A reconfigurable framework for compositional schedulability and power analysis of hierarchical scheduling systems with frequency scaling

Boudjadar, Abdeldjalil; David, Alexandre; Kim, Jin Hyun ; Larsen, Kim Guldstrand; Mikučionis, Marius; Nyman, Ulrik Mathias; Skou, Arne Joachim

Published in:
Science of Computer Programming

DOI (link to publication from Publisher):
10.1016/j.scico.2015.10.003

Publication date:
2015

Document Version
Accepted author manuscript, peer reviewed version

A Reconfigurable Framework for Compositional Schedulability and Power Analysis of Hierarchical Scheduling Systems with Frequency Scaling ☆

Abdeldjalil Boudjadar

Computer Science, Aalborg University, Denmark

Alexandre David

Computer Science, Aalborg University, Denmark

Jin Hyun Kim

Computer Science, Aalborg University, Denmark

Kim G. Larsen

Computer Science, Aalborg University, Denmark

Marius Mikučionis

Computer Science, Aalborg University, Denmark

Ulrik Nyman

Computer Science, Aalborg University, Denmark

Arne Skou

Computer Science, Aalborg University, Denmark

Abstract

This paper presents a compositional framework for the modeling and analysis of hierarchical scheduling systems. We consider both schedulability and energy consumption of individual components, while analyzing a single core setting with a voltage frequency scaling CPU. According to the CPU frequency scaling, each task has a set of different execution times. Thus, the energy consumption of the whole system varies from an execution to another.

☆The research presented in this paper has been partially supported by EU Artemis Projects CRAFTERS and MBAT.

Email addresses: jalil@cs.aau.dk (Abdeldjalil Boudjadar), adavid@cs.aau.dk (Alexandre David), jin@cs.aau.dk (Jin Hyun Kim), kgl@cs.aau.dk (Kim G. Larsen), marius@cs.aau.dk (Marius Mikučionis), ulrik@cs.aau.dk (Ulrik Nyman), ask@cs.aau.dk (Arne Skou)

We analyze each component individually by checking the feasibility of its workload against both the CPU availability and energy consumption constraints of such a component. Our periodic task model considers both static and dynamic priorities together with preemptive and non preemptive behaviors. The models are realized using different forms of Hybrid Automata, all of which are analyzed using variants of Uppaal. The CPU frequencies, task behavior and scheduling policies used in each component are some of the reconfigurable parameters of the system. Finally, we demonstrate the applicability and scalability of our framework by analyzing the schedulability and power consumption of an avionics system.

Keywords: Hierarchical Scheduling Systems, Schedulability Analysis, Power Consumption, Voltage/Frequency Scaling, Uppaal.

---

1. Introduction

In the design of modern automotive systems, in order to reduce the system cost a manufacturer devotes strong efforts in minimizing the resource requirements of individual components, provided by different suppliers, in order to maximize the number of components to be integrated on a given platform while ensuring the whole system to be continually feasible. Such concurrent components might share platform resources (e.g. processors, battery).

Resource utilization represents a common challenge in embedded systems, and thus it is important to have both an efficient and reliable scheduling policy for the individual parts of the system. Scheduling is a widely used mechanism for guaranteeing that the different components of a system will be provided with the correct amount of resources. Many classical approaches have been developed to analyze different types of scheduling systems. The working hypothesis behind the research presented in this paper is that a model-based approach to schedulability and power analysis has advantages of being more flexible than the classical analytical approaches in terms of expressiveness and precision.

In this paper, we propose a compositional model-based framework for analyzing the schedulability and power consumption of hierarchical scheduling systems. Our framework is implemented using different forms of stopwatch timed automata which are analyzed using variants of Uppaal. In order to capture the effect of preemption in hierarchical scheduling systems, we are using the concept of stopwatches. Stopwatches generally present a problem for the exact system analysis as they belong to the undecidable fragment of hybrid automata. However, since only expressions depending on the discrete part of the state can be assigned to a clock or stopwatch in Uppaal, this ensures the decidability of the system (for further details see Section 3.5).

We have not found any alternative way to model preemptive hierarchical scheduling systems, with a dense time semantics, using standard timed automata. We believe that it is indeed impossible, but have not tried to prove this fact. According to [31], the schedulability checking problem for non-preemptive

scheduling policies is decidable because the schedulability question can be translated to a reachability question. Moreover, such a problem is also decidable in case of preemptive algorithms if the computation times are constant single values. Otherwise, the schedulability checking is undecidable if the following holds:

- the execution times of tasks are intervals given in terms of best and worst cases.
- the finishing time of a task execution (instance) influences the release of new instances.
- a ready task is allowed to preempt a running task.

A proof of the decidability of FPS (Fixed Priority Scheduling) and EDF (Earliest Deadline First) is presented in [31].

Model checking in general suffers from state space explosion, where the state space that needs to be explored grows exponentially in the size of the parallel composition. In order to combat the state space explosion and be able to use model checking on larger problems, we have chosen to decompose the models. This has, in many other settings, proven to be fruitful way of applying model checking to real problems. The size of the final system is unimportant, but rather the maximum number of tasks that needs to be analyzed together. Thus our framework can be scaled to larger systems if the system tasks are grouped into components in a hierarchy.

A hierarchical scheduling system consists of a finite set of components, a scheduling policy and resources (energy and processor time). Each component, in turn, is the parallel composition of a finite set of entities which are either tasks or other components together with a scheduling policy to manage the component workload. Tasks are instances of the same timed automaton template with different (input) parameters. Thanks to the parameterization, the framework can easily be instantiated for a specific hierarchical scheduling application. Similarly, each scheduling policy (e.g. EDF: Earliest Deadline First, FP: Fixed Priority, RM: Rate Monotonic) is modeled separately and can be instantiated for any component. We extend our framework with power consumption by adding a new constraint attribute to the component interface. For energy awareness, we analyze the energy consumed by a component workload against the power consumption constraint of that component. In this way a fixed frequency for each task is found on a voltage/frequency scaling platform, for which the component is both schedulable and satisfies the maximum power constraint. For the sake of simplicity, we focus on single core systems but the framework is also applicable for the case of multi-core systems.

Compositional analysis was first introduced [18, 34] as a key model-checking technology, to deal with state space explosion caused by the parallel composition of components. We are applying compositional verification to the domain of schedulability analysis. By schedulability analysis, we mean checking whether a set of real-time tasks can be scheduled without missing any deadline.

We analyze the model in a compositional manner by layers/levels; the schedulability of each level is analyzed together with the interface specifications of the

level directly below it. In this analysis, we non deterministically supply the required resources of each component. This fact is viewed by the component entities as a contract by which the component must supply the required resources, provided by the parent level component, to its sub-entities for each period. The main contribution of the paper is combining:

- *a compositional analysis approach* where the schedulability of a system relies on the recursive schedulability analysis of its individual subsystems.
- *energy consumption analysis* of components on a voltage/frequency scaling platform where the energy consumed by the workload is analyzed against the component energy constraint.
- *an adaptable schedulability framework* where a system structure can be instantiated in different configurations to fit different applications.
- *error traces*: we demonstrate the usefulness of model-based approaches by showing that error traces can be used to diagnose why a system is non schedulable.
- *formal basis*: we describe the specification and semantics of our framework in terms of transition systems.

This paper is an extended version of [12] containing detailed background and extended related work. We also give a thorough introduction and overview of the formal basis of the modeling formalisms we use.

The main new scientific contribution of this paper lies in the extension of the framework to include modeling and analysis of power consumption on a voltage/frequency scaling platform, while providing the specification and semantics of our framework in terms of transition systems.

The rest of the paper is structured as follows: Section 2 introduces related work. Section 3 provides preliminaries on hierarchical scheduling system and the model types used to represent the scheduling systems. Section 4 is an informal description of the main contribution using a running example. Section 5 describes our formal specification, semantics and analysis of hierarchical scheduling systems. In section 6, we give the Uppaal model of our framework. Moreover, we show how the compositional analysis can be applied on the model using the Uppaal and Uppaal SMC verification engines. Section 7 presents the energy-aware modeling and analysis of components. Section 8 shows the applicability of our framework, where we analyze the schedulability of an avionics system. Finally, Section 9 concludes our paper and outlines future work.

2. Related Work

Hierarchical scheduling systems were first introduced in [30, 26]. They allow temporal partitioning and separation of concerns. A major motivation of the separation of concerns is that it allows more easily modular design, thus updating a component does not require changing the other components. We also

use the temporal partitioning [37] of components to reduce complexity, confine failure modes and temporal isolate among system applications.

An example of the increasing use of hierarchical scheduling systems is the standard ARINC-653 [5] for avionics real-time operating systems. In [6], Åsberg et al show that hierarchical scheduling is also relevant for soft real-time multicore systems by implementing and evaluating a partitioned hierarchical scheduling framework as an extension of the ExSched framework for the Linux kernel. The following sections overview the ideas that our approach relies on.

2.1. Analytical Approaches to Schedulability Analysis

An analytical compositional framework for hierarchical scheduling systems was presented in [39] as a formal way to elaborate a compositional approach for schedulability analysis of hierarchical scheduling systems [40]. In the same way, Shin et al [38] dealt with a hierarchical scheduling framework for multiprocessors with cluster-based scheduling approach. They used analytical methods to perform analysis, however both approaches [39, 38] have difficulty in dealing with complicated behavior. Compared to that, our task model is more expressive and enables one to describe more concrete behaviors and (user-defined) data types, for example preemption thanks to stopwatches. Moreover, any type of scheduling systems that can be expressed using the UPPAAL language can be analyzed in our framework as long as the framework the model is designed according to the framework definition.

In [35], Lipari et al provide an analytical framework for the formal specification and schedulability analysis of hierarchical scheduling systems. They also present a methodology for how to compute the timing requirements of the intermediate levels (servers) making a set of tasks feasible. The framework only considers static priority scheduling (Fixed Priority Scheduling). We generalize the analysis and such an estimation of the timing requirements to any scheduling mechanism.

In [25], Davis et al analyze the schedulability of hierarchical scheduling systems where fixed priority scheduling is both at the global and local levels. They improve the analysis provided by previous work and also find that harmonic tasks linked to the release of their server improve schedulability. We generalize this work by considering any combination of scheduling policies.

In [28], Easwaran et al introduce an analytical framework for the schedulability analysis of components in a hierarchical setting. The analysis is compositional so that it enables the abstraction of resource requirements of components using periodic resource models. In order to support the incremental analysis, they also provide an extension of the classical component interface to multiple periodic resource models for different periods. Compared to that, for each component we provide a unique constant period and calculate the minimum budget making such a component schedulable rather than providing different periods/budgets for each component.

In [7], Åsberg et al presents a compositional framework of hierarchcial scheduling systems that exploits information of two-level components to compute a

compact resource requirement of component interfaces and analyzes the schedulability using TIMES tool. Compared to [7], we exploit information of a component adopting the original compositional framework of hierarchical scheduling systems.

2.2. Model-based Approaches to Schedulability Analysis

Recent research within schedulability analysis increasingly uses model-based approaches, because this allows for modeling more complicated behavior of systems. While schedulability is a liveness property, it can be reduced to checking a reachability property. This can be done by adding an *Error* state which is immediately reachable from any other state of the given task once the deadline is missed.

In [8], Behnam et al analyzed the schedulability of hierarchical scheduling systems, using a model-based approach with the TIMES tool [4], and implemented their model in VxWorks [8]. Compared to our approach, the schedulability analysis in [8] is not compositional in a way that the analysis of a component does not only consider the timing attributes of that component but also the timing attributes of the other components that can preempt the execution of the component under analysis. Moreover, we are considering more refined and detailed task behaviors than that of the abstract task model in [8].

In [21], David et al introduce a model-based framework using UPPAAL for the schedulability analysis of *flat* systems. The authors model the concrete task behavior as a sequence of timed actions, each one represents a command that uses processing and system resources and consumes time. However, as stated earlier, analyzing the whole system at once might be a scalability challenge due to the state space explosion.

Carnevali et al [14] provide a compositional framework for the verification of hierarchical scheduling systems using preemptive time Petri nets. The authors restrict their work in a way that only Time Division Multiplexing (TDM) can be global schedulers while the local schedulers must be Fixed Priority (FP). Compared to our work, we give more flexibility by allowing to combine different scheduling policies (FP, EDF, RM) at any level of the hierarchy. Moreover, we consider both static and dynamic priorities.

Most recently, Sun et al in [41] presents a model of hierarchical scheduling systems in linear hybrid automata in the most similar way with this work. They introduce a component-based analysis for hierarchical scheduling systems encoded using hybrid automata. The authors prove the correctness of their models and study the decidability of the reachability (schedulability) analysis for the case of periodic tasks. However, similarly to [14], the authors of [41] restrict their framework to be applicable only for system configurations adopting EDF as global scheduler and FPS as local schedulers.

In [11], Bøgholm et al introduce a model-based approach for the verification of safety-critical hard real-time systems, implemented in safety-critical Java, is presented. This work focuses on modeling the actual behavior of the execution platform, but does not use the concept of a hierarchical scheduling system. The

concepts from this work could be combined with the current paper as the lowest level in a compositional process, where the task model is synthesized from code by abstraction.

Basically, our model of hierarchical scheduling systems is extended from [21] that presents a basic model of scheduling systems, in which the preemption of a running task is captured by the stopwatch clock of the priced timed automata. Compared to [8], our framework needs information of just one component of hierarchical scheduling systems to make it possible to analyze hierarchical scheduling system in a fully compositional way. [14] restricted scheduling policy of components of hierarchical scheduling systems to Fixed Priority. In contrast, we allows any scheduling policies that can be formulated as a behavior model of timed automata and its extensions.

2.3. Energy-aware Schedulability Analysis

Since embedded systems are usually running on limited resources, energy efficiency represents a strong factor in the setting of such systems.

In [1], Abdeddaïm et al study the schedulability of real-time embedded systems under energy constraints, such as using solar panels. We extend their approach by considering hierarchy while analyzing the schedulability in a compositional way.

In [36], Niu et al consider the schedulability and energy efficiency of weakly hard scheduling systems on frequency/voltage scaling platforms. A weakly hard scheduling system requires that at least k out of m consecutive executions do not miss their deadline. The framework of [36] is restricted to EDF scheduling and does not consider hierarchy.

In [43], Zhou et al examine energy-efficient scheduling for the Integrated Modular Avionics (IMA) platform. The framework is restricted to a two level hierarchy with Weighted Round-robin (WRR) at the top level and Fixed Priority (FP) at the low level. They use a classical analytical approach to tackle this setting.

We combine and extend the approaches from Carnevali et al [14, 21] by considering hierarchy, power consumption from Nui et al [36], and resource sharing while analyzing hierarchical scheduling systems in a compositional way. Moreover, our models can easily be reconfigured to fit any specific application. Comparing our model-based approach to analytical ones, our framework enables to describe more expressive and precisely detailed and refined systems.

3. Preliminaries

In this section, we first present the background of hierarchical scheduling systems and then give the theory behind our modeling framework.

3.1. Hierarchical Scheduling Systems

A hierarchical scheduling system consists of multiple scheduling systems in a hierarchical structure. It can be represented as a tree of nodes, where each

node in the system is equipped with a scheduler for scheduling its child components. In this paper, we structure our system model as a set of hierarchical components. Each component, in turn, is the parallel composition of a set of entities (components or tasks) together with a local scheduler and possible local resources.

Formally, a hierarchical scheduling system $System = (\mathcal{C}, \mathcal{R}, s)$ is given by a set of hierarchical components $\mathcal{C}$, a set of typed resources $\mathcal{R}$ and a scheduling algorithm s. A component $C \in \mathcal{C}$, in turn, can be either a hierarchical unit $(\{C_1, .., C_n\}, s)$ of other components C_i, or a basic composition (W, s_c) of a workload W, together with a scheduling policy s_c. A component can also declare a set of typed resources which serve as local resources. The workload $W = \langle T_1, .., T_n \rangle$ is a set of real-time tasks having time constraints. Each task $T = \langle \mathsf{prd}, \mathsf{e}, \mathsf{d}, \mathsf{prio}, \mathsf{p} \rangle$ is given with a period (prd), an execution time (e), a deadline (d), a priority (prio) and a preemption (p). The execution time (e) specifies the CPU usage time required by the task execution for each period (prd). Deadline parameter (d) represents the latest point in time that the task execution must be done before. The parameter prio specifies the user priority associated to the task. Finally, p is a Boolean flag stating whether or not the task is preemptive. Task and component parameters prd, budget and e can be single values or time intervals. Moreover, prio and p can be omitted when they are not important.

The real-time interface $\mathcal{I}$ [40] of a system or component specifies the collective resource requirements under a given scheduling policy s. The component can be a hierarchical component $(\mathcal{C}, s)$, with sub-components or a regular component (W, s), with a given workload W. The interface $\mathcal{I}$ is simply given by a period prd, a budget budget and a scheduling policy s in our framework. The budget (budget) specifies the execution time that the component should be provided by its parent level, and the scheduling policy (s) specifies how the resources are allocated by the component to its child entities. A parent component treats the real-time interface of each one of its child components as a single task with the given real-time interface. The component supplies its child entities with resource according to their real-time interfaces. The analysis of a component (scheduling unit) consists of checking that its child entities can be scheduled within the component budget according to the component scheduling policy. An example of a hierarchical scheduling system is depicted in Fig. 1.

3.2. Running Example

In this section and throughout the paper, we present the running example shown in Fig. 1 to illustrate our system model of hierarchical scheduling systems, and show the compositional analysis we claim. For the sake of simplicity, we omit some parameters like priorities and resources and only consider single parameter values instead of time intervals.

In this example, the top level System schedules Component1, Component2 with the EDF scheduling algorithm. The components are viewed by the top level System as tasks having timing requirements. Component1 has the interface (100, 33) as period and execution time and Component2 has (70, 20) as period and execution time. The system shown through this example is schedulable if each

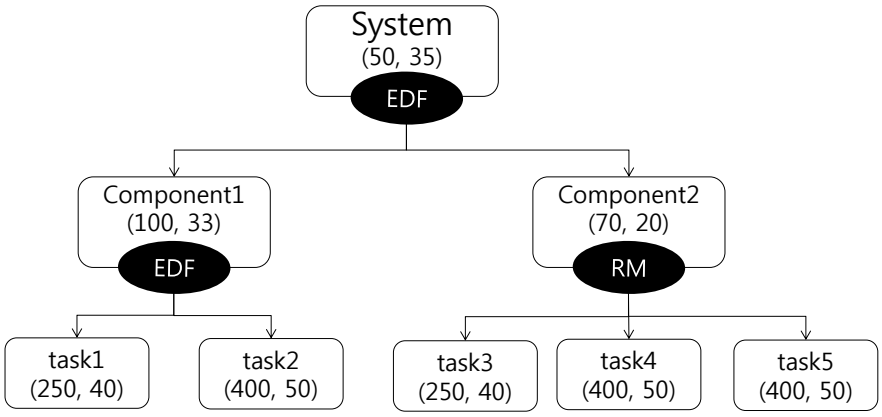


Figure 1: Example of hierarchical scheduling system.

component, including the top level, is schedulable. Thus, for the given timing requirements Component1 and Component2 should be schedulable by the top level System according to the EDF scheduling policy. The tasks task1 and task2 should be schedulable, with respect to the timing requirement of Component1 (100, 33), also under the EDF scheduling policy. Similarly, task3, task4 and task5 should be schedulable, with respect to the timing requirements of Component2, under the RM scheduling policy. The next section presents the compositional analysis of the schedulability of our example.

For a given system structure, we can have many different system configurations. A system configuration consists of an instantiation of the model where each parameter has a specific value. Fig. 1 shows one such instantiation.

3.3. From Timed Automata to Hybrid Automata

The modeling formalisms used in this paper range from classical timed automata to hybrid automata with algorithmic support from the various branches of the tool Uppaal. The classical version of Uppaal offers support for efficient symbolic verification of timed automata [2] and over-approximate verification of stopwatch automata [16]. The branch Uppaal CORA extends the symbolic verification engine of Uppaal to support cost-optimal reachability for priced timed automata [9, 3, 33].

Most recently the branch Uppaal SMC [22, 24] provides highly scalable verification engine for statistical model checking (SMC) for not only the three formalisms above but stochastic hybrid automata in general. In essence, statistical model checking is based on stochastic semantics allowing for the probability of linear time properties to be estimated (or tested) with arbitrary precision and confidence through simulations.

Uppaal SMC thus supports the analysis of stochastic hybrid automata (SHA) [19] that are timed automata whose clock rates can be changed to be con-

stants or expressions depending on other clocks, effectively defining Ordinary Differential Equations (ODEs). This generalizes the model used in previous work [22, 24] where only linear priced automata were handled. The release Uppaal SMC 4.1.18[1] supports fully hybrid automata with ODEs and a few built-in functions (such as `sin`, `cos`, `log`, `exp` and `sqrt`).

3.4. The Uppaal *Tool Family*

In the previous section several Uppaal variants were mentioned. This section aims to give an overview of both the different forms of analysis that the tools in the Uppaal family can provide and the types of models that they can handle.

Tool	Models	Methods
Uppaal	Timed Automata Stopwatch Automata	Model Checking
Uppaal CORA	Priced Timed Automata	Model Checking, Cost-optimal reachability
Uppaal SMC	Stochastic Hybrid Automata	Statistical Model Checking
Uppaal TRON [32]	Network of Timed Automata	Real-time Online Testing
Uppaal TIGA [15]	Timed Game Automata	Controller Synthesis Model Checking
Uppaal Stratego [20]	Stochastic Priced Timed Automata	Model Checking, Statistical Model Checking, Controller Synthesis, Statistical Learning

Table 1: An overview of the different tools, models and methods in the Uppaal tool family.

Table 1 gives an overview of the tools and the types of models that they can operate on. The tools Uppaal and Uppaal SMC are listed here as two different tools, but Uppaal SMC has been integrated into Uppaal such that they are one tool. If certain features are used in the models, then only statistical analysis can be performed on the models. In the analysis sections of this paper both Uppaal and Uppaal SMC are used.

3.5. Hybrid Automata

Intuitively, a hybrid automaton $\mathcal{H}$ is a finite-state automaton extended with continuous variables that evolve according to dynamics characterizing each discrete state (called a *location*). Let X be a finite set of continuous variables. A *variable valuation* over X is a mapping $\nu : X \to \mathbb{R}$, where $\mathbb{R}$ is the set of reals. We write $\mathbb{R}^X$ for the set of valuations over X. Valuations over X evolve over time according to *delay functions* $F : \mathbb{R}_{\geq 0} \times \mathbb{R}^X \to \mathbb{R}^X$, where for a delay d and valuation ν, $F(d, \nu)$ provides the new valuation after a delay of d. As is the case for delays in timed automata, delay functions are assumed to be time

[1] `www.uppaal.org`.

additive in the sense that $F(d_1, F(d_2, \nu)) = F(d_1 + d_2, \nu)$. To allow for communication between different hybrid automata, we assume a set of actions Σ, which is partitioned into disjoint sets of input and output actions, i.e., $\Sigma = \Sigma_i \uplus \Sigma_o$.

Definition 3.1. *A hybrid automaton (HA) $\mathcal{H}$ is a tuple $\mathcal{H} = (L, \ell_0, X, \Sigma, E, F, I)$, where: (i) L is a finite set of locations, (ii) $\ell_0 \in L$ is an initial location, (iii) X is a finite set of continuous variables, (iv) $\Sigma = \Sigma_i \uplus \Sigma_o$ is a finite set of actions partitioned into inputs (Σ_i) and outputs (Σ_o), (v) E is a finite set of edges of the form $(\ell, g, a, \phi, \ell')$, where ℓ and ℓ' are locations, g is a predicate on $\mathbb{R}^X$, action label $a \in \Sigma$ and ϕ is a binary relation on $\mathbb{R}^X$, (vi) for each location $\ell \in L$ $F(\ell)$ is a* delay function*, and (vii) I assigns an invariant predicate $I(\ell)$ to any location ℓ.*

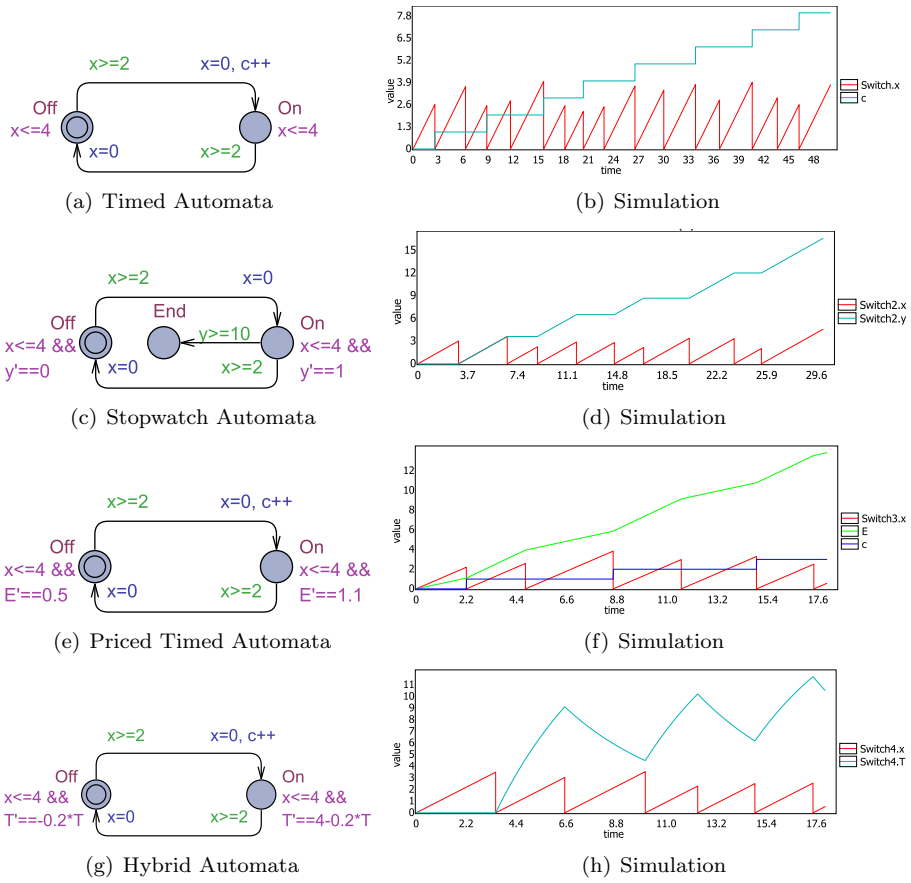
The semantics of a HA $\mathcal{H}$ is a timed labeled transition system, whose states are pairs $(\ell, \nu) \in L \times \mathbb{R}^X$ with $\nu \models I(\ell)$, and whose transitions are either delay transitions $(\ell, \nu) \xrightarrow{d} (\ell, \nu')$ with $d \in \mathbb{R}_{\geq 0}$ and $\nu' = F(d, \nu)$, or discrete transitions $(\ell, \nu) \xrightarrow{a} (\ell', \nu')$ if there is an edge $(\ell, g, a, \phi, \ell')$ such that $\nu \models g$ and $\phi(\nu, \nu')$. We write $(\ell, \nu) \rightsquigarrow (\ell', \nu')$ if there is a finite sequence of delay and discrete transitions from (ℓ, ν) to (ℓ', ν').

In the above definition, we have deliberately left open the concrete syntax for the delay function F as well as guards g, update predicate ϕ and invariant I. For *timed automata* (TA) [2], the continuous variables are simple clocks x where the delay update $F(\ell)$ is given by an implicit rate $x' = 1$. For *stopwatch automata* (SWA), the rate in a location ℓ may be either $x' = 1$ or $x' = 0$ (the latter to be annotated explicitly). For both TA and SWA, guards g and invariants I are restricted to conjunctions of simple integer bounds on individual clocks, and the update predicate are simple assignments of the form $x = e$, where e is an expression only depending on the discrete part of the current state. This restriction ensures decidability and efficiency of model checking in the case of TA and permits efficient over-approximate analysis of SWA.

For *priced timed automata* (PTA) [9, 3, 33], the continuous variables are either simple clocks as in TA or cost-variables for which the delay update is given by an explicit rate $x' = e$ appearing in the invariant of ℓ, where e again is an expression only depending on the discrete part of the current state. PTA guards, updates and invariants may only refer to discrete part or simple clocks – thus the cost-variables cannot affect the behavior of the models but are simple observers. Under these restrictions, cost-optimal (minimal or maximal) reachability is decidable and may be computed exactly and efficiently using symbolic techniques [33].

In the most general case of a hybrid automaton (HA), the delay function F may need to solve a set of ODEs. It is important to note that in specifying the delay function F and the invariant I, the full syntax of UPPAAL expressions – including user-defined functions – is at the disposal. For this class of model only simulation-based techniques are supported.

Example 3.1. *The various extended automata in Fig.2 model various quantitative aspect of a simple* Switch *with two modes* On *and* Off*. Fig. 2(a) is a*



(a) Timed Automata (b) Simulation

(c) Stopwatch Automata (d) Simulation

(e) Priced Timed Automata (f) Simulation

(g) Hybrid Automata (h) Simulation

Figure 2: Timed, Stopwatch, Priced and Hybrid Automata for Switch

timed automaton model of the Switch *using a clock* x *to enforce that the time-separation between mode-switches is between 2 and 4 time-units. In addition an integer variable* c *counts the number of times the* Switch *has been in location* On*. Using the model checker of* Uppaal *it can be verified that the total time until* c *becomes 3 is between 10 and 20 time-units as confirmed by the simulation in Fig. 2(b).*

Fig. 2(c) introduces a stopwatch y *which is running only in location* On*, thus effectively measuring the accumulated residence-time in* On*. Using the over-approximate verification offered by* Uppaal *for stopwatch automata, it can be concluded that within 11 time-units the* Switch *cannot have been in* On *for more than an accumulated time of 10 time-units. This is confirmed by the simulation in Fig. 2(d).*

Fig. 2(e) is a priced timed automaton model of the Switch *with a (single) cost-variable* E *measuring the total accumulated energy consumption during the*

behavior. Here the rate of E is 0.5 in the Off location and 1.1 in the On location. Using the cost-optimal scheduling algorithm of UPPAAL CORA*, it may be computed that most energy-efficient way of having the counter variable c reaching 3 is* 7.4*. Again this finding is confirmed by the random simulation of Fig. 2(f).*

Finally, Fig. 2(g) is a hybrid automaton model of the Switch *with the continuous variable T modeling the temperature. Here the invariants in the locations* On *and* Off *are simple linear differential equations describing the evolution of T. Fig. 2(h) provides a random simulation of the model. For this type of model no exact model checking is offered.*

3.6. Stochastic Hybrid Automata

The stochastic semantics of HA refines the non deterministic choices that may exist with respect to delay, output and next state. For each state $s = (\ell, \nu)$ of a HA $\mathcal{A}$, we shall assume that there exist probability distributions for delays, output as well as next-state:

- the *delay density function*, μ_s over delays in $\mathbb{R}_{\geq 0}$, provides stochastic information for when the component will perform an output, thus $\int \mu_s(t)dt = 1$;
- the *output probability function* γ_s assigns probabilities for resolving what output $o \in \Sigma_o$ to generate, i.e., $\sum_o \gamma_s(o) = 1$;
- the *next-state density function* η_s^a provides stochastic information on the next state $s' = (\ell', \nu') \in \mathbb{R}^X$ given an action a, i.e., $\int_{s'} \eta_s^a(s') = 1$.

For outputs happening deterministically at an exact time point d (or deterministic next states s'), μ_s (η_s^a) becomes a Dirac delta function δ_d ($\delta_{s'}$)[2].

In UPPAAL SMC, uniform distributions are applied for states where delay is bounded, and exponential distributions (with location-specified rates) are applied for the cases, where a component can remain indefinitely in a location. Also, UPPAAL SMC provides syntax for assigning discrete probabilities to different outputs as well as specifying stochastic distributions on next-states (using the function `random(b)` denoting a uniform distribution on $[0, b]$).

Example 3.2. *Under the above stochastic interpretation of timed automata, all of the extended timed automata models of the* Switch *will have the delays in* Off *and* On *being determined by a uniform distribution on the interval* $[2, 4]$*. The various simulations illustrated are obtained using this stochastic semantics. Now using the statistical model checking engine of* UPPAAL *SMC, we may establish a number of interesting performance properties. Using the timed automata model Fig. 2(a) we find that the probability that c becomes 3 before 15 time units is estimated to be in the confidence interval* $[0.419126, 0.518993]$ *with confidence 0.95 in Fig. 2 after some 402 simulation runs. Using the priced timed automaton*

[2] which should formally be treated as the limit of a sequence of delay density functions with decreasing, non-zero support around d.

model of the Switch*, we may estimate the expected energy consumption before* c *becomes 3 to be in the interval* $[11.0389 - 0.34824, 11.0389 + 0.34824]$ *with confidence 0.95 within 36 runs. Finally, using the hybrid automaton model, it may be established that the probability that the temperature drops below 5 degrees after 10 time-units is in the interval* $[0.104583, 0.204489]$.

In general a model comes as a network of HAs. For networks, the stochastic semantics is based on the principle of independence between components under the assumption of input-enabledness. Repeatedly, each component decides on its own, based on a given delay density function and output probability function, how much to delay before outputting and what output to broadcast at that moment. Obviously, in such a race between components the outcome will be determined by the component that has chosen to output after the shortest delay: the output is broadcast and all other components may consequently change state.

For more in-depth description of the semantic foundation of Uppaal SMC we refer the reader to [19]. For concrete syntax of models and queries we refer to the home-page of Uppaal.

4. Compositional Schedulability Analysis

In this section, we describe our compositional analysis technique using the running example of Fig. 1. In order to design a framework that scales well for the analysis of larger hierarchical scheduling systems, we have decided to use a compositional approach. Fig. 3 shows how the scheduling system, depicted in Fig. 1, is analyzed using three independent analysis processes. These processes can be performed in any order, or better yet in parallel.

The schedulability of each component, including the top level, is analyzed together with the interface specifications of the level directly below it. Accordingly, the system is too complex to be analyzed at once. In Fig. 3, the analysis process A consists of checking whether the two components Component1 and Component2 are schedulable under the scheduling policy EDF. In this analysis process, we only consider the interfaces of components in the form of their execution time (budget) and period, so that we consider the component as an abstract task when performing the schedulability analysis of the level above it. In this way, we consider the *component-composition* problem similarly to [40] but using a non deterministic supplier model for the interfaces. When performing an analysis process like A1, the resource supplier is not part of the analysis. In order to handle this, we add a non deterministic supplier to the model. The supplier will guarantee to provide the amount of execution time, specified in the interface of Component1, before the end of the component period. We check all possible ways in which the resources can be supplied to the subsystem in A1. The supplier of each component provides resources to the child entities of that component in a non deterministic way. During the analysis of A1, the supplier non deterministically decides to start or stop supplying, while still guaranteeing

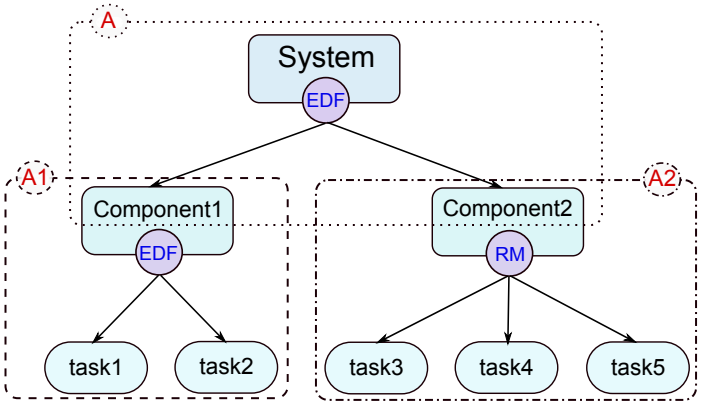


Figure 3: Compositional analysis

to provide the required amount to its sub-entities before the end of the period. The analysis A2 is performed in the same way as A1.

Our compositional analysis approach results in an over-approximation i.e., when performing the analysis of a subsystem, we over-approximate the behavior of the rest of the system by considering all potential executions and more as some of these executions may be unrealizable if the system components run together. This can result in specific hierarchical scheduling systems that could be schedulable if one considers the entire system at once, but that is not schedulable using our compositional approach. We consider this fact as a design choice which ensures separation of concerns, meaning that small changes to one part of the system do not effect the behavior of other components. In this way, the design of the system is more stable which in turn leads to predictable system behavior. This over-approximation should not be confused with the over-approximation used in the verification algorithm inside the UPPAAL verification engine (Section 6.4). The result can either be *true (false)* or *maybe-not (maybe)*, and in the case of *true (false)* the result of the analysis is conclusive and exact.

Thanks to the parameterization of system entities; scheduling policies, preemptiveness, execution times, periods and budgets can all easily be changed. In order to estimate the performance and schedulability of our running example, we have evaluated a number of different configurations of the system. This allows us to choose the best of the evaluated configurations of the system.

5. Formal Specification and Analysis

This section introduces the specification of our system units: tasks, resource model, scheduling algorithm and components as well as the semantics defined at the component level. First of all, we use the following notation:

- $\mathcal{T}$ is the set of all tasks,
- $\mathcal{C}$ is the set of all components.

5.1. Syntax Representation

In this section, we define the syntax we use to formally describe the components of a scheduling system. A task is defined by three timing properties; period, execution time and deadline, as well as its current status. The behavior of all tasks, based on their parameters, is given by the semantic rules defined in Section 5.2.

Definition 5.1 (Task). *A task $T_i \in \mathcal{T}$ is a tuple $(prd, e, d, status)$ where:*

- *prd is the task period.*
- *e is the execution time.*
- *d is the task deadline relative to the start of period, with $e \leq d \leq prd$.*
- *$status$ is one of the following properties $\{Ready, Running, Done\}$.*

The status *Ready* means that the task is ready and waiting to be scheduled, whereas the status *Running* indicates that the task is using the resource. Once the execution of a task is over, i.e., the execution time constraint is successfully satisfied, the task status is updated to *Done* waiting for the next period. As we do not consider offsets in the specification of tasks, all the tasks are initially *Ready*.

In any hierarchical scheduling context, the behavior of the resource supply model does not depend at all on the resource demanding components. Any resource model behaves in the way that it supplies resource for an amount of time then stops supplying for a given time interval. We are giving the abstract behavior of the resource models without detailing the specific characteristics like preemptiveness and single-/multi-core. Accordingly, we define the abstract behavior of any periodic resource model by a transition system consisting of two states: *Supply* and *NonSupply*. For compositionality purposes, the triggering of transitions between those states is non deterministic, i.e., the resource supply is non deterministic because we do not know when a parent level component supplies the resource to its child components. We assume that the resource model is initially at *NonSupply*.

Definition 5.2 (Resource model). *A resource model R is a timed automaton $\langle \{Supply, NonSupply\}, NonSupply, X, I, \rightarrow \rangle$ where $\{Supply, NonSupply\}$ is the set of locations, NonSupply is the initial location, X and I are the same as for hybrid automata, and $\rightarrow \subseteq \{Supply, NonSupply\} \times \{Supply, NonSupply\}$ is the transition relation.*

Such a resource model can be viewed as an implementation of the component interface $\mathcal{I}$, by which an amount of resource is guaranteed to be supplied but we do not know when the resource will be supplied.

To arbitrate the use of a resource between the competing tasks, we use a scheduling function. Basically, the scheduling function determines which task among the ready tasks queue has priority over the others at any point in time. The function is described so abstractly in order to be able to model any real scheduling algorithm.

Definition 5.3 (Scheduling function). *A scheduling function establishes a order on tasks:* $Sched : \mathcal{T} \times \mathcal{T} \times \mathbb{R}_{\geq 0} \rightarrow \mathcal{T}$, *where* $\mathbb{R}_{\geq 0}$ *is the time domain.*

Our abstract scheduling function *Sched* compares two tasks and determines the one with highest priority at any given point in time. *Sched* computes the release time of a task execution for the *current* period (instance) of such a task, based on the task period and the current system configuration (queue), but it cannot predict the release time of a future execution until the period of such an instance is reached. We achieve the comparison of a set of tasks through pairwise comparisons. *Sched* can be instantiated to fit both static and dynamic priority scheduling algorithms. In case of static priorities, such as FPS (Fixed Priority Scheduling), each task has always the same priority (and thus the same order) regardless of the time instant when the comparison is made. For example, given two tasks T_1 and T_2 having priorities 2 and 3 respectively (higher number means higher priority). Thus, whatever the instant x if both tasks are ready then $Sched(T_1, T_2, x) = T_2$.

However, in case of dynamic priority scheduling, such as EDF (Earliest Deadline First), two tasks may have different orders at different time instants. For example, initially task T_1 has a shorter time to deadline than T_1 but it is not ready. Thus, $Sched(T_1, T_2, 0) = T_2$ so that task T_2 is scheduled first. Once task T_1 becomes ready, let's say at time instant 4, and since it has shorter deadline it will have priority over T_2, i.e., $Sched(T_1, T_2, 4) = T_1$, thus T_1 preempts T_2 at time 4. *Sched* can be instantiated for EDF in the following:

$$Sched(T_i, T_j, t) = T_i \textbf{ if } \begin{cases} status_i \neq done \textbf{ and } status_j = done \\ \textbf{or} \\ ((t/prd_i) * prd_i + d_i) \leq ((t/prd_j) * prd_j + d_j) \end{cases}$$

In Section. 5.2, function *Sched* will be used with quantifiers such that the task with highest priority will be found. In a practical implementation, the scheduling function operates on the whole queue of ready tasks using its particular scheduling principle.

We define components as scheduling units which can be any level in the hierarchy. Since each component has its own timing requirements, we dedicate to each component a resource model stating how the current component is served by its parent component. Therefore, we extend the component definition given in Section 3.1 by the resource model implementing the component interface.

Definition 5.4 (Component). *A component is a tuple* $(W, R, Sched)$ *where:*

- *$W \subseteq \mathcal{T}$ is the workload defined as a set of tasks.*
- *R is the resource model supplying the component tasks.*
- *Sched is the scheduling function, which behaves according to a scheduling policy (s of the component description given in Section. 3.1).*

For hierarchical components, each task in the workload W represents the interface of one of the child components. Of course, the deadline in this case will be the same as period and both deadline and status attributes will be omitted in the interface as they are not necessary when analyzing the corresponding child component.

5.2. Semantics of Components

In this section, we define the semantics of individual components according to the non deterministic behavior of the resource model. In order to capture the state of component configuration during execution, we introduce the following:

- Variable called loc_R to store the current location of the resource model of the component in question.
- For each task $T_i \in W$, we introduce three variables x_i, x_i^e and x_i^r.
  - Variable x_i will store the point in time where the current period has started in order to compute the time left to deadline.
  - Variable x_i^e will be used to store the remaining execution time for the current period.
  - Variable x_i^r will be used to keep track of the starting time of the current supply for task T_i.
- Besides, we dedicate a particular variable clk to keep track of the global time of the component.

The semantics of individual components will be given in terms of timed transition systems (TTS). Basically, a timed transition system $\langle S, s^0, \rightarrow \rangle$ is given by a set of states S, the initial state $s^0 \in S$ and a transition relation $\rightarrow \subseteq S \times S$. Each transition of the TTS can be discrete or continuous according to its label. However, since we are mainly focusing on the resource utilization and reachability of deadlock states we do not consider transition labels.

Since the resource model is intentionally made to be non deterministic for compositionality purposes, we do not know how long the resource is available for each supply, i.e., once the resource model moves to state *Supply* the delay at that state (supply time) is unknown as the transitions between states *Supply* and *NonSupply* are non deterministic. Thus, we describe the continuous transitions (delays) of the semantics with the assumption that the resource is continuously available during each delay transition.

Definition 5.5 (Component Semantics). *Given a component $C = (W, R, Sched)$, the semantics of C is given by a timed transition system (TTS) $\langle S, s^0, \rightarrow \rangle$ where:*

- $S \subseteq \mathbb{R}_{\geq 0} \times 2^{P(W)} \times V \times V_e \times V_r \times \{loc_R\}$ *is a state space,*
  - $V = \{v \mid \forall T_i \in W \; v : x_i \mapsto \mathbb{R}_{\geq 0}\}$ *is the set of valuations of variables x_i corresponding to the tasks of W.*
  - $V_e = \{v \mid \forall T_i \in W \; v : x_i^e \mapsto \mathbb{R}_{\geq 0}\}$ *is the set of valuations of variables x_i^e corresponding to the tasks of W.*
  - $V_r = \{v \mid \forall T_i \in W \; v : x_i^r \mapsto \mathbb{R}_{\geq 0}\}$ *is the set of valuations of variables x_i^r of the component tasks.*
- $s^0 = (0, \{T_1, .., T_{|W|}\}, \{\forall i \; v(x_i) = 0\}, \{\forall i \; v(x_i^e) = e_i\}, \{\forall i \; v(x_i^r) = 0\}, Non-Supply)$ *where 0 is the initial global time; $\{T_1, .., T_{|W|}\}$ is the identifiers of component tasks such that $\forall i \; T_i.status := Ready$ as we do not consider the offset attribute; NonSupply is the initial state of the resource model R.*

  Since each of the tasks does not initially start running yet, the amount of execution time elapsed is set to zero ($x_i^r := 0$) and the remaining execution time is the whole requirement e_i ($x_i^e := e_i$). Moreover, we store the current time (0) in each variable x_i as the release time of the current period that will be used to compute the relative deadline.
- *the transition relation $\rightarrow \subseteq S \times S$ is given by the following 7 rules:*

$$\textbf{Done} : \frac{s(T_i.status) \neq Done, s(clk) - s(x_i) \leq s(T_i.d) \quad s(x_i^e) - (s(clk) - s(x_i^r)) = 0}{s \rightarrow s[T_i.status \mapsto Done]}$$

The rule **Done** describes the successful termination of a task execution without missing the deadline. $s(x_i^r)$ is the time instant when the task T_i starts running for the current supply, $s(x_i^e)$ is the remaining execution time of T_i at time $s(x_i^r)$, $s(clk)$ is the current time and $s(x_i)$ is the time point when we started counting for the relative deadline. For any existing task T_i in the current component state (queue), once the execution time constraint is satisfied $s(x_i^e) - s(clk) - s(x_i^r) = 0$ and the deadline is not missed ($s(clk) - s(x_i) \leq s(T_i.d)$) the task will be declared as successfully done for its current period.

In order to avoid updating the status of a task which is already done, we check first whether the task status is not updated yet ($s(T_i.status) \neq Done$).

$$\textbf{New\_Period} : \frac{s(T_i.status) = Done, s(clk) - s(x_i) = s(T_i.prd)}{s \rightarrow s[T_i.status \mapsto Ready, x_i := clk, x_i^e := T_i.e]}$$

The rule **New_Period** describes the release of a new period for any task once 1) it has successfully achieved the execution of its current period, i.e., without missing the deadline, and 2) the current task period is expiring ($s(clk) - s(x_i) = s(T_i.prd)$). When a task T_i starts a new period, its status is updated to *Ready*, the corresponding variable x_i is reinitialized to the current global time ($x_i := clk$) and the remaining execution time (x_i^e) is also reset to the original execution time requirement e of task T_i.

$$\textbf{Missed}: \frac{\begin{array}{l}\exists T_i \mid s(clk) - s(x_i) > s(T_i.d) \\ \wedge \begin{cases} s(x_i^e) > 0 \wedge s(T_i.status) = Ready \\ \vee \\ (s(clk) - s(x_i^r) < s(x_i^e)) \wedge s(T_i.status) = Running \end{cases}\end{array}}{s \rightarrow Deadlock}$$

The rule **Missed** describes the deadline miss of a task which is either running or waiting to be scheduled. A deadlock occurs when a deadline is reached ($s(clk) - s(x_i) > s(T_i.d)$) and the remaining execution time is greater than zero. The rule has two cases for *Running* and *Ready* respectively. In the simple case of *Ready*, we simply check that the remaining execution time is greater than zero ($s(x_i^e) > 0$). In the case where the task is *Running*, we check that even with the current supply that started at time $s(x_i^r)$ the remaining execution time $s(x_i^e)$ has not been fulfilled.

The current interpretation of missing a deadline is modeled as a deadlock to fit for hard real-time systems. The semantics could be changed such that the status of the individual task is just updated to *Missed* in case of soft real-time systems.

$$\textbf{Run}: \frac{\begin{array}{c} s(loc_r) = Supply, \exists T_i \mid s(T_i.status) = Ready \wedge \\ \forall T_j \in W \quad Sched(T_i, T_j, s(clk)) = T_i \end{array}}{s \rightarrow \quad s[T_i.status \mapsto Running, x_i^r := clk]}$$

The rule **Run** describes when a task starts running. In fact, if the resource is available ($s(loc_r) = Supply$), a task is ready and determined to be prioritary at the current time ($s(clk)$) by the scheduling function *Sched*, then such a task T_i switches its status from *Ready* to *Running* and starts its execution immediately. We store the current time *clk* in x_i^r in order to be able to determine when the execution time of task T_i will expire. Namely, the execution time of a task expires when the global time *clk* reaches the time instant ($x_i^r + x_i^e$) while the task is continuously running. The scheduled task runs until it is either done, preempted or misses its deadline. The rule **Run** will only be applied when no task is currently running. In all other cases, one of the two preemption rules will be applied.

$$\textbf{Delay}: \frac{\begin{array}{c}\exists T_i \mid s(T_i.status) = Running \;\; \exists \delta \leq s(x_i^e) - (s(clk) - s(x_i^r)) \mid \\ \forall x \in [0,\delta] \; s(loc_r) = Supply \wedge \forall T_j \in W \quad Sched(T_i, T_j, x) = T_i\end{array}}{s \rightarrow \quad s[clk := clk + \delta]}$$

The rule **Delay** describes a delay δ (continuous transition) of the semantics. Such a delay occurs when 1) a task T_i is currently running (at time $s(clk)$), and 2) none of the other tasks can preempt it during a time interval $[0, \delta]$ for which 3) the resource is continuously available. Obviously δ could be any value, bounded by the actual remaining execution time of T_i, satisfying the aforementioned conditions.

In fact, this is not the unique case where delays occur. Delays may occur if all of the tasks are neither *Running* nor *Ready*, so that the corresponding TTS semantics delays until one of the tasks becomes *Ready*. However, since we are ultimately focusing on the resource utilization we only presented the case of delays where the resource is being used.

$$\textbf{Preempt1}: \frac{\begin{array}{c}s(loc_r) = Supply, \\ \exists T_i \; T_j \;\; \mid s(T_i.status) = Ready \wedge s(T_j.status) = Running \wedge \\ Sched(T_i, T_j, s(clk)) = T_i\end{array}}{\begin{array}{l}s \rightarrow \quad s[T_i.status \mapsto Running, T_j.status \mapsto Ready, \\ x_j^e := x_j^e - (clk - x_j^r), x_i^r := clk]\end{array}}$$

The rule **Preempt1** describes the preemption of a task T_j by another task T_i having priority at the current time $s(clk)$, i.e., $Sched(T_i, T_j, s(clk)) = T_i$. Of course, we keep track about the remaining execution time of the preempted task T_j and store the current time as a supply release time ($x_i^r := clk$) of the running task T_i while updating the status of tasks to *Ready* and *Running* respectively.

$$\textbf{Preempt2}: \frac{\exists T_i \mid s(T_i.status) = Running \wedge s(loc_r) = NonSupply}{s \rightarrow s[T_i.status \mapsto Ready, x_i^e := x_i^e - (clk - x_i^r)]}$$

The rule **Preempt2** describes the preemption of the execution of a task T_i due to the non availability of the resource. In fact, once the resource model moves to state *NonSupply* while a task is running the former gets preempted. Similar to rule **Preempt1**, we store the remaining execution time and update the status of the preempted task.

Based on the TTS semantics defined above, the schedulability of a component can simply be checked as a reachability property using the following CTL [3] query: $\forall$ `[] not Deadlock`.

[3] CTL stands for Computation Tree Logic.

6. Modeling and Analysis using UPPAAL

In this section we detail the actual Timed Automata models we used for the schedulability analysis of hierarchical scheduling systems. The use of statistical model checking (SMC) to disprove the schedulability of systems is presented together with a statistical method for estimating budgets of the individual components. For absolute certainty, symbolic model checking techniques will be used to make sure that the results obtained using SMC are consistent.

The purpose of modeling and analyzing hierarchical scheduling systems is to check whether the tasks nested in each component are schedulable, with respect to resource constraints given by the component. This means that the minimum budget of a component supplier, for a specific period, should satisfy the timing requirements of the child tasks. Namely, a scheduling unit [40] consists of a set of tasks, a supplier and a scheduler, in [40] known respectively by the terms workloads, resource model and scheduling policy.

We revisit the running example shown in Fig. 1, which is built on the instances of four different Uppaal timed automata templates: 1) non deterministic supplier 2) periodic task 3) CPU scheduler (EDF, RM), and 4) resource manager. Similarly to [21], we also use broadcast channels where no sender can be blocked when performing a synchronization. We use stopwatches, writing $x' == e$ to specify a clock x that can only progress when e evaluates to 1. Uppaal also allows for clocks to progress with other rates than 1. This will be used to measure the energy consumption in Fig. 15(a).

6.1. Stochastic Resource Supplier Model

In this section, we argue why we use a stochastic supplier model in our compositional analysis. The hierarchical scheduling system structure is a set of scheduling components, each of which includes a single specific scheduling algorithm and a set of entities (either tasks or components). The resource model we consider to implement the interface of a component is the Periodic Resource Model (PRM), which provides a specific amount of resource to a workload every period [39]. The PRM represents the interface requirement between a set of tasks and a higher level scheduler. The high level scheduler is referred to as Supplier, which satisfies the interface requirement given by the periodic resource model. To represent the behavior of the supplier, based on the interface requirement, we use a PRM given in terms of a hybrid automaton.

It is necessary to consider the interrupted behavior of a component by the other concurrent components within the same system when analyzing this component individually (in a compositional manner). However, it is not trivial to capture the interrupting behavior of the other components that influence the component under analysis. For this reason, we introduce a stochastic supplier to model all scenarios that the component under analysis can run. Such a permissive model simulates the influence of the other system components on the execution of the component under analysis, so that all potential preemptions can be captured.

As mentioned earlier, the stochastic supplier is a resource model that allocates resources to the component. The scheduling policy within the component then allocates the resources to tasks within that component. It also abstracts the possibility that a task from another part of the system (not part of the current analysis process) could preempt the execution of tasks of the current component.

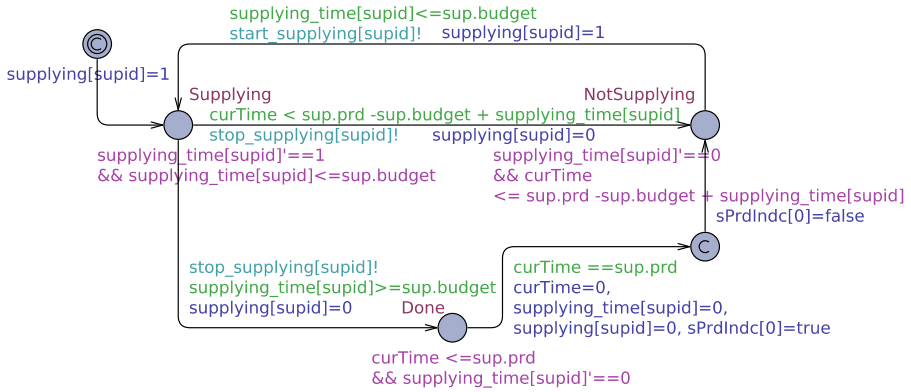


Figure 4: Stochastic resource supplier template

Fig. 4 depicts our UPPAAL template model of the stochastic supplier. Basically, the supplier allocates a resource, denoted by rid, to a set of tasks according to the timing requirements (interface) of the component, given in Listing 1.

Listing 1: Component's interface requirements

```
typedef struct {
  time_t        prd;
  time_t        budget;
} sup_t;
```

A resource rid can represent a processing unit (CPU) or any other system resource, represented in the model by a semaphore. prd is a period and budget is the amount of resource to be provided during each period. The supplier assigns the budget amount of resources to tasks in task[tid_t]. In this model, supplying_time[supid] (supid is the supplier identifier) represents the effective duration when the supplier provides a resource. start_supplying[supid] and stop_supplying[supid] are broadcast channels that notify tasks of the beginning and completion of the supply. curTime denotes the time elapsed since the beginning of the supplier's resource supply. supplying[supid] contains the supplier's status, 0 (not supplying) or 1 (supplying).

The transitions between NotSupplying and Supplying in Fig. 4 are non deterministically taken until the budget is fulfilled ($supplying\_time[supid] >= sup[supid].budget$). The supplier stays at location Supplying to fully provide the remaining amount of resource when the slack time ($sup.prd - sup.budget +$

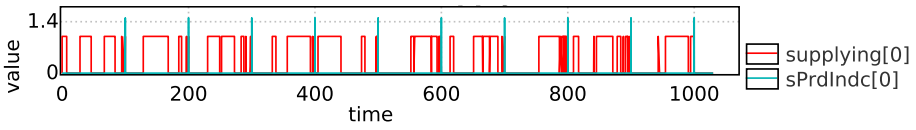


Figure 5: Simulation of the resource supply

supplying_time[*supid*]) is over. Then, it moves to location Done and waits for the expiry of the current period before starting a new period.

Fig. 5 shows one particular resource supply pattern for Component1 of Fig. 1. supplying[0] in red denotes whether the supplier allocates resources or not, thus it maintains 1 when the supplier is providing resources. In this example, the supply of resources is maintained for 33 time units. One can notice that Fig. 5 shows a non deterministic supply, in which the values of supplying[0] are irregular in behavior within the supplier's period (blue sparked line).

6.2. Task model

We consider a finite set of tasks and refer to them as $T_1, T_2, \ldots, T_n$. Each task is defined by the timing attributes given in Listing 2.

Listing 2: Task's execution attributes

```
typedef struct {
  rid_t          cpuid;                // Dedicated CPU Id
  pri_t          pri ;                 // Priority
  time_t         initial_offset    ;   // Initial   offset
  time_t         offset ;              // Period offset
  time_t         min_period;           // Minimum period
  time_t         max_period;           // Maximum period
  time_t         deadline ;            // Deadline
  time_t         bcet[freqmode_arr];   // Best-case execution time
  time_t         wcet[freqmode_arr];   // Worst-case execution time
  bool           preemptive;           // Preemptability
} task_t ;
```

cpuid is a CPU identity that is assigned to the execution of a task (prepares the framework to be applied in a multi-core setting). pri is a task priority. initial_offset is an offset for the initial release of the task, and offset represents the offset time of each period. A task has also best-case execution time (bcet) and worst-case execution time (wcet). It is periodically instantiated between minimum (min_period) and maximum (max_period) period lengths, and the period is regular if the minimum and maximum values are the same. Moreover, a task can be characterized individually by a preemptability (preemptive) stating whether task execution can be preempted or not. Following the same principle, a resource can be set to be non preemptive so that any task using the resource can never be preempted while it is using the resource set to be non preemptive. The timing attributes above are given as a structure associated to a timed automaton template.

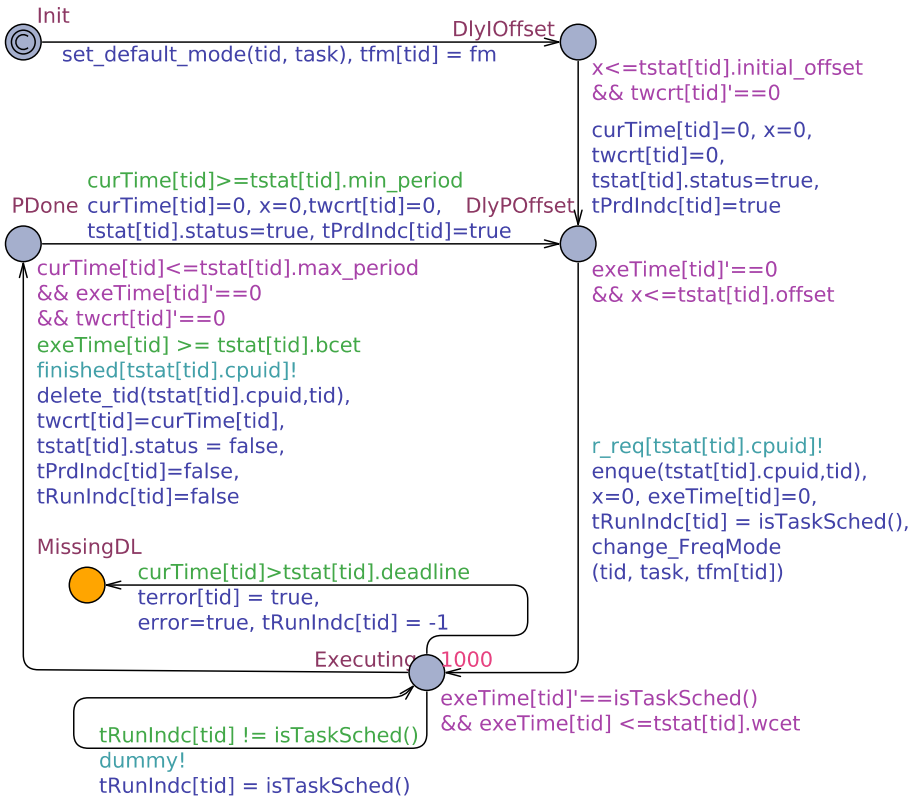


Figure 6: Task template

A periodic task model is given by the template shown in Fig. 6. The followings are clocks and variables used for the task template:

- curTIme[tid] (clock): it stores time elapsed since the execution of the task started. It keeps on running until the task starts a new period.
- exeTime[tid] (clock): it stores the execution time during which the task has already acquired the resource.
- twcrt[tid] (double): this variable stores the worst-case response time of the task for each period.
- tPrdIndc[tid] (bool): this is an indicator used to produce the trace information by stating when a task is released and when it finishes its execution.
- tRunIndc[tid] (bool): this indicator is used to produce the trace information about when a task is running.

From location Init, the task sets its default execution attributes to task's execution context variable structure tstat[tid] using the function set_default_mode(). The locations DlyOffset and DlyPOffset delays the execution of the tasks individually for an initial offset (tstat[tid]. initial_offset) and a period offset (tstat[tid]. offset) respectively. On the transition to Execution, the task requests the CPU resource from the scheduler of the CPU cpuid through the function enque() by feeding its id. If there is a need to change the CPU speed (frequency), one can update the variable tfm[tid], then the function change_FreqMode() updates the corresponding execution attributes according to the updated CPU speed. At location Execution, the task executes its job for a specified execution time which is between bcet and wcet. The execution time is measured by the stopwatch exeTime[tid], of which the progress depends on the following condition:

Listing 3: isTaskSched() function

```
bool isTaskSched() {
  return (rq[tstat[tid]. cpuid]. element[1] == tid && supplying[tstat[tid]. cpuid]);
}
```

cpuid is a CPU id, and tid contains the task id which is scheduled to use the CPU. The UPPAAL user-defined function isTaskSched() returns 1 or 0 according to whether 1) the corresponding task is scheduled or not and 2) the resource is available to use for the task's parent component. Thus, exeTime[tid] increases only when isTaskSched() returns 1. When the current time denoted by curTime[tid] is over the deadline, the task joins location MissedDL. If the execution time of a task is fulfilled, the task finalizes its job by retrieving its id from the CPU queue via delete_tid() and joining the location PDone .

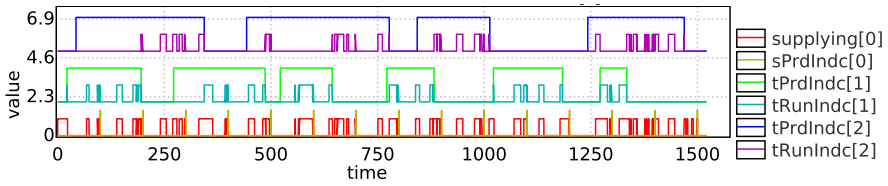


Figure 7: Simulation of the task behavior

The task is scheduled, according to its priority, by a scheduling algorithm implemented as a UPPAAL template [4]. Again, a task can execute only when it is scheduled to use CPU and the supplier of its parent component is currently providing the resource. Fig. 7 shows the timed behavior of task1 and task2; tRunIndc[1] and tRunIndc[2] which toggles according to the resource supply (supplying[0]) from the supplier. The graphs are at the bottom when the supplier

[4] We consider Fixed Priority Scheduling (FPS), First In First Out (FIFO) and Earliest Deadline First (EDF) scheduling policies. Our UPPAAL implementation of these algorithms is available on `http://people.cs.aau.dk/~ulrik/submissions/771691/models.zip`

stops supplying the resource, their corresponding tasks are not scheduled to use CPU, or complete executing within their periods. tPrdIndc[1] and tPrdIndc[2] denote the tasks' periods, and sPrdIndc[0] denotes the supplier's period.

6.3. CPU Resource Model and Scheduling

Fig 8 and Fig 9 show respectively the CPU resource manager template and EDF scheduling algorithm template. These two templates behave like a function. They process and return data instantaneously after they receive a CPU scheduling request. Listing 4 shows the structure of the CPU queue used by the resource manager.

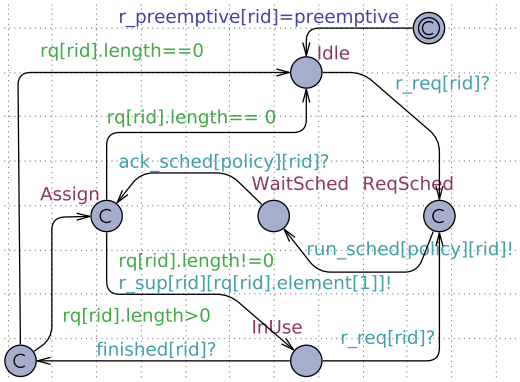


Figure 8: CPU manager template

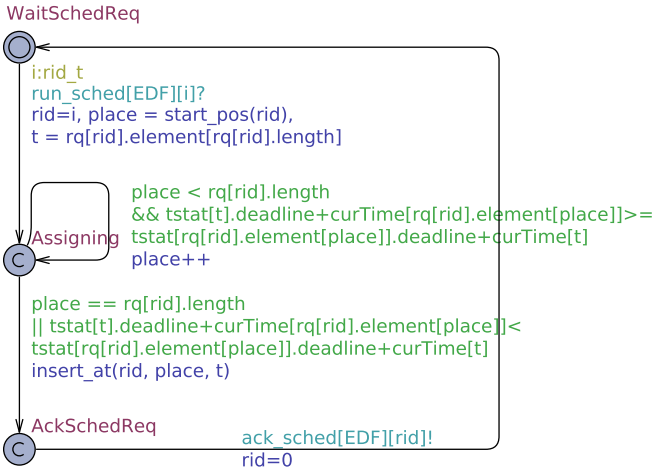


Figure 9: EDF scheduler

Listing 4: CPU queue structure

```
typedef struct {
  int [0, tid_n +1]   length;
  tid_t               element[tid_arr ];
} queue_t;
```

In fact, the CPU resource manager shown in Fig. 8 receives a scheduling request (r_req[rid]) from a task, and requires a scheduling algorithm to select the highest priority task via channel run_sched[policy][rid]. The scheduling algorithm model of Fig. 9 selects the highest priority task and places it as the first element of the CPU queue. The scheduling algorithm model acknowledges the CPU scheduling request via channel ack_sched[policy][rid] after the selection of the highest priority task. Then, the CPU resource manager notifies the tasks in the CPU queue of the end of the CPU scheduling via channel r_sup[rid][selected_taskid].

Thanks to the UPPAAL instantiation mechanism, our system structure can easily be reconfigured. As mentioned earlier, we have modeled each system entity (task, resource, supplier, scheduling policy) by a template so that if, for example, we need to use a scheduling policy instead of another one, we just replace the scheduling policy name in the system instantiation.

6.4. Symbolic Model Checking

In this section, we explain how to check the schedulability using the symbolic reachability engine of UPPAAL. We consider the system with various configurations in terms of preemption, scheduling policy, etc.

Let us start with an illustration of the schedulability analysis of Component1, depicted in Fig. 1. The schedulability of a component is verified with respect to the following safety property:

$$\text{A[] error !=true}$$

Here, the global variable error is a Boolean variable that will be updated to true whenever a task misses its deadline. Thus, this property expresses the absence of deadline violation (i.e., all tasks are schedulable). To check the schedulability of individual tasks, the following query can be used:

$$\text{A[] terror[tid] !=true}$$

For a given supplier with a timing requirement and preemptiveness of the CPU, the verification results of Component1 consisting in task1 (250, 40) and task2 (400, 50) are stated in Table 2.

For the same task set under the EDF scheduling policy, the minimal budget in our verification framework can be greater than the optimal budget of the supplier given in [40]. One of the reasons is that the supplier behaves non deterministically. The fact that UPPAAL uses an over-approximation technique to analyze models containing stop-watches leads to our framework also being an over-approximation. This results in the answer *maybe-not* to some of our

Table 2: Budget evaluation based on scheduling policy and preemptiveness.

Component1	(100, 32)	(100, 33)	(100, 44)	(100, 45)	(100, 46)
	P.	P.	P.	NP.	NP.
EDF	maybe not	Safe	Safe	maybe not	Safe
RM	maybe not	maybe not	Safe	maybe not	Safe

P. : Preemptive; NP.: Non preemptive

verification attempts. In the result shown in Table 2, we use the same task set as in [40] where the authors report that the optimal budget is 33 for the EDF scheduling policy, and the minimal budget we have computed to satisfy the same task set by symbolic model checking is also 33. The minimal budgets we have computed, for RM scheduling and the same task set, are also the same as the budgets presented in [40].

In order to obtain the upper bound on the worst-case response time (WCRT) of tasks, with respect to the EDF policy and a preemptive resource model, we check the following property:

sup: twrts[1], twrts[2]

where the twrts[1] and twrts[2] are stopwatches that are increasing while the corresponding tasks are running. sup is a UPPAAL keyword that refers to a function returning the supremum of the expressions (maximal values in case of integers; upper bounds, strict or not, for clocks). The verification results in twrts[1] $\leq$ 239 and twrts[2] $\leq$ 391, signifying that the WCRTs of each task is less than or equal to 250 and 450, respectively. So none of the tasks miss their deadline.

6.5. Statistical Model Checking

As stated in [21], the use of stop-watches in UPPAAL leads to an over-approximation which guarantees that safety properties are valid but reachability properties could be spurious. Thus, symbolic model checking cannot disprove whether tasks are schedulable but only prove when they are schedulable. For that reason, we apply statistical model checking (SMC) to disprove the schedulability and estimate the minimum budget of the supplier with respect to a specific period.

SMC is a simulation-based approach which estimates the probability for a system to satisfy a property by simulating and observing some of its executions, and then applies statistical algorithms to obtain the result [23]. In this section, we show a way not only of checking schedulability but also to reason about the execution of tasks. To estimate the probability of a component's missed deadline, we use the following UPPAAL SMC query:

Pr[<=runTime](<> error)

where runTime is a simulation time for which a simulation is conducted, and the property <> error implies that a missed deadline of tasks happens during the

simulation. So that the query states the possibility that any task misses the deadline during the simulation.

Table 3: Probability of error. Estimated with 95% confidence.

(100, 31)	(100,32)	(100, 33)	(100, 34)	(100,35)
[0.9410,0.9610]	[0.6911,0.7110]	[0,0.019956]	[0,0.019956]	[0,0.019956]

Table 3 shows the analysis result of Component1 with EDF scheduling policy with 100,000 time units of simulation time according to different budgets of the supplier. The SMC computed the mentioned results with a certain level of confidence and precision, i.e., each result is given as an interval. However, if the lower bound is strictly positive, it guarantees that the checker found at least one witness trace where a task missed its deadline [21]. One may remark that the probability of tasks missing their deadline is much higher when the supplier budget is too small. Note that the possibility that tasks will miss their deadline is between 0.9410 and 0.9610 for the supplier timing requirement (100,31) of our example. This counter-example can be used to estimate the necessary budget for a component, and that will be explained in the following section.

One distinguished advantage of our model-based approach over classical analytical methods is to provide a counter-example that disproves the schedulability of a component. To visualize a witness of the deadline violation, we can request the checker to generate random simulation runs and show the value of a collection of expressions. For example, run the following query on the system:

simulate 100[≤2000]{supplying[0], sPrdIndc[0]*1.5, 2+tPrdIndc[1]*2
, 2+tRunIndc[1], 5+tPrdIndc[2]*2, 5+tRunIndc[2]}:1:error

This query asks the checker to simulate randomly the system execution until the condition error becomes satisfied, and to generate the task status and the accumulated amount of the resource used by the two tasks. Such a counter-example is not easy to obtain by the existing analytical methods because one needs to consider the task behavior and only only the timing requirements.

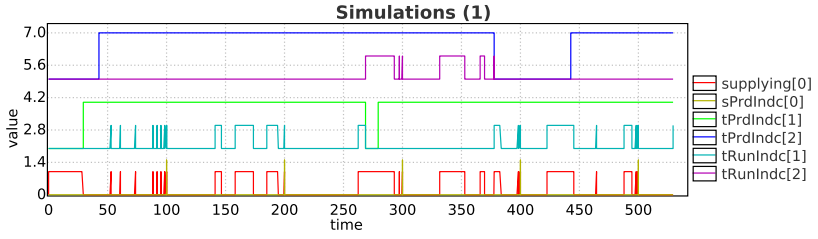


Figure 10: Non schedulable tasks: task1 misses its deadline at time 529.409

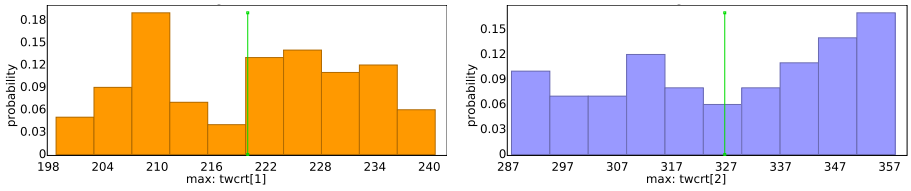
Fig. 10 shows a counter-example where task1 misses the deadline, visualizing the running status of tasks (tRunIndc[1] and tRunIndc[2]) within individual periods

in a Gantt chart. One can remark that task1 stops at time 529.409 within its second period. In this way, one can figure out which task misses the deadline and why by means of UPPAAL SMC.

We apply the following queries for different supplier requirements to generate the probability distribution of the worst-case execution time of tasks:

$$\mathsf{E[<=\ runTime;runCnt]\ (max:\ twcrt[1])}$$
$$\mathsf{E[<=\ runTime;runCnt]\ (max:\ twcrt[2])}$$

where runTime denotes the simulation time and runCnt the simulation count.



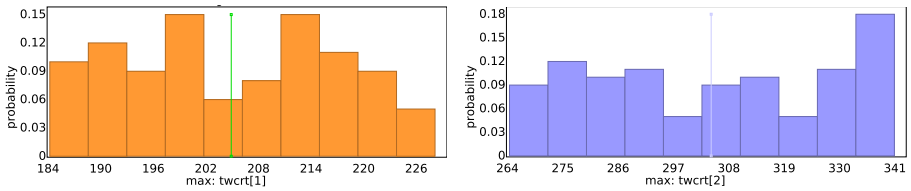
(a) task1 (b) task2

Figure 11: Probability distribution of the WCRT of tasks for the supplier (100, 33)

The results are shown in Fig. 11 and Fig. 12. In fact, Fig. 11 shows the probability distribution of the worst-case execution time of tasks for the supplier timing requirement (100, 33), where task1 and task2 have 221.005 and 331.662 as worst-case response times. For the supplier timing requirement (100, 37), as shown in Fig. 12, task1 and task2 have 206.642 and 307.204 as worst-case response times. By means of this reasoning, it can be checked that both cases for the supplier satisfy the task resource requirements and make them schedulable.

6.6. Budget Estimation

Using UPPAAL SMC, one can obtain a probability distribution of a case that satisfies a property. For the estimation of budget for a component, one can look for a budget that would satisfy the workload of a component by obtaining a probability distribution of possible budgets that disproves the schedulability.



(a) task1 (b) task2

Figure 12: Probability distribution of the WCET of tasks for the supplier (100, 37)

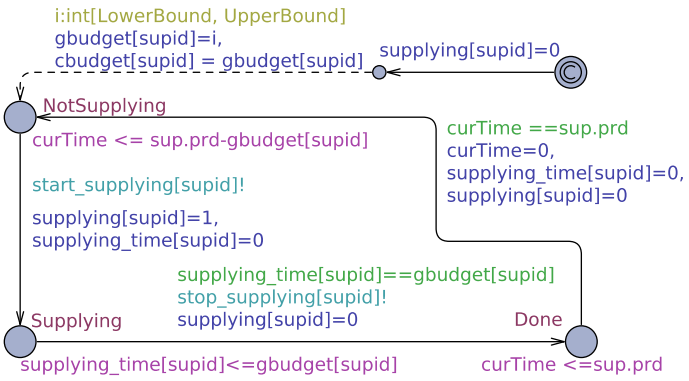


Figure 13: The supplier's template for budget estimation

To estimate the component budget, we present another stochastic supplier as shown in Fig. 13. It starts supplying by selecting a random amount of budget using gbudget[supid] and cbudget[supid]. The process of Fig. 13 is checked against the following property:

$$\Pr[\text{cbudget}[0]<=\text{runTime}](<>\ \text{gclock} >= \text{runTime and error})$$

where runTime is the limit of the simulation. This property requires UPPAAL SMC to produce a probability distribution based on the cases where error happens, i.e., a deadline is missed. In this way, UPPAAL SMC checks whether any task misses a deadline and generates a probability distribution of budgets leading to a deadline miss of a component.

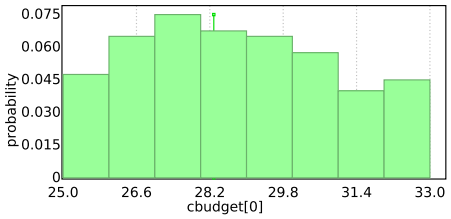


Figure 14: Probability distribution of estimated budgets leading to the deadline missing

Fig. 14 shows the estimated budget numbers that makes the component of task1 and task2 non schedulable, and it can be concluded that 33 is the minimum budget for the component because we get no counter-example beyond 33.

7. Modeling and Analysis with Power Consumption

In the area of real-time embedded systems where energy comes from a limited source, mastering and minimizing the energy consumed by system software and hardware devices is a primordial task. In this paper, we consider a single

core voltage/frequency scaling execution platform where the CPU can operate with different frequencies ($freq_d, freq_1, .., freq_n$), and thus consumes a specific voltage $volt_i$ for each frequency $freq_i$. The default frequency level $freq_d$ is the lowest frequency and corresponds to the **idle** state of the CPU, where no task is requesting the CPU resource. At such a frequency level, the CPU requires the default (smallest) voltage supply $volt_d$. The use of voltage/frequency scaling [36] has proven to be an effective way to reduce the energy consumption [42]. Since energy consumption is proportional to the CPU frequency [17], and the voltage supply is quadratic compared to the CPU clock frequency, relaxing the execution of software when it is enabled may lead to a longer execution time but consumes less energy. Thus, playing with the CPU speed and voltage can be viewed as a schedulability problem where the goal is to reduce as much as possible the energy consumption but without leading system tasks to miss their deadlines.

7.1. Energy Model

To reflect the CPU frequency scaling on the task execution, we consider different execution times for each task. Each execution time corresponds to a CPU speed (frequency), where the shortest execution time corresponds to the highest CPU speed. Accordingly, we extend the task description, introduced earlier in Section. 4, to consider the power aspect. So that a task T is given by $T = (prd, \langle e_1, .., e_n \rangle, d)$ where prd is the task period, $\langle e_1, .., e_n \rangle$ is a set of n execution times each (e_i) of which corresponds to the frequency scaling $freq_i$, and d is the task deadline. Voltage levels are assigned statically such that when a system is deployed, each task is always going to be executed at a certain voltage level and frequency.

For a given execution time e_i, the energy E_i consumed by CPU [36] to execute one time the task T is given by:

$$E_i = Cap \times volt_i^2 \times freq_i \times e_i$$

where $volt_i$ and $freq_i$ are respectively the CPU clock frequency and voltage supply corresponding to e_i. Cap is the load capacitance which represents the ability of the CPU to store an electrical charge. The energy consumed during a specific execution time of the task is given by the accumulation of the energy consumed by each execution of the task within that duration.

According to the task model depicted in Fig. 6, the energy consumed during the execution of a task identified by tid is computed at location Executing using the following expression:

```
taskenergy'== isTaskSched() * volt[freqmode(tid)] * volt[freqmode(tid)]
                                  * cpufreq[freqmode(tid)] * capacitance
```

where taskenergy is a stopwatch, isTaskSched() is a function determining whether the task is running (scheduled) or not, volt[freqmode(tid)] determines the CPU voltage supply that corresponds to the frequency mode associated to the execution of task tid, and cpufreq[freqmode(tid)] is the CPU frequency associated to

the execution of that task. capacitance is the CPU capacitance. So that if the task is not running (isTaskSched()= 0), the stopwatch derivation taskenergy'==0 which means that the amount of the energy consumed by the execution of that task does not progress right now. taskenergy keeps holding the latest value of the consumed energy. Once the task execution resumes, the stopwatch taskenergy starts progressing with a rate equal to volt[freqmode(tid)]* volt[freqmode(tid)] * cpufreq[freqmode(tid)]. One can see that our computation of the consumed energy is real-time and always depending on the length of system run time.

The power consumption [36] is defined to be the energy consumed per time unit. It is computed via the following expression:

$$power = \frac{E}{t}$$

where E is the amount of energy consumed during the time interval t. The power is often computed in `Watt`. While we consider the analysis of energy consumption as schedulability, we extend the component interface, given earlier as period and budget (prd, $budget$), by the power consumption constraint $power_{max}$ so that the interface of components has the form $(prd, budget, power_{max})$. Similarly to the resource budget $budget$, the energy constraint $power_{max}$ states the maximum power amount that can be consumed by the execution of the component workload.

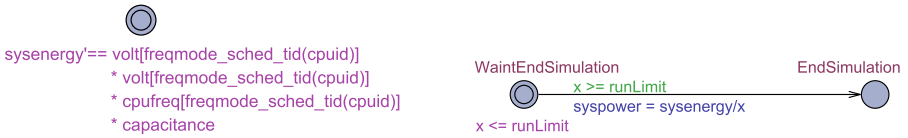
7.2. Computation and Analysis of Energy Consumption

A basic idea to compute the energy consumed by a component workload is to compute the energy consumed by each task, and in the end to sum all of the individual consumptions. In our framework, we profit from the advance of UPPAAL and use the real-time computation mechanism `stopwatch` to estimate the energy consumed by a component workload. Fig. 15(a) shows our energy-meter template to calculate the energy consumed by a component. The energy-meter template consists of one location that has been assigned the following invariant:

```
sysenergy'== volt[freqmode_sched_tid(cpuid)] * volt[freqmode_sched_tid(cpuid)]
                                  * cpufreq[freqmode_sched_tid(cpuid)] * capacitance
```

sysenergy is a stopwatch that can progress with different rates according to the CPU frequency and voltage supply. freqmode_sched_tid(cpuid) is a function determining the current task scheduled on the CPU, and returning the frequency scale assigned to that task. Based on that information, we deduce the current CPU frequency (cpufreq[freqmode_sched_tid(cpuid)]) and voltage supply (volt[freqmode_sched_tid(cpuid)]). Using all this data, we continuously compute the energy consumed by the execution of the component workload. If no task is scheduled on the CPU, the function freqmode_sched_tid(cpuid) returns the CPU default mode where frequency and voltage supply are the lowest ones.

Fig. 15(b) depicts our power-meter template. At the initial location WaitEndSimulation, the power-meter keeps waiting until the simulation time is expired



(a) Energy-meter of a component (b) Power-meter of a component

Figure 15: Energy and power calculators.

(x <= runLimit). Once the simulation is over, the power-meter computes the average power consumption by retrieving the amount of energy consumed sysenergy from the energy-meter, dividing it by the simulation time (syspower=sysenergy/x); and moves to location EndSimulation. x is a global clock measuring the simulation time.

In order to get more realistic analysis results for the energy consumption, we perform multiple experiments with different statically assigned voltage levels. Components can be individually analyzed in terms of energy consumption. The analysis of the power/energy consumed by a component workload is checked against the component interface. So the energy consumed by the workload should not exceed the maximum amount of energy that is supplied to the component. In fact, we consider energy as a consumable resource where the scheduling of tasks has to take into consideration the amount of energy left at component level.

Our energy consumption analysis can be used by engineers, during the design exploration, to determine the optimal assignment of voltage level and frequency scaling to tasks execution in order to decrease the energy consumption while keeping the system schedulable.

7.3. Power Analysis of the Running Example

In this section, we illustrate a way of computing the consumed energy and power for Component1 of the running example.

Listing 5: Declaration of energy-related factors in UPPAAL model

```
const int volt[freqmode_t]          = {  0,    3,    4,    5};   // \Supply voltage
const double cpufreq[freqmode_t]    = { 0.0, 0.5, 0.75,  1.0};   // Clock frequency
const int capacitance = 7;                                       // Load capacitance
```

Listing 5 shows the specification of UPPAAL implementation for parameters used to compute consumed energy and power. In this specification, it is assumed that a task would run with three different frequency modes. The first elements of the voltage and the frequency are dedicated to the case where no task is running, i.e., only idling task is running. One can notice the default voltage and frequency values are 0 and 0.0 respectively; in the hierarchical scheduling system, another task in different components might be running even if no task in the component under consideration is running. Thus, the consumed energy

of a component in a hierarchical scheduling system only takes into consideration the situations where a task of the component under consideration is running.

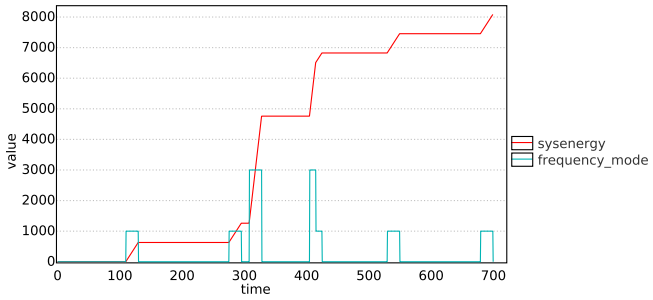


Figure 16: Energy consumption with different frequency modes

The plot in Fig. 16 illustrates the amount of energy (sysenergy) consumed by the execution of Component1's workload according to different frequency modes of the running task (frequency_mode). In this example, task1 and task2 are set to frequency modes 1 ($freq_1$) and 3 ($freq_3$), respectively, that are characterized by the specification in Listing 5. The frequency mode in Fig. 16 is either of 1,000 or 3,000, which denote $freq_1$ and $freq_3$, respectively. One can remark the increasing rate of the consumed energy in red line is different with different frequency modes. Uppaal SMC is used to analyze how much energy is consumed for a specific time (runTime) by a component with the following query:

E[<=runTime;runCnt](max:sysenergy)

where runCnt is a simulation count conducted by Uppaal SMC to compute the consumed energy of a component.

In order to compute the consumed power of a component, the following query is fed to Uppaal SMC:

E[<=runTime;runCnt](max:syspower)

Using this query, the power consumed by Component1 of the running example are computed as shown in Table 4. As shown in Table 4, the tasks have different worst-case execution times according to the frequency mode. As a result, the necessary budget required by the component where the tasks belong to are changed as well as the consumed power.

Based on budgets and power consumption obtained in Table 4, one can select the preferred system configuration that satisfies resource constraints.

8. Case Study

To show the applicability of our compositional framework, we have modeled the avionics system introduced in [27, 14], and analyzed its schedulability. Unlike the running example in the previous sections, this case study does not

Table 4: Budgets and power needed for different tasks' attributes

Conf	Tasks	FM *	Budgets	Power
1	Task1(250, 40)	1	33	8.97575
	Task2(400, 50)	1		
2	Task1(250, 40)	1	30	13.4385
	Task2(400, 40)	2		
3	Task1(250, 40)	1	29	18.1662
	Task2(400, 30)	3		
4	Task1(250, 30)	2	28	14.017
	Task2(400, 50)	1		
5	Task1(250, 30)	2	26	18.4795
	Task2(400, 40)	2		
6	Task1(250, 30)	2	26	23.2049
	Task2(400, 30)	3		
7	Task1(250, 20)	3	24	17.9375
	Task2(400, 50)	1		
8	Task1(250, 20)	3	20	22.4
	Task2(400, 40)	2		
9	Task1(250, 20)	3	20	27.125
	Task2(400, 30)	3		

*FM: CPU Frequency Mode

consider energy, but provides a larger case study of the compositional schedulability analysis. The application is a flat composition of 12 tasks declared with different priorities and timing requirements. Based on the features of tasks, we have structured this application in 4 components. Many different grouping of tasks could have been chosen. We based our division on the functionality of tasks, as this could be a natural division between development teams. The structure and timing requirements of the hierarchical system are given in Fig. 17. Under each component is a black ellipse containing the name of the component scheduling policy.

The case study is obtained by instantiating our framework with the correct number of tasks, scheduling policy and supplier model for a given component. By instantiation we only need to refer to the template name with the correct number of parameters. Of course a template can be instantiated as many times as needed, such as for the task template. At a practical level, one needs to save one copy of the UPPAAL model used per component and plug the specific timing requirements into this model. Thus, once the framework has been established, the modeling of a specific case study is a trivial task.

The budgets for each component are computed using the budget estimation technique described in Section 6.6.

Following the analysis method described in Section 4, we associate to each component a non deterministic supplier. By holding the same timing require-

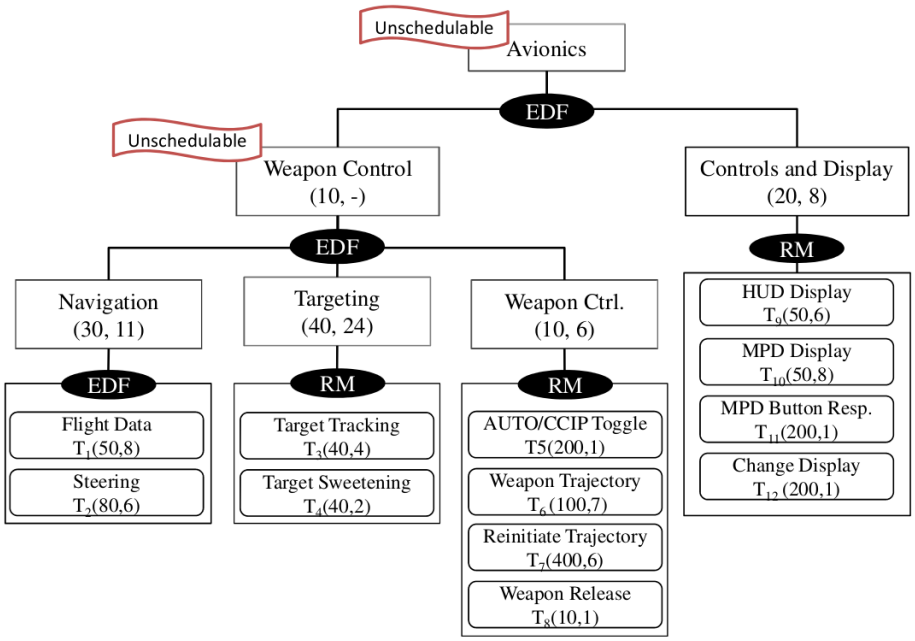


Figure 17: Avionics Component Architecture in Hierarchical Structure

ments of tasks as [27], our compositional analysis shows that all components are schedulable under different scheduling policies with or without preemption. The top level component is however not schedulable and thus the system as a whole is not schedulable. Our schedulability result of this avionics system matches perfectly with the schedulability result obtained, in a non compositional way, in [27] stating that the flat composition of the system tasks is not schedulable.

One of the strength of using a compositional approach is that as soon as a component has been found to be non schedulable, the analysis process can be immediately stopped concluding that the current system configuration is non schedulable. A remaining challenge for our approach is how to tackle power consumption compositionally in multi-level hierarchical systems.

Table 5 shows the analysis time that UPPAAL spends to check the schedulability of the system components. The analysis was performed on a regular laptop with an Intel(R) Core(TM) i7-3520M CPU @ 2.90GHz and 8GB of main memory. The analysis time is so short that it can be used by engineers as part of an interactive design process.

9. Conclusions

This paper introduced a compositional framework for modeling and analyzing the schedulability and energy efficiency of hierarchical real-time systems.

Table 5: Runtimes for the verification

Component name	**Verification time (second)**
Navigation	2.36
Targeting	0.01
Weapon Ctrl	0.026
Pilot Weapon Ctrl	6.47
Controls and Display	0.03

The component model we considered comes with constraints on both the CPU resource and the maximum power allowed to be consumed. Components are concurrently running on a single-core execution platform with voltage/frequency scaling. We tackle both detailed task behavior and power consumption of the individual tasks in the same framework. We achieve this in a modular manner, enabling thus compositional analysis, such that a system engineer can decide the best system configuration before system deployment. Our task model captures both preemptive and non preemptive behaviors while both static and dynamic priority schedulers can be used. To represent the energy awareness, we have extended the classical component interface with a maximum power constraint that is provided to the component. This constraint can then be used to find a frequency configuration for the workload which is both schedulable and satisfies the power constraint.

The framework has been instantiated as reusable templates given in terms of extensions of timed automata which we analyzed using Uppaal and Uppaal SMC. The reusable models ensure that when modeling a hierarchical scheduling application, only the hierarchical structure and the interface of each component need to be specified by the system engineer. The framework also allows for instant changes of the scheduling policy at each given level in the hierarchy. Comparing our model-based approach to analytical ones, our framework enables the modeling of more complicated and concrete systems. We have successfully applied our compositional framework to model an avionics system and analyze its schedulability. The Uppaal models we used in our framework are available at `http://people.cs.aau.dk/~ulrik/submissions/771691/models.zip`.

9.1. Future Work

A very important aspect for the future application of our model-based schedulability analysis is to validate the framework on case studies with industrial implementations. For a modeling framework to be truly useful, it should be able to predict the behavior of the actual implementation. We want to analyze the implications of replacing non determinism used in the symbolic analysis with stochastic probability distributions. The only way to prove the validity of this method is an in depth case-study where the results are validated against the actual code running on a target platform. Moreover, it would be natural to con-

sider a case study that also includes the energy aspect on a voltage/frequency scaling platform.

Using Stochastic Hybrid Automata (SHA) to characterize a timing attribute of a task, such as we have suggested for the sporadic task arrival patterns in [13], it can be problematic to find and validate that the probability distribution is sufficiently accurate. The approach to validate this would likewise be to investigate a number of case studies in order to evaluate the frameworks ability to predict the behavior of a system. The probability functions could be obtained by sampling the execution of the individual tasks.

References

[1] Y. Abdeddaïm and D. Masson. Real-time scheduling of energy harvesting embedded systems with timed automata. In *RTCSA*, pages 31–40. IEEE Computer Society, 2012.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[3] R. Alur, S. La Torre, and G. J. Pappas. Optimal paths in weighted timed automata. In Benedetto and Sangiovanni-Vincentelli [10], pages 49–62.

[4] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Times: A tool for schedulability analysis and code generation of real-time systems. In K. G. Larsen and P. Niebert, editors, *FORMATS*, volume 2791 of *LNCS*, pages 60–72. Springer, 2003.

[5] ARINC 653. Website. https://www.arinc.com/cf/store/documentlist.cfm.

[6] M. Åsberg, T. Nolte, and S. Kato. Towards partitioned hierarchical real-time scheduling on multi-core processors. *SIGBED Rev.*, 11(2):13–18, Sept. 2014.

[7] M. Åsberg, T. Nolte, and P. Pettersson. Prototyping and code synthesis of hierarchically scheduled systems using times. *Journal of Convergence (Consumer Electronics)*, 1(1):77–86, December 2010.

[8] M. Behnam, T. Nolte, I. Shin, M. Åsberg, and R. J. Bril. Towards hierarchical scheduling on top of VxWorks. In *Proceedings of the 4th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT08)*, pages 63–72, July 2008.

[9] G. Behrmann, A. Fehnker, T. Hune, K. G. Larsen, P. Pettersson, J. Romijn, and F. W. Vaandrager. Minimum-cost reachability for priced timed automata. In Benedetto and Sangiovanni-Vincentelli [10], pages 147–161.

[10] M. D. D. Benedetto and A. L. Sangiovanni-Vincentelli, editors. *Hybrid Systems: Computation and Control, 4th International Workshop, HSCC 2001, Rome, Italy, March 28-30, 2001, Proceedings*, volume 2034 of *Lecture Notes in Computer Science*. Springer, 2001.

[11] T. Bøgholm, H. Kragh-Hansen, P. Olsen, B. Thomsen, and K. G. Larsen. Model-based schedulability analysis of safety critical hard real-time java programs. In *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '08, pages 106–114, New York, NY, USA, 2008. ACM.

[12] A. Boudjadar, A. David, J. Kim, K. Larsen, M. Mikuionis, U. Nyman, and A. Skou. Hierarchical scheduling framework based on compositional analysis using Uppaal. In J. L. Fiadeiro, Z. Liu, and J. Xue, editors, *Formal Aspects of Component Software (FACS 2013)*, Lecture Notes in Computer Science, pages 61–78. Springer International Publishing, 2014.

[13] A. Boudjadar, A. David, J. H. Kim, K. G. Larsen, M. Mikucionis, U. Nyman, and A. Skou. Degree of schedulability of mixed-criticality real-time systems with probabilistic sporadic tasks. In *Theoretical Aspects of Software Engineering Conference (TASE), 2014*, pages 126–130, Sept 2014.

[14] L. Carnevali, A. Pinzuti, and E. Vicario. Compositional verification for hierarchical scheduling of real-time systems. *IEEE Transactions on Software Engineering*, 39(5):638–657, 2013.

[15] F. Cassez, A. David, E. Fleury, K. G. Larsen, and D. Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *CONCUR*, 2005.

[16] F. Cassez and K. G. Larsen. The impressive power of stopwatches. In C. Palamidessi, editor, *CONCUR*, volume 1877 of *Lecture Notes in Computer Science*, pages 138–152. Springer, 2000.

[17] A. Chandrakasan, S. Sheng, and R. Brodersen. Low-power CMOS digital design. *Solid-State Circuits, IEEE Journal of*, 27(4):473–484, Apr 1992.

[18] E. M. Clarke, D. E. Long, and K. L. Mcmillan. Compositional model checking. MIT Press, 1999.

[19] A. David, D. Du, K. G. Larsen, A. Legay, M. Mikucionis, D. B. Poulsen, and S. Sedwards. Statistical model checking for stochastic hybrid systems. In E. Bartocci and L. Bortolussi, editors, *HSB*, volume 92 of *EPTCS*, pages 122–136, 2012.

[20] A. David, P. Jensen, K. Larsen, M. Mikuionis, and J. Taankvist. Uppaal stratego. In C. Baier and C. Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 9035 of *Lecture Notes in Computer Science*, pages 206–211. Springer Berlin Heidelberg, 2015.

[21] A. David, K. G. Larsen, A. Legay, and M. Mikucionis. Schedulability of Herschel-Planck revisited using statistical model checking. In *ISoLA (2)*, volume 7610 of *LNCS*, pages 293–307. Springer, 2012.

[22] A. David, K. G. Larsen, A. Legay, M. Mikucionis, D. B. Poulsen, J. van Vliet, and Z. Wang. Statistical model checking for networks of priced timed automata. In Fahrenberg and Tripakis [29], pages 80–96.

[23] A. David, K. G. Larsen, A. Legay, M. Mikucionis, D. B. Poulsen, J. van Vliet, and Z. Wang. Statistical model checking for networks of priced timed automata. In Fahrenberg and Tripakis [29], pages 80–96.

[24] A. David, K. G. Larsen, A. Legay, M. Mikucionis, and Z. Wang. Time for statistical model checking of real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 349–355. Springer, 2011.

[25] R. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, pages 10 pp.–398, Dec 2005.

[26] Z. Deng and J. W.-S. Liu. Scheduling real-time applications in an open environment. In *RTSS*, pages 308–319. IEEE Computer Society, 1997.

[27] R. Dodd. Coloured Petri net modelling of a generic avionics missions computer. Technical report, Defence Science and Technology Organisation, Department of Defence. Australia, 2006.

[28] A. Easwaran, I. Lee, I. Shin, and O. Sokolsky. Compositional schedulability analysis of hierarchical real-time systems. In *Tenth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2007), 7-9 May 2007, Santorini Island, Greece*, pages 274–281. IEEE Computer Society, 2007.

[29] U. Fahrenberg and S. Tripakis, editors. *Formal Modeling and Analysis of Timed Systems - 9th International Conference, FORMATS 2011, Aalborg, Denmark, September 21-23, 2011. Proceedings*, volume 6919 of *Lecture Notes in Computer Science*. Springer, 2011.

[30] X. A. Feng and A. K. Mok. A model of hierarchical real-time virtual resources. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, RTSS '02, pages 26–35, Washington, DC, USA, 2002. IEEE Computer Society.

[31] E. Fersman, P. Krcál, P. Pettersson, and W. Yi. Task automata: Schedulability, decidability and undecidability. *Inf. Comput.*, 205(8):1149–1172, 2007.

[32] A. Hessel, K. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou. Testing Real-Time systems using UPPAAL. In *Formal Methods and Testing*, pages 77–117. 2008.

[33] K. G. Larsen, G. Behrmann, E. Brinksma, A. Fehnker, T. Hune, P. Pettersson, and J. Romijn. As cheap as possible: Efficient cost-optimal reachability for priced timed automata. In G. Berry, H. Comon, and A. Finkel, editors, *CAV*, volume 2102 of *Lecture Notes in Computer Science*, pages 493–505. Springer, 2001.

[34] J. Lind-Nielsen, H. R. Andersen, H. Hulgaard, G. Behrmann, K. J. Kristoffersen, and K. G. Larsen. Verification of large state/event systems using compositionality and dependency analysis. *Formal Methods in System Design*, 18(1):5–23, 2001.

[35] G. Lipari and E. Bini. A methodology for designing hierarchical scheduling systems. *J. Embedded Comput.*, 1(2):257–269, Apr. 2005.

[36] L. Niu and J. Xu. Improving schedulability and energy performance for weakly hard real-time systems. In *IPCCC, 2012 IEEE 31st*, pages 41–50, 2012.

[37] K. Purna and D. Bhatia. Temporal partitioning and scheduling data flow graphs for reconfigurable computers. *Computers, IEEE Transactions on*, 48(6):579–590, Jun 1999.

[38] I. Shin, A. Easwaran, and I. Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *Proceedings of the 2008 Euromicro Conference on Real-Time Systems*, ECRTS '08, pages 181–190. IEEE Computer Society, 2008.

[39] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *RTSS*, pages 2–13. IEEE Computer Society, 2003.

[40] I. Shin and I. Lee. Compositional real-time scheduling framework with periodic model. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.

[41] Y. Sun, G. Lipari, R. Soulat, L. Fribourg, and N. Markey. Component-based analysis of hierarchical scheduling using linear hybrid automata. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014 IEEE 20th International Conference on*, pages 1–10, Aug 2014.

[42] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced CPU energy. In *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*, pages 374–382, Oct 1995.

[43] T. ZHOU and H. XIONG. Design of energy-efficient hierarchical scheduling for integrated modular avionics systems. *Chinese Journal of Aeronautics*, 25(1):109 – 114, 2012.