in state 0 for a start command, and then, enters the running state (state 1). In case of preemption by the scheduler, its enters state 2 waiting for a new start command, until it completes its execution (state 3). Stopwatch y denotes the execution time (suspended in state 2), and x denotes the real-time. The task must complete before x reaches the deadline D, and can complete when y is between the task BCET B and WCET W parameters.

Scheduler Automaton. The scheduler is encoded by a timed automaton extended with a list of set of tasks. This encoding generalizes the results of Fersman et al. (2007) for the

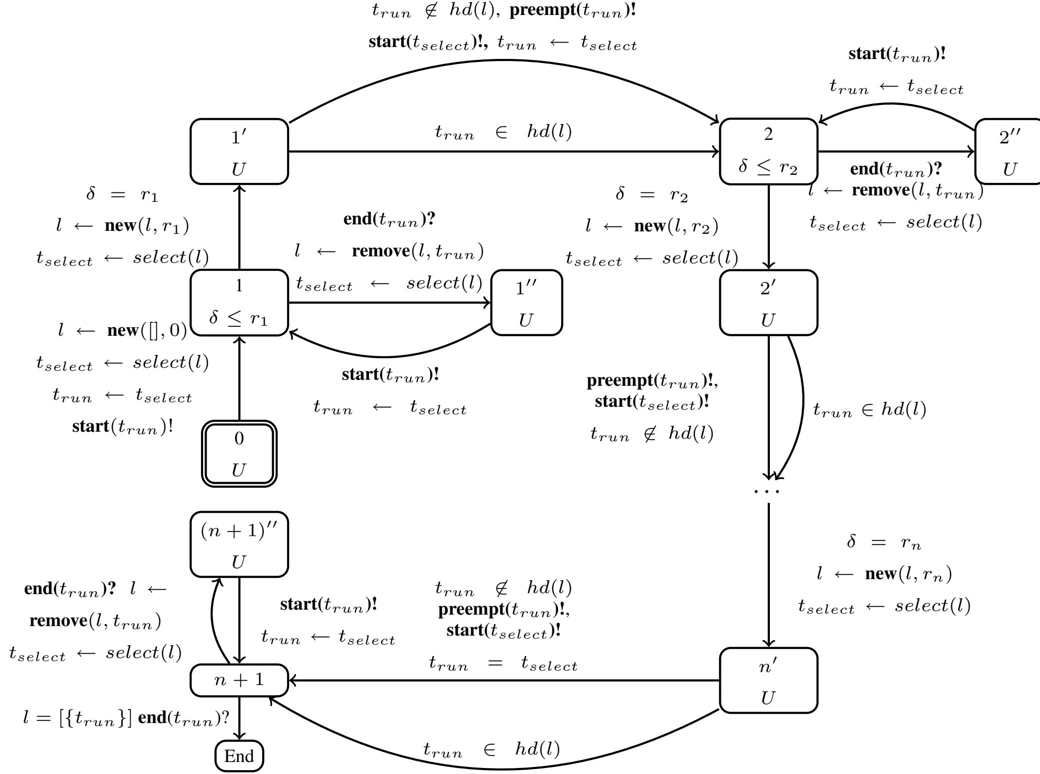

FIG. 3 – *Stopwatch automaton associated to a scheduler for n unrolled tasks*

extended Liu-Layland model. The list l managed by the scheduler automaton is ordered by a priority relation $\sqsubset$ between tasks depending on the scheduling policy. More formally, the list l is either the empty list [] if no task is asking for the processor, or a list $l = [\{t_1^1, \dots, t_{n_1}^1\}, \dots, \{t_1^p, \dots, t_{n_p}^p\}]$ meaning that: (1) tasks $t_1^1 \dots t_{n_p}^p$ are waiting for the processor (their release date are elapsed), (2) for all $i \leq p$, for all $k, k' \in 1 \dots n_i$, tasks t_k^i and $t_{k'}^i$ have the same priority, and (3) for all $i, j \leq p$ such that $i < j$, for all $k \in 1 \dots n_i$ and $k' \in 1 \dots n_j$, then $t_k^i \sqsubset t_{k'}^j$, i.e., task t_k^i has a higher priority than $t_{k'}^j$.

The order relation encodes the scheduling policy. For Earliest Deadline First scheduling policy, tasks are ordering according to the time left until their deadline expires. Deadlines are then relative. Consider then two (unrolled) tasks $t = (\infty, r, D, B, W)$ and $t' = (\infty, r', D', B', W')$ in $\mathcal{T}^*$. Then $t \sqsubset t'$ at time θ iff $r + D - \theta < r' + D' - \theta$. This definition does not depend on the date θ. It means that order priority is fixed for unrolled tasks. But, two unrolled tasks t and t' of a same "rolled" task from $\mathcal{T}$ may have different priorities.

Let us define $L_{\mathcal{T}^*,\sqsubset}$, in short L, the set of lists l of set of tasks from $\mathcal{T}^*$ and sorted by a order relation $\sqsubset$. We introduce four functions operating on list l: (a) **new:**$L \times \mathbb{R}_{\geq 0} \mapsto L$, (b)

select:$L \mapsto \mathcal{T}^*$, (c) **remove:**$L \times \mathcal{T}^* \mapsto L$, and (d) **hd:**$L \mapsto \mathcal{T}^*$. **new**$(l, r)$ returns an ordered list by inserting in l tasks whose release date is r. **select**(l) returns one task in l with the highest priority. **select** can be non deterministic whenever several tasks have the same highest priority. **remove**(l, t) returns $l' = l \backslash t$. If after removing t the head of the resulting list is the empty set, then this set is also removed from l. And finally **hd**(l) returns the set of tasks in l with the highest priority.

We can now build the timed automaton encoding the behavior of the scheduler for one hyperperiod. This automaton is depicted figure 3. It is organized as a chain of $n + 1$ states where n is the number of different release dates of tasks in $\mathcal{T}^*$. The scheduler enters the initial state (state 0) and immediately (a) creates the list l of tasks whose release date is $r_0 = 0$ and (b) starts one of the most priority task. It waits then in state 1 until the global time δ reaches the release date r_1. Whenever the running task ends, it is removed from l, and another priority task is started (note that states tagged by letter U are instantaneous, i.e. no time can elapse). When the time reaches the second release date r_1, the scheduler computes the new list l by adding and sorting tasks whose release date is r_1, and it enters the instantaneous state $1'$. If the running task is one of the priority one, its execution continues, else it is suspended and higher priority task is started (arriving in state 2). This behavior is repeated during the feasibility interval until all release dates are elapsed. The whole behavior of the system (tasks + scheduler) is then obtained by synchronizing the previous automata.

Flattening into an Acyclic List Free Automaton. The second step of the proof consists in showing that this whole behavior (tasks + scheduler) can be transformed in an acyclic list free automaton. The number of tasks in $\mathcal{T}^*$ is finished and each task can occur one and only once in the list l. Consequently, l can be encoded by a finite automaton. This entails that the scheduler automaton can be transformed into a finite list free timed automaton. The task automaton depicted figure 2 contains a loop taken whenever event *preempt* is emitted by the scheduler. A task can only be preempted a finite number of times, at most n times (n is the number of different release dates). Consequently, the synchronized product of the tasks and the scheduler automata can be unrolled into a loop free stopwatch automaton.

Determinism as a Reachability Problem. Finally the deterministic property is encoded by a reachability formula. Following definition 1, a scheduled system is functionally deterministic if whenever reads and writes occur, the data have always the same value. A necessary condition for that purpose is the communicating tasks execution's order is always the same. In other words, let be t_1 and t_2 two tasks in $\mathcal{T}^*$ such that t_1 produced data to t_2. Then, the scheduled system is considered as deterministic if either t_1 always ends its execution before t_2 begins, or t_2 always begins before t_1 completes. Formally, the system is deterministic with respect to t_1 and t_2 if the following formula *reach*$(\ldots, 3, \ldots, 0, \ldots) \Rightarrow$ *not reach*$(\ldots, s, \ldots, 1, \ldots)$ is satisfied where $s \in \{0, 1, 2\}$ and *reach*$(\ldots, i, \ldots, j, \ldots)$ means that the global state including local state i for t_1 and local state j for t_2 can be reached. This formula stands for: if t_1 can reach state 3 (execution is completed) while t_2 is still in state 0 (execution is not yet begun), then it is impossible to reach a state where t_2 is running while t_1 is not completed. Such a formula can be checked by 4 reachability analyses. The reachability analysis being decidable upon loop free stopwatch automata, as consequence of Adélaïde and Roux (2002), verification of the formula above is then decidable.

4 Conclusion and Perspectives

A long discussion has pointed some implementation solutions of a software to deal with functional determinism. With the increasing complexity in the hardware and in the functionalities, these solutions may become insufficient or difficult to manage. This is the reason why we propose a first step in the study of preemptive (on-line) scheduling for high critical systems. In this paper, we have proved that deciding if a scheduling respects functional determinism is decidable. This opening is insufficient since we do not explain how a non deterministic scheduling can be modified to become deterministic. For a RM policy, any communicating tasks at the same period entail non determinism. We have implemented the algorithm with the tool HYTECH but, because of its complexity, it is perfectly illusive to use this method. We are currently developing a new methodology and evaluate the concrete complexity. This discussion will be the next step of the reflexion.

References

Adélaïde, M. and O. Roux (2002). A class of decidable parametric hybrid systems. In *AMAST'02*, pp. 132–146.

Alur, R., C. Courcoubetis, T. A. Henzinger, and P.-H. Ho (1992). Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, pp. 209–229.

Ermont, J. and F. Boniol (2007). Verification of embbeded systems with preemption: a negative result. In *ISOLA'07*.

Fersman, E., P. Krcal, P. Pettersson, and W. Yi (2007). Task automata: Schedulability, decidability and undecidability. *International Journal of Information and Computation*.

J. Leung, M. M. (1980). A note on preemptive scheduling of periodic real-time tasks. *Information Processing Letters 11*(3), 115–118.

Leung, J. Y.-T. and J. Whitehead (1982). On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation 2*(2), 237–250.

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

Pailler, S. (2006). *Analyse hors ligne d'ordonnançabilité d'applications temps réel comportant des tâches conditionnelles et sporadiques*. Ph. D. thesis, Université de Poitiers.

S. Tripakis, C. Sofronis, N. S. and P. Caspi (2005). Semantics-preserving and memory-efficient implementation of inter-task communication under static-priority or edf schedulers. In *5th ACM Intl. Conf. on Embedded Software (EMSOFT'05)*.

Scaife, N. and P. Caspi (2004). Integrating model-based design and preemptive scheduling in mixed time- and event-triggered systems. In *Euromicro Conference Real-Time Systems, ECRTS04*.