The Journal of Logic and Algebraic Programming 77 (2008) 1–19

Contents lists available at ScienceDirect

The Journal of Logic and Algebraic Programming

journal homepage: www.elsevier.com/locate/jlap

Models and formal verification of multiprocessor system-on-chips

Aske Brekling[*], Michael R. Hansen, Jan Madsen

Informatics and Mathematical Modelling, Technical University of Denmark, DK-2800 Lyngby, Denmark

ARTICLE INFO

Available online 18 July 2008

Keywords:
Multiprocessor system-on-chips
Schedulability
Timed automata
UPPAAL
Verification

ABSTRACT

In this article we develop a model for applications running on multiprocessor platforms. An application is modelled by task graphs and a multiprocessor system is modelled by a number of processing elements, each capable of executing tasks according to a given scheduling discipline. We present a discrete model of computation for such systems and characterize the size of the computation tree it suffices to consider when checking for schedulability.

Analysis of multiprocessor system on chips is a major challenge due to the freedom of interrelated choices concerning the application level, the configuration of the execution platform and the mapping of the application onto this platform. The computational model provides a basis for formal analysis of systems.

The model is translated to timed automata and a tool for system verification and simulation has been developed using UPPAAL as backend. We present experimental results on rather small systems with high complexity, primarily due to differences between best-case and worst-case execution times. Considering worst-case execution times only, the system becomes deterministic and using a special version of UPPAAL, where the no history is saved, we could verify a smart-phone application consisting of 103 tasks executing on four processing elements.

1. Introduction

Modern hardware systems are moving toward *execution platforms* made up of multiple programmable and dedicated *processing elements* implemented on a single chip, known as multiprocessor system-on-chip (MPSoC). The different parts of an embedded *application* are executing on these processing elements; but the activity of mapping the parts of an embedded program onto the platform elements is non-trivial. First of all, there may be various and often conflicting resource constraints. Real-time constraint, for example, should be met together with constraints on the use of memory and energy. There also are a huge variety in freedom of choices in the mapping of an application to a platform because there are many ways to partition an embedded program into parts, there are many ways these parts can be assigned to processing elements, and each processing element can be set up in many ways.

In this work we aim at models and tools for analysis of problems that must be considered when an application is mapped to an execution platform. Such models are called *system models* [25] as they comprise a model for the application executing on the platform, and the analysis of such systems is also called *cross-layer analysis* as it deals with problems where decisions concerning one layer of abstraction, e.g. concerning the scheduling principle used in a processing element, has influence on the properties at another level of abstraction, e.g. a task is missing a deadline.

To illustrate the problems we shall address, consider the simple example in Fig. 1, where the application is specified by five cyclic tasks $\tau_1,\ldots,\tau_5$ and three processing elements pe_1,pe_2,pe_3. There are causal dependencies between tasks, and τ_1,

[*] Corresponding author.
*E-mail address:*awb@imm.dtu.dk (A. Brekling).

1567-8326/$ - see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.jlap.2008.05.002

task	execution time ($bcet_\tau, wcet_\tau$)	processor
τ_1	(1,2)	pe_1
τ_2	1	pe_2
τ_3	1	pe_3
τ_4	1	pe_2
τ_5	1	pe_3

Casual dependencies:
$\tau_1 \prec \tau_2$ and $\tau_3 \prec \tau_4 \prec \tau_5$

Scheduling information:
pe_2 : fixed-priority scheduling
τ_2 has highest priority

Fig. 1. Example: best-case execution time gives slower run.

for example, must finish before τ_2 can start (written $\tau_1 \prec \tau_2$). We want to find the shortest period where all tasks meet their deadlines and analyze two runs corresponding to the two possible execution times for τ_1 in Fig. 2. In both runs, τ_1 and τ_3 are executing on pe_1 and pe_3, respectively, in the first time step, where no task is executing on pe_2 due to the causal dependencies. The later time steps have similar explanations. Observe that the shortest possible period is $\pi = 4$ corresponding to the case where the best-case execution time $bcet_{\tau_1} = 1$ is chosen for τ_1. Thus, an analysis based on the worst-case execution time $wcet_{\tau_1} = 2$ would, in this case, not lead to the worst-case scenario. This is an example of a *multiprocessing timing anomaly* [7] exhibiting a counterintuitive timing behavior. A locally faster execution, either by making the processor faster or by making the algorithm more efficient, may lead to an increase of the execution time of the whole system. The presence of such behavior makes multiprocessor timing analysis more difficult [27].

It is easy to check whether a period $\pi = 3$ can be achieved for this application, simply by changing the priorities so that τ_4 gets higher priority than τ_2. But the problems cannot get much bigger than the one in Fig. 1 before the consequences of design decisions cannot be comprehended, and it is necessary to have tool support for the *design space exploration* [10,24].

1.1. Related work

As indicated by the example in Fig. 1, the schedulability analysis of systems relates to the classical job-shop scheduling problem [23]. A job-shop problem is specified as follows: suppose a number of non-cyclic tasks and a number of machines (processing elements) are given. Each task τ is assigned a machine for a specified amount of time ($bcet_\tau = wcet_\tau$). A job is a sequence of tasks (e.g. $\tau_3 \prec \tau_4 \prec \tau_5$), and the job-shop optimization problem is the problem of finding a minimum-time schedule for a collection of jobs (two in the above example). This problem is known to be NP-hard. The problem we address can be considered a generalization of the job-shop problem: we consider cyclic tasks, and tasks can have an offset, best case and worst-case execution times. Furthermore, we consider sequential components of task given by directed acyclic graphs (not just a linear order), and we consider a variety of preemptive scheduling disciplines. We will, however, just consider a decision version of the problem. In [2] it is shown that finding an optimal schedule for a job-shop optimization problem corresponds to finding a shortest path in a certain kind of acyclic timed automata.

Most modelling methodologies for multiprocessor embedded systems are based on the Y-chart of system design [25]. In the Y-chart there is a clear distinction between the application and the execution platform, and there is an explicit mapping of the elements of the application onto elements of the execution platform. This mapping results in a particular system design that can be evaluated using simulation or formal analysis.

Simulation-based methodologies are still the predominant technique for performance evaluation and, for example, MPARM [17] is a SystemC-based [16] framework for multiprocessor MPSoC architectures aimed at exploring on-chip communication networks. It is a cycle-accurate model that requires detailed knowledge of the execution platform. The detailed level allows for executing compiled software, including the real-time operating systems, on the platform model. Refs. [11,22,9] propose a higher level modelling approach, by viewing the application as a set of tasks executing on a real-time operating system. As embedded software plays a dominating role in defining the functionality of an embedded system, choosing the right scheduling policy has a distinct influence on the system's rate and reactivity. All three approaches are based on SystemC and target single processor systems.

The challenges of MPSoC platforms have, for instance, been addressed in [25,10,24,13,19,21]. In [13], the use of virtual processing units (VPU) for representing processors is proposed. Each VPU supports a priority-based scheduler that models the execution of software tasks. Tasks are modelled as communicating finite state machines, where each transition is

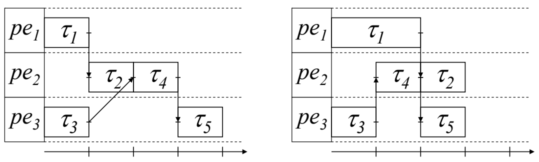


Fig. 2. (a) Execution time for τ_1 is 1. (b) Execution time for τ_1 is 2.

regarded an atomic operation taking a fixed amount of processor cycles. A similar approach is used in ARTS [20,21], where synchronization among processors and resource allocation is incorporated into the model of the real-time operating system. The designer can make simulations analyzing quantitative properties such as timing, memory and power usage, as well as communications. A tool is proposed in [24] for performance evaluation and exploration based on SystemC. The focus is multimedia applications that are modelled as Kahn process networks and the performance is evaluated using a trace-driven simulation approach.

SystemCoDesigner [10] is a SystemC-based simulation framework that supports automatic performance evaluation and design space exploration. The design space exploration in [10] is based on a multi-objective evolutionary approach. A similar approach is used for design space exploration in CHARMED [14] which focus on periodic multi-mode embedded systems.

The advantage of simulation-based tools is that the designers, early in the development process, receive insight into the feasibility of the chosen designs before the concrete implementations are considered. However, although the simulation tool can exhibit a collection of executions of a system, it can in no way be used to guarantee that every execution is meaningful. In order to handle shortcomings of simulation-based approaches, several formal approaches have been proposed. In [26], an approach for analyzing communication delays in message scheduling, together with optimization strategies for bus access, is presented. Thiele et al. [30] provide a real-time calculus for scheduling of preemptive tasks with static priorities and, in [28], a formal approach based on event flow interfacing is described. A simple tool for analysis of interfacing is provided, together with algorithms for configuring a global analysis model.

In [6], a timed-automata approach for schedulability analysis of single processor systems is provided. It appears that the computation model behind preemptive scheduling strategies is stopwatch automata, for which it is known that the reachability problem is undecidable; however, in [6] it is shown that the schedulability problem for extended timed automata is decidable for preemptive scheduling strategies. This approach is used in the Times tool [4], which considers schedulability of tasks on a single processor only. As for the Times tool, our approach is based on a timed automata model.

In [8], Uppaal is used in connection with verification and analysis of the legOS scheduler. We provide a more general model where different schedulers can be verified on multiprocessor platforms. Altisen and Tripakis propose in [31] an implementation methodology for transforming a timed automaton into a program, and through that, checking properties of the execution of that program. This is done by modelling the program as well as *the execution platform* as untimed and timed automata. However, the notion of an execution platform is basically modelled in terms of the digital clock of the platform. Many issues such as multiprocessor anomalies, which our work concerns, cannot be investigated through this methodology. In [2,1], the timed-automata model is proposed as a natural tool for posing and solving scheduling problems in general. However, the concept of an execution platform is not considered, and therefore issues such as the multiprocessor anomalies are not investigated.

Overview: In this paper we provide, in Section 2, a model for applications (in terms of task graphs), execution platforms (in terms of a number of processing elements each capable of executing tasks according to a specified scheduling discipline), and systems (in terms of a mapping of an application onto the execution platform). Furthermore, we present the computational model for systems (in Section 2.4) and a bound on the size of the computation tree (in Section 2.5) it suffices to consider in order to decide whether all tasks meet their deadlines. We present, in Section 3, a timed-automata [3] implementation of the model, and, in Section 4, a tool using Uppaal [5,15] as backend, which can be used for design space exploration. Experimental results are presented in Section 4.1. Finally, the last section contains a conclusion.

2. Model for a system-level framework

We shall now give a model for a system consisting of an application which is running on an execution platform. An execution platform is made up of multiple programmable and dedicated processing elements (*pe*) and an application can be divided up into a number of tasks (τ), each of which represents a piece of sequential code. These tasks might have timing constraints as well as causal dependencies with other tasks.

Having processing elements that can execute several different tasks and manage execution time dynamically introduces a need for a dedicated *real-time operating system* (*os*) as a layer between the application and the execution platform. The real-time operating system should manage scheduling of the different tasks as well as the dependencies tasks might have with each other.

A *system-level framework* should be recognized as a framework for the overall system comprising application, real-time operating system and processing elements. There are tools available for simulation of MPSoCs – one of these is ARTS [21], and the model in this section as well as the timed-automata model of the next section have background in concepts and features of ARTS.

Fig. 3 provides an overview of the different components of a system-level framework. A system in ARTS is a mapping of an application (made up of a number of tasks), onto an execution platform. Dashed arrows from tasks to processing elements in Fig. 3 represent this mapping, e.g. τ_1 is mapped to pe_1. The arrows in the task graph show dependencies between tasks, e.g. τ_1 and τ_2 must finish execution before τ_3 can start. Each processing element has its own real-time operating system. The communication network is modelled as a special processing element on which communication tasks are mapped (See [19] for further details). We shall now present a formal model for the execution of an application on an execution platform.

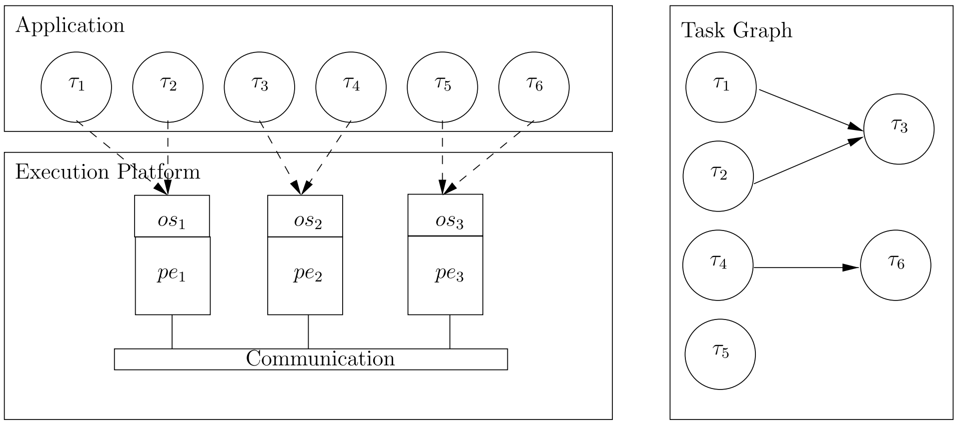


Fig. 3. System-level framework for a MPSoC.

2.1. Application – the task model

Let a finite set $\mathcal{T}$ of *tasks* be given. Each task $\tau \in \mathcal{T}$ is characterized by a *period* $\pi_\tau \in \mathbb{N}$, a *best-case execution time* $bcet_\tau \in \mathbb{N}$, and a *worst-case execution time* $wcet_\tau \in \mathbb{N}$, where $wcet_\tau \geq bcet_\tau > 0$. A task τ needs a certain amount, between $bcet_\tau$ and $wcet_\tau$, of time units of a processor's time to finish its job in a given period. (This notion is formalized later when we introduce the execution model for multiprocessor platforms.) Furthermore, a task τ is characterized by an *initial offset* $o_\tau \in \mathbb{N}$, which means that the first period of τ starts o_τ time units after the system has started. Thus, the nth period of task τ is the time interval $[o_\tau + (n-1) \cdot \pi_\tau, o_\tau + n \cdot \pi_\tau[\subset \mathbb{R}_{\geq 0}$ for $n = 1,2,\ldots$

An application is modelled by a *task graph* $G = (\mathcal{T}, \prec)$, where $\prec \subseteq \mathcal{T} \times \mathcal{T}$ is a directed, acyclic graph. An edge $(\tau,\tau') \in \prec$ (also written $\tau \prec \tau'$) represents a causal dependency, i.e. τ must finish its job before τ' can start.

A *sequential component* $G_s = (\mathcal{T}_s, \prec_s)$, where $\mathcal{T}_s \subseteq \mathcal{T}$ and $\prec_s \subseteq \mathcal{T}_s \times \mathcal{T}_s$, is a connected sub-graph of G, for which

- all tasks have the same period $\pi_{\mathcal{T}_s}$, i.e. $\pi_i = \pi_j = \pi_{\mathcal{T}_s}$ for $\tau_i,\tau_j \in \mathcal{T}_s$, and
- the offsets of tasks are so close to each other that their first (and hence the nth) period overlaps, i.e. $|o_i - o_j| < \pi_{\mathcal{T}_s}$ for $\tau_i,\tau_j \in \mathcal{T}_s$.

We shall, from now on, assume that G can be partitioned into sequential components $G_i = (\mathcal{T}_i, \prec_i)$, for $i = 1,\ldots,N$, where $\mathcal{T}_i \cap \mathcal{T}_j = \emptyset$, whenever $i \neq j$, $\mathcal{T} = \mathcal{T}_1 \cup \cdots \cup \mathcal{T}_N$ and $\prec = \prec_1 \cup \cdots \cup \prec_N$.

2.2. The model of the platform

A *platform* consists of $M \geq 1$ *processing elements* (also called *processors*) $PE = (pe_1, pe_2, \ldots, pe_M)$, where each processing element pe_i is capable of executing tasks according to a given *scheduling principle*. We will here consider the following preemptive scheduling strategies: rate-monotonic RM, fixed-priority FP and earliest-deadline-first EDF scheduling. There will be no principal difficulties in extending the model with more scheduling principles; but this will, of course, add more details.

2.3. The system model

A *system* consists of a *mapping* $m : \mathcal{T} \to \{1,\ldots,M\}$ of tasks to processing elements, and a *configuration* $Sch : PE \to \{\text{RM,FP,EDF}\}$ of the scheduling principle used by each processing element. We shall often consider the set of tasks $\mathcal{T}_{pe_i}$ mapped to a particular processing element pe_i, i.e. let

$$\mathcal{T}_{pe_i} = m^{-1}(i) = \{\tau \in \mathcal{T} \mid m(\tau) = i\}$$

2.3.1. Scheduling of tasks

We assume that each task has a unique number given by an injective function $pr : \mathcal{T} \to \{1,\ldots,\text{card}(\mathcal{T})\}$. This numbering is used to define total orderings of tasks in connection with the various scheduling principles.

A task τ has *higher fixed priority than* τ', written $\tau >_{\text{FP}} \tau'$, iff $pr(\tau) < pr(\tau')$, i.e. smaller number given by pr means higher priority.

For rate-monotonic scheduling, the priority relation among tasks is primarily given by their periods (shorter periods mean higher priority) and secondarily by their numbering according pr:

A task τ has *higher rate-monotonic priority than* τ', written $\tau >_{\text{RM}} \tau'$, iff $\pi_\tau < \pi_{\tau'} \vee (\pi_\tau = \pi_{\tau'} \wedge pr(\tau) < pr(\tau'))$.

The priority relations $>_{\text{FP}}$ and $>_{\text{RM}}$ are total orders on the set of tasks due to the unique numbering given by pr, and, therefore, there will never be a non-deterministic choice when scheduling according to the rate-monotonic and fixed-priority disciplines in the system.

The relations $>_{\text{FP}}$ and $>_{\text{RM}}$ are simple since the fixed-priority and rate-monotonic principles are static scheduling principles, i.e. the ordering relations among tasks are independent of the current point in time. For earliest deadline first this is different as, for a given time point t, the task with the highest priority is that with the shortest distance to its nearest deadline, and hence the priority relation between tasks may change when some new period starts during the execution.

Let $t \in \mathbb{N}$ and $\tau \in \mathcal{T}$. By $\text{dist}_\tau(t)$ we denote the *distance from t to the nearest deadline (i.e. the start of a new period) of* τ:

$$\text{dist}_\tau(t) = p - t \qquad \text{where} \qquad \begin{Bmatrix} p = o_\tau + n \cdot \pi_\tau \\ \text{for the smallest } n \in \mathbb{N}_+ \text{ so that } p > t \end{Bmatrix}$$

Notice that we have a simple setting where the deadline for a task is the same as the start point of its next period. The model is easily extended to cope with deadlines that are earlier than the start of the next period.

A task τ has *higher earliest-deadline-first priority than* τ' *at time* $t \in \mathbb{N}$, written $\tau >^t_{\text{EDF}} \tau'$, iff

$$dist_\tau(t) < \text{dist}_{\tau'}(t) \vee (\text{dist}_\tau(t) = \text{dist}_{\tau'}(t) \wedge pr(\tau) < pr(\tau'))$$

The priority relation $>^t_{\text{EDF}}$ is also a total order, and observe that when no task has a deadline between two time points t_1 and t_2, then $>^t_{\text{EDF}} = >^{t'}_{\text{EDF}}$, for every t,t' where $t_1 < t,t' < t_2$. Thus, the earliest-deadline-first priorities can change at time points only when some new period for a task starts.

2.4. The model of computation

To model the computations of a system, the notion of a state, which is a snapshot of the state of affairs of the individual processing elements, is introduced. Consider a processing element pe_i. For that processing element the state component must record which task in $\mathcal{T}_{pe_i}$ is currently executing, where we denote by $\perp$ that no task is currently executing on pe_i. Furthermore, for every task $\tau \in \mathcal{T}_{pe_i}$, the state component also records the execution time $\epsilon_i(\tau) \in \{bcet_\tau, bcet_\tau + 1, \ldots, wcet_\tau - 1, wcet_\tau\}$ that is needed by τ to finish its job in the current period. We call ϵ_i an *execution vector* for pe_i. Each time a new period for τ starts, a non-deterministic choice is taken concerning the execution time for τ in that period. Thus, a *state component* for pe_i is a tuple (s_i, ϵ_i), where $s_i \in \mathcal{T}_{pe_i} \cup \{\perp\}$ and ϵ_i is an execution vector for pe_i.

A *system state* or just a *state* is an M-tuple $\sigma = ((s_1,\epsilon_1),(s_2,\epsilon_2),\ldots,(s_M,\epsilon_M))$, where (s_i,ϵ_i), is a state component for pe_i, for $i \in \{1,\ldots,M\}$.

A *trace* (or *history*) is a finite sequence of states:

$$\hbar = \sigma_1\sigma_1 \cdots \sigma_k$$

where $k \geq 0$ is the length of $\hbar$, denoted by $\text{length}(\hbar)$. A trace with length k describes a system behavior in the interval $[0,k[$.

Consider a task $\tau \in \mathcal{T}$. The *completed periods of* τ *in a trace* $\hbar$ is the set

$$\text{Completed}_\hbar(\tau) \;=\; \{n \in \mathbb{N}_+ \mid o_\tau + n \cdot \pi_\tau \leq \text{length}(\hbar)\}$$

and the *current period number* of τ in $\hbar$ is

$$\text{cpn}_\hbar(\tau) = \begin{cases} 0 & \text{if length}(\hbar) < o_\tau \\ 1 & \text{if } o_\tau \leq \text{length}(\hbar) < o_\tau + \pi_\tau \\ 1 + \max \text{Completed}_\hbar(\tau) & \text{otherwise} \end{cases}$$

The nth *period*, $n \geq 1$, of τ in $\hbar = \sigma_1\sigma_1 \cdots \sigma_k$ is the (possibly empty) sub-sequence of $\hbar$:

$$\hbar(\tau,n) = \sigma_{\alpha+1} \cdots \sigma_{\alpha+\pi_\tau}, \text{ where } \alpha = o_\tau + (n-1) \cdot \pi_\tau.$$

Notice that $\hbar(\tau,n)$ consists of π_τ states just when $n \in \text{Completed}_\hbar(\tau)$ and fewer (possibly no) states otherwise.

The *execution time* $\text{exec}_\sigma(\tau)$ of a task $\tau \in \mathcal{T}_{pe_i}$, in a state σ is

$$\text{exec}_\sigma(\tau) = \begin{cases} 1 & \text{if } s_i = \tau \\ 0 & \text{otherwise} \end{cases}$$

where $\sigma = ((s_1,\epsilon_1),\ldots,(s_i,\epsilon_i)\ldots,(s_M,\epsilon_M))$.

This notion extends to a finite sequence of states as follows:

$$\text{exec}_{\sigma_1\cdots\sigma_j}(\tau) = \sum_{m=1}^{j} \text{exec}_{\sigma_j}(\tau)$$

We denote by $\text{Finished}_\hbar(\tau,n) \in \text{Bool}$ whether $\tau \in \mathcal{T}_{pe_i}$ has finished its job in the nth period, $n \geq 1$, in $\hbar$:

$$\text{Finished}_\hbar(\tau,n) = \begin{cases} \text{true} & \text{if exec}_{\hbar(\tau,n)}(\tau) = \epsilon_i(\tau) \\ \text{false} & \text{otherwise} \end{cases}$$

where ϵ_i is the execution vector for processing element pe_i in the last system state of $\hbar$, or the *zero-execution vector* ϵ_i^0, where $\epsilon_i^0(\tau) = 0$, for $\tau \in \mathcal{T}_{pe_i}$, if the trace $\hbar$ is empty.

We shall now define the set of possible successor components states for a processing element pe_i given a trace $\hbar$ with length k. Let ϵ_i be the execution vector for pe_i in the last state in $\hbar$ or the zero-execution vector if the trace is empty. For each $\tau \in \mathcal{T}_{pe_i}$, there are the following choices for the execution vector ϵ' for the next state component of pe_i:

- $\epsilon'(\tau) = 0$, if $k < o_\tau$, i.e. the first period for τ has not started yet,
- $\epsilon'(\tau) \in \{bcet_\tau, bcet_\tau + 1, \ldots, wcet_\tau - 1, wcet_\tau\}$, if $\pi_\tau \mid k - o_\tau$ (π_τ divides $k - o_\tau$), i.e. if a new period for τ starts at time k, then one of the possible execution times for τ is chosen, and
- $\epsilon'(\tau) = \epsilon(\tau)$, if $k \geq o_\tau$ and $\pi_\tau \not| k - o_\tau$.

Let $EV_\hbar(i)$ be the (finite) set of all possible execution vectors for the next state component of pe_i given $\hbar$.

For a given $\epsilon \in EV_\hbar(i)$, a task $\tau \in \mathcal{T}_{pe_i}$ is *enabled given $\hbar$ and ϵ*, if

- $\epsilon(\tau) > 0$, i.e. $n = \mathrm{cpn}_\hbar(\tau) > 0$,
- $\mathrm{Finished}_\hbar(\tau,n) = \mathrm{false}$, i.e. τ is not yet finished in the current period, and
- Every task τ' on which τ depends, i.e. $\tau' \prec \tau$, is finished in the nth period, i.e. $\mathrm{Finished}_\hbar(\tau',n) = \mathrm{true}$.

Let $Enable_\hbar(\epsilon,i) \subseteq \mathcal{T}_{pe_i}$ be the set of enabled tasks on pe_i given $\hbar$ and ϵ.

The enabled task (if any) with the highest priority is *selected* to execute on the processing element:

$$Select_\hbar(\epsilon,i) = \begin{cases} \perp & \text{if } Enable_\hbar(\epsilon,i) = \emptyset \\ \tau & \text{if } \tau \text{ is the biggest element in } Enable_\hbar(\epsilon,i) \text{ wrt. } >^k_{Sch(pe_i)} \end{cases}$$

where $k = \mathrm{length}(\hbar)$ and the priority relations $>_{\mathrm{FP}}$ and $>_{\mathrm{RM}}$ are extended to the time domain in the trivial way: $>^t_{\mathrm{FP}} = >_{\mathrm{FP}}$ and $>^t_{\mathrm{RM}} = >_{\mathrm{RM}}$, for $t \in \mathbb{N}$.

The set of *possible next component states for pe_i* is defined by

$$Next_\hbar(i) = \{ (s,\epsilon) \mid \epsilon \in EV_\hbar(i) \text{ and } s = Select_\hbar(\epsilon,i) \}$$

and the set of *possible next system states* is defined by

$$Next_\hbar = \{ ((s_1,\epsilon_1),\ldots,(s_n,\epsilon_n)) \mid (s_i,\epsilon_i) \in Next_\hbar(i), \text{ for } 1 \leq i \leq M \}$$

The *computation tree for a system* is a finitely branching, infinite tree, where the root is the empty trace $\langle\rangle$ and the internal nodes are (labelled by) system states. Furthermore,

- there is an edge from the root to a distinct node for each system state in $Next_{\langle\rangle}$, and
- for every internal node *node* in the tree, let $\hbar_{node}$ be a trace of system states leading from the root to *node*. For every system state $\sigma \in Next_{\hbar_{node}}$ there is an edge from *node* to a distinct node labelled by σ.

A *run of the system* is an infinite sequence of states

$$\rho = \sigma_1\, \sigma_2\, \sigma_3\, \cdots$$

occurring on some path starting from the root of the computation tree.

Notice that when the worst and best case execution times are equal for every task in the system, then there is exactly one run of the system, as the scheduling is deterministic.

We call a system *schedulable* if for every run, each task finishes its job in all its periods. We shall now show that this problem is decidable, and in the next section we shall use Uppaal to solve it.

2.5. Decidability

We now consider the problem of determining schedulability of a system and we will give an upper bound on the size of the part of the computation tree it suffices to consider when checking for schedulability.

We first establish a periodic behavior of the priority relations among tasks. The two relations $>^t_{\mathrm{FP}}$ and $>^t_{\mathrm{RM}}$ for the static scheduling principles are trivial as they are, in fact, static. We just need to consider the priorities with regard to the earliest-deadline-first principle.

To this end, let a *situation at time t*, sit_t, $t \in \mathbb{N}$, be defined by the k-tuple:

$$sit_t = (\mathrm{dist}_{\tau_1}(t), \mathrm{dist}_{\tau_2}(t), \ldots, \mathrm{dist}_{\tau_k}(t))$$

where $k = \mathrm{card}(\mathcal{T})$ and the numbering is given by pr. For a given time t, the earliest-deadline-first priorities can easily be extracted from sit_t (it would actually suffice just to consider situations for tasks that are mapped to processing elements with an earliest-deadline-first discipline).

Consider an arbitrary task τ. It is easy to see that dist_τ satisfies the following properties:

$$\mathrm{dist}_\tau(t) = 1 \iff \mathrm{dist}_\tau(t+1) = \pi_\tau \tag{1}$$

$$\mathrm{dist}_\tau(t) > 1 \iff dist_\tau(t+1) = \mathrm{dist}_\tau(t) - 1 \tag{2}$$

Therefore, for every $t,c \in \mathbb{N}$:

task	execution time $wcet_\tau = bcet_\tau$	period π_τ	offset o_τ
τ_1	1	3	0
τ_2	1	3	1
τ_3	2	3	2

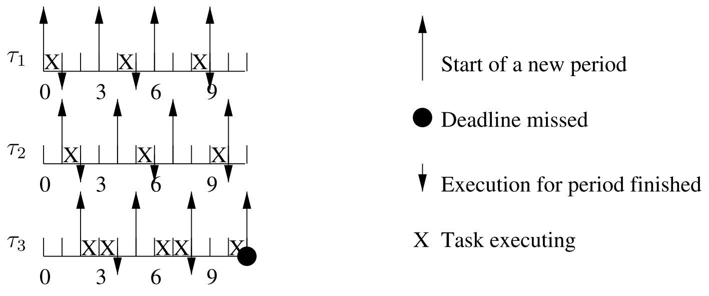


Fig. 4. Example: deadline missed at time $O_M + 3 \cdot \Pi_H$, where $O_M = 2, \Pi_H = 3$.

$$\mathrm{dist}_\tau(o_\tau + t) = \mathrm{dist}_\tau(o_\tau + c \cdot \pi_\tau + t) \tag{3}$$

Hence, when the time of the offset for τ has passed, dist_τ becomes periodic with period π_τ and, hence, any multiplum of π_τ is a period as well.

This generalizes to situations in the following way. Let $O_M = \max\{o_{\tau_1}, \ldots, o_{\tau_k}\}$ be the maximal offset in the system, and $\Pi_H = \mathrm{LCM}\{\pi_{\tau_1}, \ldots, \pi_{\tau_k}\}$ be the *hyper-period* for the tasks, i.e. the least common multiple of all periods of tasks in the system. Since the period of any task in the system is a divisor of the hyper-period, we have, using (3), the following periodic properties:

$$\mathrm{dist}_\tau(O_M + t) = \mathrm{dist}_\tau(O_M + c \cdot \Pi_H + t) \tag{4}$$

$$sit_{O_M+t} = sit_{O_M + c\cdot\Pi_H + t} \tag{5}$$

for every $t, c \in \mathbb{N}$.

Hence, the infinite sequence of situations

$$Sit = sit_1\, sit_2\, sit_3\, \ldots$$

has, using (5), the following form:

$$Sit = sit_1\, sit_2\, sit_3 \ldots sit_{O_M} (sit_{O_M+1}\, sit_{O_M+2} \ldots sit_{O_M+\Pi_H})^\omega \tag{6}$$

Therefore, for any time $t \geq O_M + \Pi_H$, we meet situations (and scheduling priorities) that we have seen earlier. This does not mean, however, that it suffices to consider the computation tree to a depth $O_M + \Pi_H$. The problem is that the execution times may not have "stabilized" yet at time $O_M + \Pi_H$ as the example in Fig. 4 shows. Even though the utilization $U = 1/3 + 1/3 + 2/3 = 4/3 > 1$ and the system obviously is not schedulable, the first deadline is missed three hyper-periods after the maximal offset.

Consider a run $\rho = \sigma_1\, \sigma_2\, \sigma_3 \cdots$ where, for every task, the same execution time is chosen throughout the run, i.e. for $\tau \in \mathcal{T}_{pe_i}$ and $j, k \geq o_\tau$ we have that $\epsilon_i(\tau) = \epsilon'_i(\tau)$, where $\sigma_j = ((s_1, \epsilon_1), \ldots, (s_i, \epsilon_i), \ldots, (s_M, \epsilon_M))$ and $\sigma_k = ((s'_1, \epsilon'_1), \ldots, (s'_i, \epsilon'_i), \ldots, (s'_M, \epsilon'_M))$. For a given $t \in \mathbb{N}$, let ρ_t be the prefix of ρ up to time t and $\mathrm{exec}_\rho(\tau, t)$ be the execution time of τ in the current period up to time t in ρ_t, i.e.

$$\rho_t = \sigma_1\, \sigma_2 \cdots \sigma_t \quad \text{and} \quad \mathrm{exec}_\rho(\tau, t) = \mathrm{exec}_{\overline{\sigma}}(\tau)$$

where $\overline{\sigma} = \rho_t(\tau, \mathrm{cpn}(\rho_t, \tau))$.

Consider a time $t \geq o_\tau$. At time t, some tasks may not have started yet and, therefore, τ may have been granted more executing time in its current period at time t than one hyper-period later at time $t + \Pi_H$, i.e.

$$\mathrm{exec}_\rho(\tau, t) \geq \mathrm{exec}_\rho(\tau, t + \Pi_H) \geq 0 \tag{7}$$

where we exploit that some task is executing on a processing element whenever there is an enabled task on that element – execution time is not wasted.

When a time $t_p \geq O_M$ is reached where for every task $t \in \mathcal{T}$:

$$\mathrm{exec}_\rho(\tau, t_p) = \mathrm{exec}_\rho(\tau, t_p + \Pi_H)$$

then ρ is periodic from time t_p, i.e.

$$\rho = \rho_{t_p}\,(\sigma_{t_p+1}\,\sigma_{t_p+2}\,\ldots\sigma_{t_p+\Pi_H})^{\omega}$$

An upper bound on t_p is

$$O_M + \Pi_H \cdot (1 + \Sigma_{\tau\in\mathcal{T}} wcet_\tau) \tag{8}$$

since, by (7), the worst-case scenario is when one task only gets its execution time decreased (by one) when one hyper-period has passed. This upper bound (8) can be improved since $\text{exec}_\rho(\tau, O_M) = 0$ for every task τ that starts a new period at time O_M. These tasks satisfy the property: $\pi_\tau \mid (O_M - o_\tau)$. Thus, in order to search the computation checking for schedulability, it suffices to search to the depth: $O_M + \Pi_H \cdot (1 + \Sigma_{\tau\in\mathcal{T}_X} wcet_\tau)$, where $\mathcal{T}_X = \{\tau \in \mathcal{T} \mid \pi_\tau \nmid (O_M - o_\tau)\}$. A missed deadline that occurs deeper in the computation tree will also occur at a depth closer to the root, but possibly on another path.

Let us examine the number of nodes at the following two depths $O_M + \Pi_H$ and $O_M + \Pi_H \cdot (1 + \Sigma_{\tau\in\mathcal{T}_X} wcet_\tau)$ in the computation tree. These numbers are the minimum and maximum number of paths in the tree that need examination in order to check for schedulability. For a given task τ, we let

- $periodicChoices_\tau$ denote the number of non-deterministic choices for execution time in each period for τ, i.e.
  $$periodicChoices_\tau = wcet_{\tau_i} - bcet_{\tau_i} + 1$$
- $initialChoices_\tau$ denote total number of non-deterministic choices for evaluation time of τ until O_M, i.e.
  $$initialChoices_\tau = \lceil (O_M - o_\tau)/\pi_\tau \rceil \cdot periodicChoices_\tau$$
- $hyperChoices_\tau$ denote the number of non-deterministic choices for the evaluation time of τ in a hyper-period, i.e.
  $$hyperChoices_\tau = (\Pi_H/\pi_\tau) \cdot periodicChoices_\tau$$

Hence the total number of non-deterministic choices for the system in the time interval $[0, O_M + \Pi_H[$ is

$$totalChoices = \prod_{\tau\in\mathcal{T}} (initialChoices_\tau + hyperChoices_\tau)$$

which equals the number of nodes in the computation tree at depth $O_M + \Pi_H$, i.e. the minimum depth in the computation tree to check before a system can be deemed schedulable. Unschedulability may be determined earlier.

The total number of non-deterministic choices for the system in the time interval $[0, O_M + \Pi_H \cdot (1 + \Sigma_{\tau\in\mathcal{T}_X} wcet_\tau)[$, is

$$maxChoices = \prod_{\tau\in\mathcal{T}} (initialChoices_\tau + hyperChoices_\tau \cdot (1 + \Sigma_{\tau\in\mathcal{T}_X} wcet_\tau))$$

which equals the number of nodes in the computation tree at depth $O_M + \Pi_H \cdot (1 + \Sigma_{\tau\in\mathcal{T}_X} wcet_\tau)$, i.e. the depth in the computation tree it suffices to search when checking for schedulability.

A small example showing a high complexity

It is easy to give a small system with a huge number of states in the computation tree up to the depth $O_M + \Pi_H$. Consider, for example, the system with three tasks given in Fig. 5. The maximal offset and the hyper-period of this system are:

$$O_M = \max\{0,10,27\} = 27$$
$$\Pi_H = \text{LCM}\{11,8,251\} = 22088$$

and the number of non-deterministic choices for each task in the initial part, each period and each hyper-period are given by

Task	Periodic choices $periodicChoices_\tau$	Initial choices $initialChoices_\tau$	Hyper-choices $hyperChoices_\tau$
τ_1	3	9	6024
τ_2	4	12	11044
τ_3	13	0	1144

Hence the number of nodes at depth $O_M + \Pi_H = 22115$ is

$$totalChoices = (9 + 6024) \cdot (12 + 11044) \cdot (0 + 1144) \approx 7.6 \cdot 10^{10}$$

and the number of nodes at depth $O_M + \Pi_H \cdot (1 + \Sigma_{\tau\in\mathcal{T}_X} wcet_\tau) = 176731$ is

$$maxChoices = (9 + 6024 \cdot 8) \cdot (12 + 11044 \cdot 8) \cdot (0 + 1144 \cdot 8) \approx 3.9 \cdot 10^{13}$$

3. A timed-automata model for MPSoC

Having a decidability result for schedulability, the next question is to get an implementation of the decision algorithm. One way would be to construct a program directly on the basis of the computation tree and the results from Section 2.5. We

task	execution time $(bcet_\tau, wcet_\tau)$	period π_τ	offset o_τ
τ_1	(1, 3)	11	0
τ_2	(1, 4)	8	10
τ_3	(1, 13)	251	27

Fig. 5. Example: small example with a huge state space in the non-periodic part.

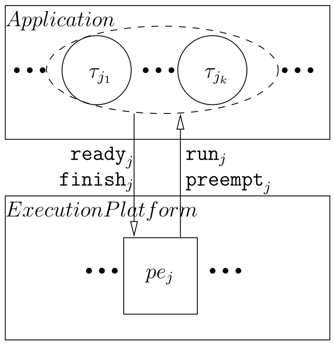


Fig. 6. Communication between execution platform and application.

will, however, take another approach as we aim at a more general framework supporting both simulation and verification of systems, and, therefore, we will build a model that comprises the components of MPSoCs and exhibits the parallel activities occurring in such systems. We will aim at a model comprising the concepts supported by the ARTS framework [20] and our approach is to develop a model of systems using a model-checking tool. Since the model of computation for systems is discrete, we could use a system like SPIN [12]. But the ARTS framework has notions that are naturally modelled by clocks and timed automata [3], and, therefore, we will use the UPPAAL [5,15] system for modelling, verification and simulation. We will not give introductions to timed automata and the UPPAAL system in this paper. For such introductions we refer to [3,5,15].

As the current version of UPPAAL does not support stopwatches, and we are dealing with preemptive scheduling (where the running time of a task can be temporarily stopped), we make the execution time in the UPPAAL model discrete. The clocks handling the periodicity and the dynamically updated scheduling criteria still operate continuously. In a SPIN model the periodicity and the dynamically updated scheduling criteria would be handled in discrete steps. Thus, it has a cost to get rid of the clocks. Whether a SPIN model would give more efficient verification a later experiment must show. At this point we have chosen to use UPPAAL as this will give a more natural model.

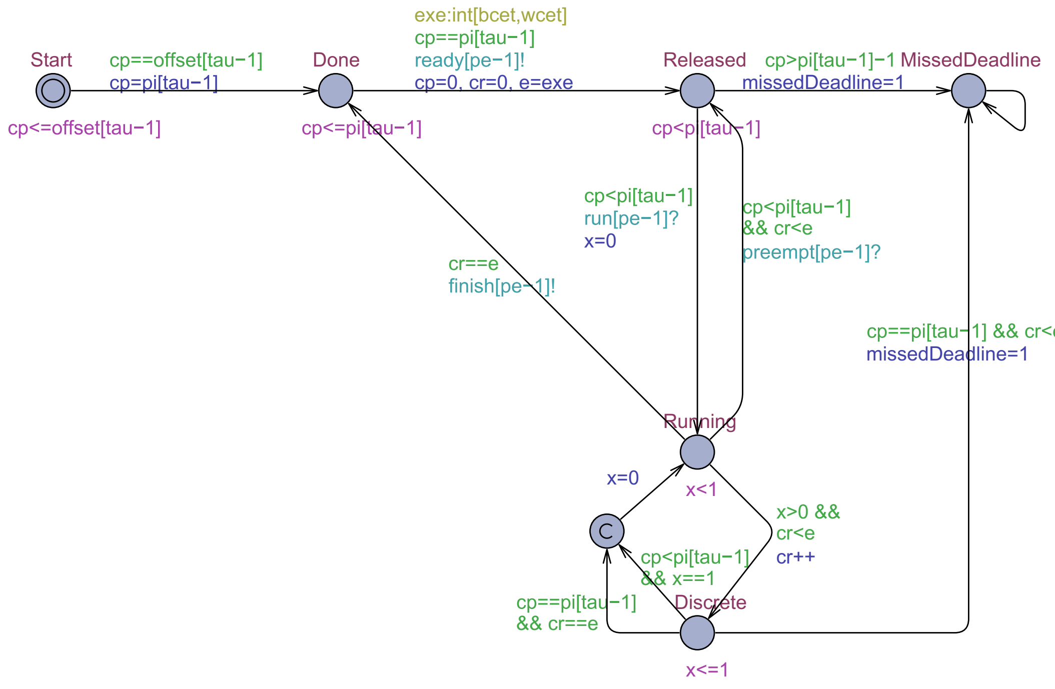
The UPPAAL model is not a direct implementation of the model of computation. However, the notions and general structure introduced are recognizable in the UPPAAL model, and we explain the UPPAAL model by relating it to the model of computation.

We will give a top-down presentation of the timed-automata model, where a system is described as a parallel composition of an application and an execution platform:

$$\begin{aligned}&\mathit{System} = \mathit{Application} \parallel \mathit{ExecutionPlatform}\\&\mathit{Application} = \parallel_{\tau\in\mathcal{T}} \mathit{TA}(\tau)\\&\mathit{ExecutionPlatform} = \parallel_{j=1}^{M} \mathit{TA}(pe_j) \parallel \mathit{DynamicPriorities}\end{aligned}$$

where $\parallel$ denotes parallel composition of timed automata. Thus, an application consists of a collection of timed automata for tasks combined in parallel and an execution platform consists of a parallel composition of timed automata for processing elements combined in parallel with a single timed automaton handling the dynamic priorities. The timed automaton *DynamicPriorities* works for all processing elements, having earliest deadline first as scheduling discipline. From a model point-of-view it would be more natural to have one such automaton for each processing element; from a verification viewpoint, the global timed automaton is preferred in order to reduce the number of clocks needed for this purpose to just one. The definitions of $\mathit{TA}(\tau)$,$\mathit{TA}(pe_j)$ and *DynamicPriorities* are given in Sections 3.1, 3.2 and 3.2.4, respectively.

The timed automata comprising a system communicate by synchronous communication and shared variables. We give an overview of the channel communication in Fig. 6. Suppose that the set of tasks $\{\tau_{j_1},\ldots,\tau_{j_k}\} = \mathcal{T}_{pe_j}$ is executed on the processing element pe_j. The timed automata for these k tasks and the processing element pe_j share four channels ready$_j$, run$_j$, preempt$_j$ and finish$_j$. The model will be constructed so that all channel events will occur at integer time points only, and the correctness of the construction relies on that property.



Identifier	Type	Comment
`tau`	int	τ
`pe`	int	Identification of τ's processing element: $m(\tau) =$ `pe`
`cp`	clock	Modelling the periodicity of a task
`offset`	int array	$o_{\tau_i} =$ `offset[`i`-1]`
`pi`	int array	$\pi_{\tau_i} =$ `pi[`i`-1]`
`e, exe`	int	Non-deterministic selection of an int in the interval $[bcet; wcet]$fl
`cr`	int	Modelling $exec$ - the accumulated running time of a task
`x`	clock	Used for discretization of the running time
`missedDeadline`	bool	Flag set when a task misses a deadline

Fig. 7. Simplified template for the task.

3.1. Template for a task

We present the timed automaton $TA(\tau)$ for a task $\tau \in \mathcal{T}_{pe_j}$ (`tau`) with offset o (`offset`), period π (`pi`), and best-case and worst-case execution times *bcet* (`bcet`) and *wcet* (`wcet`), respectively, in two steps. In this Uppaal model `offset` and `pi` are global arrays with an entry for each task τ. In the same way there are four channel arrays `ready`, `run`, `preempt` and `finish` each with an entry for every processing element *pe*.

Simplified template for a task

A first simplified template can be seen in Fig. 7 together with a table explaining the identifiers. We will focus on timing and channel synchronization in this part.

A clock `cp` is used to handle the offset and the periodicity of the task. The timed automaton stays in the `Start` location until the offset time has elapsed. In the remaining locations, the task has entered the periodic behavior, or the "error-location" `missedDeadline`.

In the `Done` location, the task has finished its job in the current period and is awaiting the start (when `cp == pi`) of the next, where the transition from `Done` to `Released` is taken. This is signalled to the processing element pe_j using the channel $\texttt{ready}_j$. In this transition the clock `cp` and other variables are initialized. In particular, a choice is taken setting `e` non-deterministically

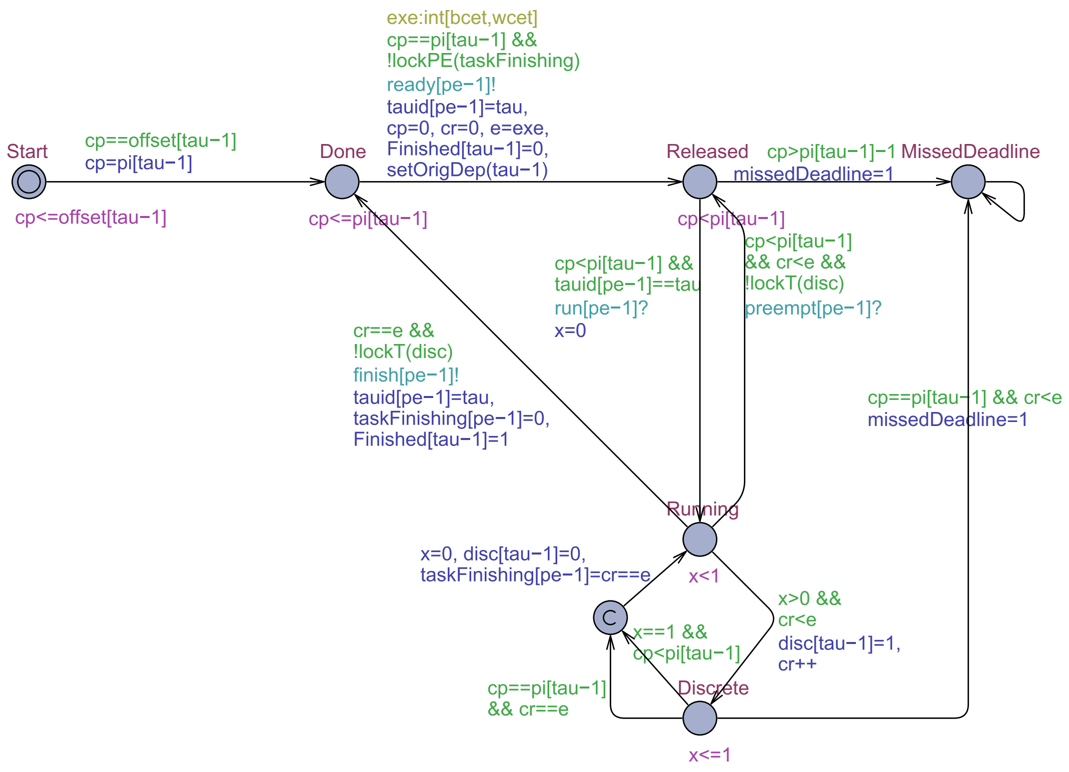


Fig. 8. Full template for the task.

to a value in $\{bcet,bcet+1,\ldots,wcet\}$. This corresponds to the choices for the execution vector in the computation tree when a new period for τ starts. Furthermore, the integer variable `cr` holds the execution time for τ in the current period and is initialized to 0.

In the "standard version" of Uppaal clocks cannot be stopped.[1] Since we have preemptive scheduling, we cannot use a clock to hold the execution time for τ in the current period. We therefore have to introduce the integer variable `cr` and update it in discrete steps as described in the following.

The transition from `Released` to `Running` takes place when the processing element pe_j sends a signal on the $\texttt{run}_j$ channel to the task. This transition takes place when the task is enabled and has highest priority among the enabled tasks on pe_j. On this transition, the clock `x`, which is used to record one time unit, is set to 0. In one round from state `Running` via `Discrete` and back to `Running` the effect is that one time unit has elapsed and that the running time `cr` is incremented by 1 (as the previous $\texttt{run}_j$ signal is issued at an integer time point where `cp < pi`). Notice that the committed location is introduced as Uppaal does not allow disjunctions in guards.

Transitions from `Released` and `Discrete` to `missedDeadline` are taken when a deadline is missed. Furthermore, a transition from `Running` to `Done` is taken when the task has finished its job in the current period `cr == e` and this is signalled to the processing element pe_j by a $\texttt{finish}_j$ event. Since `cr` is not a clock we ensure that the transition is taken without delay by making the `finish` channels urgent. In the `Running` location, the task may be preempted when a task on pe_j with higher priority becomes enabled. In this case the processing element issues a $\texttt{preempt}_j$ signal and the task makes a transition to the `Released` state.

Full template for a task

The full timed-automaton template for a task can be seen in Figs. 8, and 9 provides a table with explanations for the introduced identifiers. The added information concerns dependencies, value transfer in channel communications and locking mechanisms. We will briefly discuss the main parts.

Dependencies. In order to manage dependencies, two global Boolean matrices `depend` and `origdep` are used, where `origdep` represents the static task graph $\prec$, and `depend` the dependencies at the current time point. Initially `origdep` and `depend` are equal, and if $\tau_i \prec \tau_j$ and τ_i has finished its job in the current period at the current time, then the entry for (τ_i,τ_j) in `depend` is false and we say that that particular dependency is *resolved*. The procedure `setOrigDep` updates `depend` each time a task is

[1] We are currently experimenting with a version of Uppaal supporting stopwatches and verification using over-approximations.

Identifier	Type	Comment
`tauid`	int array	Global array used for channel communication
`pr`	int array	Priority of τ according to pr: $pr(\tau_i) = \texttt{pr}[\texttt{i} - 1]$
`lockPE`	predicate	is true iff a task on a processing element is finishing but not in the `Done` location
`taskFinishing`	bool array	$\texttt{taskFinishing}[j-1] = \text{true}$ iff a task on pe_j is currently finishing but not in the `Done` location
`Finished`	bool array	$\texttt{Finished}[i-1] = \text{Finished}_{\hbar}(\tau_i, n)$, where n is the current period
`setOrigDep`	procedure	Maintaining dynamic dependency information
`lockT`	predicate	is true iff some task is executing but not in the running location
`disc`	bool array	$\texttt{disc}[i-1] = \text{true}$ iff τ_i is currently executing but not in the `Running` location

Fig. 9. Explanation of some of the identifiers used in Fig. 8.

entering the `Release` location. It is easy to implement this procedure, as the needed information is available in the variables: `origdep`, `Finished`, and `offset`.

Value transfer. Communication between the platform and the application is done over the synchronization channels `ready`, `run`, `preempt` and `finish`. Some of this communication requires value transfer in order to identify the task in question (as all tasks on a given processing element share the same channels). The global array `tauid` is used so that `tauid[`j`-1]` handles the communication on pe_j.

Locking mechanisms. In order to avoid some undesired behavior, two locking mechanisms are used. One locking mechanism – implemented by the array `disc` and the Boolean function `lockT` – makes sure that all tasks are able to react (i.e. they are in the location `Running`) when a `preempt` signal can occur. The other locking mechanism – implemented by the array `taskFinishing` and the Boolean function `lockPE` – makes sure that all `finish` signals are reacted to by the platform before any `ready` can be handled. This is done in order not to preempt a task that has actually just finished.

3.2. Timed-automata model for processing elements

The model for a processing element is described by a parallel composition of a controller, a synchronizer and a scheduler:

$$TA(pe_j) = Controller_j \parallel Synchronizer_j \parallel Scheduler_j$$

where the controller coordinates the activities inside the processing element (i.e. with the tasks, the synchronizer and the scheduler) and it coordinates the interaction between the processing elements, which is needed since dependent tasks may be mapped to different processing elements. The synchronizer maintains a consistent view of the dynamic dependency relation between tasks, and the scheduler implements the scheduling discipline.

The communication internally on each processing element and between the different processing elements is shown in Fig. 10 in terms of communication between pe_j and pe_l.

3.2.1. Template for the controller

A timed-automaton template for a controller is given in Fig. 11 and the identifiers introduced are explained in Fig. 12. The main idea is that the controller, in the initial location `Idle`, can react to `reschedule` signals from other controllers and to `ready` and `finish` signals from its own tasks. On such events it invokes its synchronizer and scheduler on the `synchronize` and `schedule` channels as needed, and before reentering the `Idle` location it issues the necessary `preempt`, `run` and `reschedule` signals.

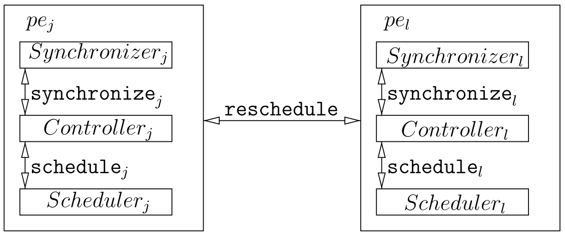


Fig. 10. Communication internally and between processing elements.

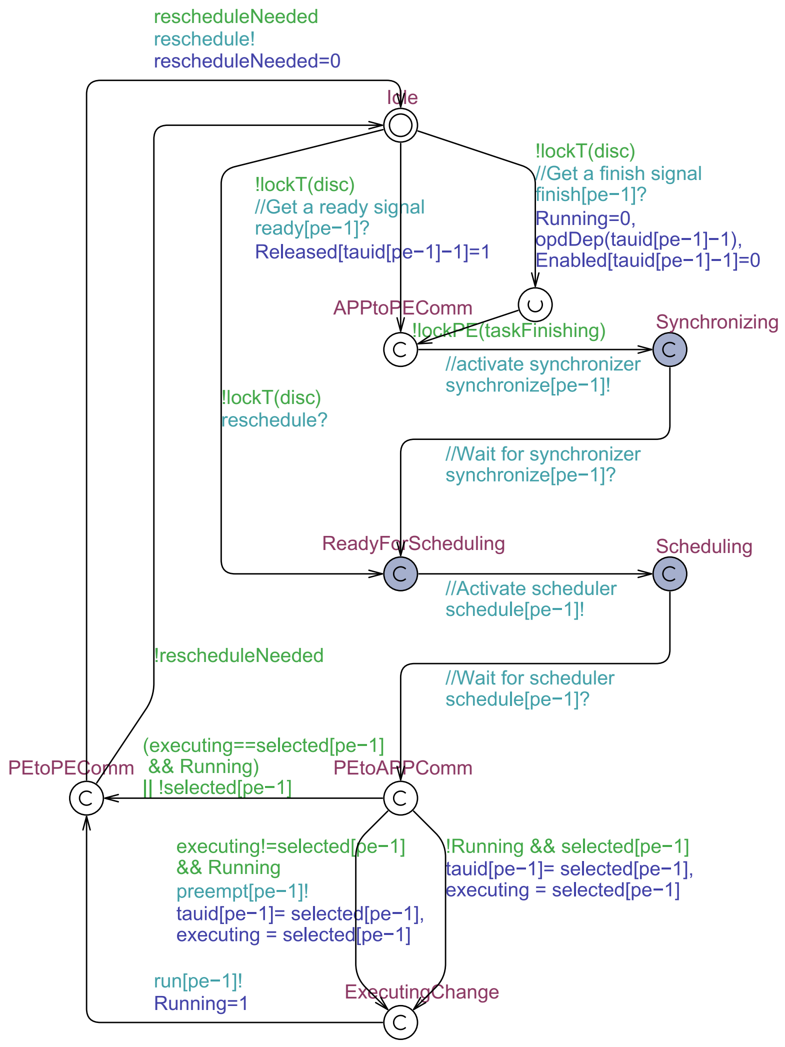


Fig. 11. Template for the controller.

Notice that time does not progress in one round from the `Idle` location and back, as all other locations of the controller are committed or urgent. The single urgent location is used to capture all `finish` signals available at that point in time.

Identifier	Type	Comment
`pe`	int	Identification of processing element
`Released`	bool array	`Released`$[i-1]$ = true iff τ_i is currently released
`Running`	bool	Processing element is currently executing a task
`opdDep`	procedure	Updates dependencies when a task finishes
`Enabled`	bool array	`Enable`$[i-1]$ = true iff τ_i is enabled
`selected`	int array	`selected`$[i-1]$ = true iff τ_i is selected on pe
`executing`	int	The task currently executing
`rescheduleNeeded`	bool	Flag set when a global reschedule is needed

Fig. 12. Explanation of some of the identifiers used in Fig. 11.

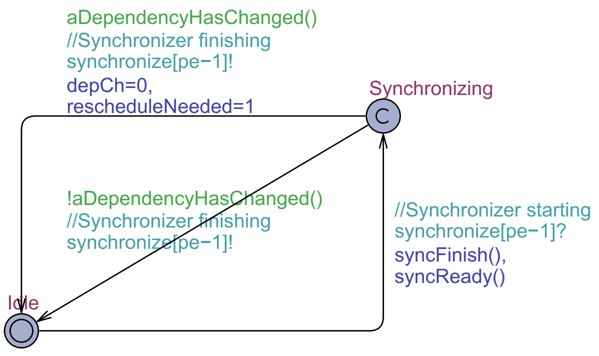


Fig. 13. Template for the synchronizer.

Each controller maintains arrays describing the tasks that are currently released, enabled and executing, and on the basis of this information it is not difficult to implement the procedure `opdDep`, which updates the dynamic dependencies due to a finish signal. Notice that the flag `rescheduleNeeded` is set by the synchronizer and the variable `selected` is set by the scheduler.

3.2.2. *Template for the synchronizer*

The template for the synchronizer is given in Figs. 13 and 14 gives an explanation of the identifiers introduced.

When receiving a `synchronize` signal from the controller, the synchronizer updates the dynamic dependencies caused by the currently released and finished tasks, using the procedures `syncReleased` and `syncFinish`, respectively. These procedures, which maintain the arrays `WaitDep`, `Enabled` and `Released` as well as the flag `depCh`, are easily implemented. In the transition from the `Synchronizing` location to the `Idle` location, the flag `rescheduleNeeded` is set when a dependency has changed (i.e. `depCh` is true).

3.2.3. *Template for the scheduler*

The template for the scheduler is shown in Figs. 15 and 16 provides an explanation for the identifiers introduced. The procedure `tasksReadyOnProc` initiates the scheduling decision making by finding a task that is enabled, setting `selected` accordingly, and setting `aTaskIsReady` to true. If no tasks on the processing element are enabled, `aTaskIsReady` is set to false.

The procedure `findHighestPriority` finds the enabled task with the highest priority on the processing element based on the scheduling discipline given in `Sch` and sets `selected` to the number of that task. The cases for the static scheduling disciplines (fixed priority and rate monotonic in our case) are straightforward implementations that are based on the definitions in Section 2.3.1. The difficult part concerning earliest-deadline-first scheduling are dealt with in the next section.

3.2.4. *Template for dynamic priorities*

The template for managing dynamically updated priorities is shown in Figs. 17 and 18 provides an explanation for the identifiers introduced. This timed automaton works for all processing elements having earliest deadline first as scheduling discipline.

The construction of this timed automaton is based on the property (6), which describes the cyclic behavior of situations, where each situation corresponds to an earliest-deadline-first priority list for the tasks. It is, of course, only necessary to consider tasks that are scheduled according to the earliest-deadline-first discipline. The self-loop for the location `OffsetPriorities` covers the parts until the time of the maximal offset, and the self-loop for the location `Priorities` covers the cyclic part of (6). At each transition a reschedule signal is broadcasted to all processing elements (i.e. all controllers).

The hard part is the construction of values for the variables `OffSteps`, `Steps`, `OffPrios` and `Prios`. But these can be generated by a program traversing the sequence of situations until the end of the first hyper-period. The idea is simple. Start

Identifier	Type	Comment
`syncFinish`	procedure	Updates dynamic dependencies for finished tasks
`syncReleased`	procedure	Updates dependencies for released tasks
`depCh`	bool	Flag indicating that a dependency has changed
`WaitDep`	bool array	Tasks with unresolved dependencies

Fig. 14. Explanations for the identifiers used in Fig. 13.

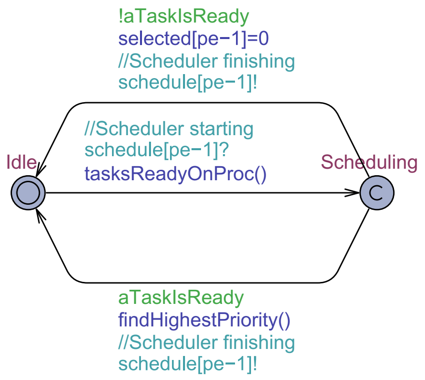


Fig. 15. Template for the scheduler.

Identifier	Type	Comment
`tasksReadyOnProc`	procedure	Initiating the scheduling process `aTaskIsReady`
`aTaskIsReady`	bool	Flag indicating whether any task is ready
`findHighestPriority`	procedure	Implements the $Select_{\hbar}$ function
`Sch`	array	Implements Sch, i.e. $Sch(pe_i) = \texttt{Sch}[i-1]$

Fig. 16. Explanation for some of the identifiers used in Fig. 15.

finding the longest prefix of $sit_1\, sit_2\, \cdots\, sit_{k_0}$ for which the priority list is constant. Then `OffPrios[0]` contains this priority list and `OffSteps[0]` is equal to k_0. Proceed finding the longest prefix $sit_1\, \cdots\, sit_{k_0} sit_{k_0+1}\, \cdots sit_{k_0+k_1}$, for which the priority list is constant for the subsequence $sit_{k_0+1}\, \cdots sit_{k_0+k_1}$. Then `OffPrios[1]` contains that priority list and `OffSteps[1]` is k_1. This process stops when the situation for the maximal offset is reached, and then a similar process for `Steps` and `Prios` can start.

4. Verification of MPSoC Using UPPAAL

The representation of a system model in the Uppaal system can be used for simulation as well as for verification. We will here focus on the verification aspect for the schedulability problem only, where we verify whether every task meets its deadline in every period. This is **not** the case if there is some state on some path where the Boolean variable `missedDeadline` (see Fig. 8) is true. Hence, we issue the Uppaal query:

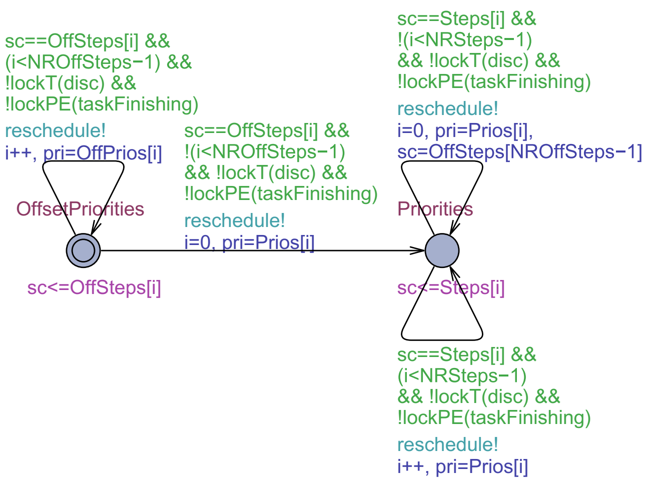


Fig. 17. Template for maintaining dynamic priorities.

Identifier	Type	Comment
`sc`	clock	Used for changes in dynamic priorities
`OffSteps`	int array	Time steps for changes before the maximal offset O_M
`Steps`	int array	Time steps for changes after O_M until $O_M + \Pi_H$
`i`	int	Variable used for counting
`OffPrios`	array	Elements are priority lists which are valid before O_M
`Prios`	array	Elements are priority lists which are after before O_M
`NROffSteps`	int	The length of `offSteps` and `offPrios`
`NRSteps`	int	The length of `Steps` and `Prios`

Fig. 18. Explanation of some of the identifiers used in Fig. 17.

`E<>missedDeadline`,

and every task meets every deadline iff this property is not satisfied.

The UPPAAL system can generate a diagnostic trace for such queries and this is a very useful facility for understanding the scenario where a deadline is missed. It should be noted that this trace is expressed in terms of the timed-automata model and not in the context of the overall MPSoC model in Section 2.

We have tested the UPPAAL model on a collection of examples found in the literature and the verifications gave correct results.

A tool has been developed for the generation of a UPPAAL model on the basis of a simple description of a system model. The UPPAAL model is developed on the basis of a java-program. This program contains constructors for the components of a system, i.e. for tasks, applications, processors and platforms. Fig. 20 contains the java-program corresponding to the example in Fig. 19.

The system consists of two processors with two tasks on each. Rate-monotonic scheduling is used on both processors. The system has an inter-processor dependency from τ_2 to τ_3. The tasks have the following timing constraints, where the best-case execution time equals the worst-case execution time for every task:

Task	Processor	Execution time	Period	Offset
τ_1	pe_1	2	4	0
τ_2	pe_1	2	6	0
τ_3	pe_2	2	6	0
τ_4	pe_2	3	6	4

This system has an inter-processor dependency and an offset on τ_4.

The system is not schedulable and the following trace is extracted by the tool from the diagnostic UPPAAL -trace:

```
Time    1   5
---------------
Task: 1 1100110
Task: 2 0011001
Task: 3 000000x
Task: 4 ----111
```

The trace covers the first seven time units and the missed deadline is marked with `x`. Furthermore, the symbol `-` marks that the task has not yet started, `0` marks that the task is not executing and `1` marks that the task is executing. Hence, task τ_3 misses its deadline after 6 time units.

It is easy to explore various designs using this tool and, for example, changing the real-time operating system on pe_2 to use earliest deadline first will give a schedulable system. The only change needed in the java code concerns the line initializing `p2` as described in the comment. Another observation, which is easily verified, is that if the offset for τ_4 is set to 0, the system is schedulable in both the case where rate-monotonic scheduling is used and when earliest deadline first is used. Hence, the case where the offset is zero for all tasks, which is the classical worst-case scenario for single processor systems, does not necessarily give the worst-case scenario in the multiprocessor case.

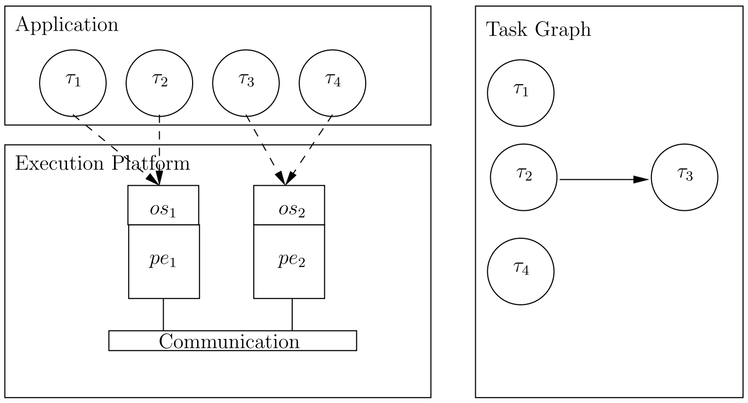


Fig. 19. Example of a multiprocessor system with a dependency.

```
import java.io.*;
public class Example{
  public static void main(String[] args){
    String fl = "EXm4";
    Task t1 = new Task(2, 4, 0, 4, 1, 1);
    Task t2 = new Task(2, 6, 0, 6, 2, 1);
    Task t3 = new Task(2, 6, 0, 6, 3, 2);
    Task t4 = new Task(3, 6, 4, 6, 4, 2);

    Processor p1 = new Processor(1, Processor.RM);
    Processor p2 = new Processor(2, Processor.RM);
    // Processor p2 = new Processor(2, Processor.EDF);

    Task[][] tasks = { { t1, t2}, { t3, t4} };

    Processor[] ps = {p1,p2};
    Platform platform = new Platform(ps);
    Applications apps = new Applications(tasks);

    apps.addDep(t2,t3);

    SysM sys = new SysM(apps,platform);
    try{
      PrintStream fout = new PrintStream(new File(fl+".xml"));
      fout.println(sys.s());
    }
    catch(Exception e){e.printStackTrace();}
  }
}
```

Fig. 20. Java-code for the system in Example 19.

Consider the example in Fig. 21, which shows a multiprocessor anomaly where, contrary to usual single-processor scheduling theory, the system is not schedulable using dynamically updated scheduling criteria, but it is schedulable using statically updated scheduling criteria.

This system is not schedulable using earliest-deadline-first scheduling disciplines. However, when using fixed-priority scheduling with $pr(\tau_1) = 2$, $pr(\tau_2) = 1$ and $pr(\tau_3) = 3$, the system is schedulable. This is easily verified using the tool.

task	processor	execution time	period	offset
τ_1	pe_1	2	4	0
τ_2	pe_1	2	5	0
τ_3	pe_2	2	5	0

Dependency: $\tau_2 \prec \tau_3$

$wcet_\tau = bcet_\tau$

Fig. 21. Example: Rate-monotonic vs. earliest-deadline-first scheduling.

We will now examine the system given in Fig. 5. In section 2.5 it was shown that the minimum number of nodes in the computation tree will be $7.6 \cdot 10^{10}$ at the depth $O_M + \Pi_H$ (i.e. $t = 22115$). The maximal depth that is needed when checking for schedulability is $O_M + 8 \cdot \Pi_H$ (i.e. $t = 176731$) with $3.9 \cdot 10^{13}$ nodes. The schedulability was verified using the Uppaal model. The verification used 3.1 GB of memory and took less than 11 minutes on an AMD CPU of 1.8 MHz and 32 GB of RAM.

If the system is changed slightly by adding an extra choice for execution time to τ_3 i.e. $wcet_{\tau_3} = 14$, the number of nodes at depth $O_M + \Pi_H$ will be $8.2 \cdot 10^{10}$ and the number of nodes at the depth $O_M + 8 \cdot \Pi_H$ will be $4.2 \cdot 10^{13}$. When attempting verification of this revised system on the same CPU, the verification aborts after 19 min with an *Out of memory* error message after having used 3.4 GB of memory.

4.1. Further experimental results

In order to make verification on real-life systems we have experimented [18] with a specification of a smart-phone device [29] consisting of 103 tasks on 4 processing elements. By only using worst-case execution times and, therefore, only having one run in the computation tree, we could verify the system using a specialized version of Uppaal where no history is saved and therefore less memory is used. This indicates that the model can also be used in analysis of larger real-life systems.

A more intuitive implementation of the MPSoC model would be using stopwatch automata. However, the absence of stopwatches in the standard Uppaal version moved our focus to that of discretization of the running time. A version of Uppaal under development introducing stopwatches – using over approximations for verification – has however been made available to us, and we have made some initial experiments with this. The stopwatch model eliminates the need for the integer variable `cr` and the clock `x` and introduces instead a stopwatch. Furthermore, the locking mechanism used for the discretization is eliminated. The experiments with verification of the stopwatch model have given encouraging results, as the example that gave an *out of memory* error with the standard version can be verified with the stopwatch version. All current experiments have provided exact results. The tendencies on all experiments so far is that memory consumption as well as verification times are reduced by approximately 40% in comparison to the original model given here.

5. Summary

We have developed a discrete model of computation capturing the synchronization and timing behavior of an embedded application executing on multiple interconnected processors. Tasks mapped to a processor are executed according to a particular scheduling policy. Analysis of such systems is a major challenge due to the freedom of interrelated choices concerning the application level, the configuration of the execution platform and the mapping of the application onto this platform, which often leads to timing anomalies. We have characterized the maximal size of the computation tree it suffices to be considered when checking for schedulability.

The computational model provides a basis for formal analysis of systems and has been expressed as timed automata and implemented in the Uppaal system. This allows for verification using the Uppaal model checker. We have presented experimental results on rather small systems with high complexity, primarily due to differences between best-case and worst-case execution times. By considering worst-case execution times only, where the system becomes deterministic, we have shown that it is possible to verify a smart-phone application consisting of 103 tasks executing on a platform with four processors.

We are currently extending our model to introduce costs other than timing, being able to formally verify properties such as memory and power usage. Furthermore, initial experiments with a stopwatch model have given interesting results, as the analyzes where more efficient than those using the "standard Uppaal tool" and they were precise on the examples conducted despite the fact that verification is based on over-approximations.

Acknowledgements

We would like to thank Jens Sten Ellebæk Nielsen and Kristian Stålø Knudsen for contribution to the simplification of the model and working extensively on the frontend tool. Furthermore, we are grateful for comments from Martin Fränzle, Kim G. Larsen and Tolga Ovatman on this work. Finally, we are grateful to Jacob Illum and Alexandre David for providing us with versions of Uppaal under development, in order for us to conduct initial experiments with verification of larger real-life systems as well as models with stopwatches.

The work presented in this paper has been supported by ARTIST2 (IST-004527), MoDES (Danish Research Council 2106-05-0022) and the Danish National Advanced Technology Foundation under project DaNES.

References

[1] Y. Abdeddaïm, E. Asarin, O. Maler, Scheduling with timed automata, Theoret. Comput. Sci., 354 (2) (2006) 272–300.
[2] Y. Abdeddaïm, O. Maler, Job-shop scheduling using timed automata, Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01), vol. 2102, Springer, 2001.
[3] R. Alur, D.L. Dill, A theory of timed automata, Theoret. Comput. Sci. 126 (2) (1994) 183–235.

[4] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, W. Yi, Times – A tool for modelling and implementation of embedded systems, Lecture Notes in Comput. Sci. 2280 (2002) 460–464.
[5] G. Behrmann, A. David, K.G. Larsen, A tutorial on uppaal, Lecture Notes in Comput. Sci. 3185 (2004) 200–236.
[6] E. Fersman, P. Pettersson, W. Yi, Timed automata with asynchronous processes: schedulability and decidability, Lecture Notes in Comput. Sci. 2280 (2002) 67–82.
[7] R.L. Graham, Bounds on multiprocessor timing anomalies, SIAM J. Appl. Math. 17 (2) (1969) 416–429.
[8] L. Halkjaer, K. Haervi, A. Ingolfsdottir, Verification of the LegOS scheduler using uppaal, Electron. Notes Theor. Comput. Sci. 39 (3) (2000) 273–292.
[9] P. Hastrono, S. Klaus, S.A. Huss, An integrated SystemC framework for real-time scheduling assessments on system level, in: Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS'04), 2004.
[10] C. Haubelt, J. Falk, J. Keinert, T. Schlichter, M. Streubühr, A. Deyhle, A. Hadert, J. Teich, A SystemC-based design methodology for digital signal processing systems, EURASIP J. Embedded Syst. 2007 (1) (2007) 15.
[11] F. Hessel, V.M. da Rosa, I.M. Reis, R. Planner, C.A.M. Marcon, A.A. Susin, Abstract RTOS modeling for embedded systems, Proceedings of the 15th IEEE International Workshop on Rapid System Prototyping (RSP'04), IEEE Computer Society, Washington, DC, USA, 2004.
[12] G.J. Holzmann, The SPIN Model Checker: Primer and Reference Manual, Addison-Wesley, 2003.
[13] T. Kempf, M. Doerper, R. Leupers, G. Ascheid, H. Meyr, T. Kogel, B. Vanthournout, A modular simulation framework for spatial and temporal task mapping onto multi-processor SoC platforms, Proceedings of the Conference on Design, Automation and Test in Europe (DATE'05), vol. 2, IEEE Computer Society, Washington, DC, USA, 2005.
[14] V. Kianzad, S.S. Bhattacharyya, CHARMED: a multi-objective co-synthesis framework for multi-mode embedded systems, in: Proceedings of the 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP'04), 2004.
[15] K.G. Larsen, P. Pettersson, W. Yi, Uppaal in a nutshell, Int. J. Software Tools Technol. Transf. 1 (1–2) (1997) 134–152.
[16] T.G.S. Liao, G. Martin, S. Swan, System Design with systemC, Springer, 2002.
[17] M. Loghi, F. Angiolini, D. Bertozzi, L. Benini, R. Zafalon, Analyzing on-chip communication in a MPSoC environment, in: Proceedings of the Conference on Design, Automation and Test in Europe (DATE'04), IEEE, 2004.
[18] J. Madsen, M.R. Hansen, K.S. Knudsen, J.E. Nielsen, A.W. Brekling, System-level verification of multi-core embedded systems using timed-automata, in: Proceedings of the 17th World Congress International Federation of Automatic Control Seoul, Korea, July 6–11, 2008, pp. 9302–9307.
[19] J. Madsen, S. Mahadevan, K. Virk, Network-centric system-level model for multiprocessor SoC simulation, Interconnect-Centric Design for Advanced SoC and NoC, Kluwer Academic, 2004, pp. 341–365.
[20] J. Madsen, K. Virk, M.J. Gonzalez, A SystemC-based abstract real-time operating system model for multiprocessor system-on-chip, in: Multiprocessor System-on-Chip, Morgan Kaufmann, 2004, pp. 283–312.
[21] S. Mahadevan, K. Virk, J. Madsen, ARTS: a SystemC-based framework for multiprocessor systems-on-chip modelling, Des. Automat. Embedded Syst. 11 (4) (2007) 285–311.
[22] R.L. Moigne, O. Pasquier, J.-P. Calvez, A generic RTOS model for real-time systems simulation with SystemC, in: Proceedings of the Conference on Design, Automation and Test in Europe (DATE'04), vol. 3, IEEE Computer Society, 2004.
[23] J.F. Muth, G.L. Thompson, Industrial Scheduling, Prentice-Hall International, Englewood Cliffs, NY, USA, 1963.
[24] A.D. Pimentel, C. Erbas, S. Polstra, A systematic approach to exploring embedded system architectures at multiple abstraction levels, IEEE Trans. Comput. 55 (2) (2006) 99–112.
[25] A.D. Pimentel, L.O. Hertzberger, P. Lieverse, P. van derWolf, E.F. Deprettere, Exploring embedded-systems architectures with artemis, IEEE Comput. 34 (11) (2001) 57–63.
[26] P. Pop, P. Eles, Z. Peng, Bus access optimization for distributed embedded systems based on schedulability analysis, in: Proceedings of the Conference on Design, Automation and Test in Europe (DATE'00), ACM, New York, NY, USA, 2000.
[27] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, B. Becker, A Definition and classification of timing anomalies, in: Proceedings of 6th International Workshop on Worst-Case Execution Time (WCET) Analysis, 2006.
[28] K. Richter, M. Jersak, R. Ernst, A formal approach to MpSoC performance verification, IEEE Comput. 36 (4) (2003) 60–67.
[29] M.T. Schmitz, B.M. Al-Hashimi, P. Eles, System-Level Design Techniques for Energy-Efficient Embedded Systems, Kluwer Academic Publishers, Norwell, MA, USA, 2004.
[30] L. Thiele, S. Chakraborty, M. Naedele, Real-time calculus for scheduling hard real-time systems, in: Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS 2000), vol. 4, Geneva, Switzerland, 2000.
[31] S. Tripakis, K. Altisen, Implementation of timed automata: an issue of semantics or modeling?, Lecture Notes in Comput. Sci. 3829 (2005) 273–288.