Timed Automata Model for Component-Based Real-Time Systems

Georgiana Macariu and Vladimir Creţu
Computer Science and Engineering Department
Politehnica University of Timişoara
Timişoara, Romania
Email: georgiana.macariu@cs.upt.ro,vladimir.cretu@cs.upt.ro

***Abstract*—One of the key challenges in modern real-time embedded systems is safe composition of different software components. Formal verification techniques provide the means for design-time analysis of these systems. This paper introduces an approach based on timed automata for analysis of such component-based real-time embedded systems. The goal of our research is to provide a method for treating the schedulability problem of such systems on multi-core platforms. Since the components are developed, analyzed and tested independent of each other, the impact of one component on the others does not depend on its internal structure. Therefore, we reduce the problem of proving the schedulability of the composed system to proving the schedulability of each component on the resource partition allocated to it based on the interface of the component. The proposed verification method is demonstrated on a H.264 decoder case study.**

***Keywords*-real-time scheduling; model checking; components**

I. Introduction

Nowadays, real-time embedded software development is focusing more and more on how to build flexible and extensible systems. Component-based software systems achieve this objective by gluing individually designed software components, each component with its own timing requirements. The compositional design of real-time systems can be done using hierarchical scheduling and schedulability analysis of the composed system can be addressed based on component interfaces that abstract the timing requirements of each component. Furthermore, the rapid developments in multiprocessor technology determined a growing interest in multiprocessor scheduling theories and, consequently also in multiprocessor hierarchical scheduling theories [1], [2].

In this context, providing formal guarantees on the schedulability of such component-based systems running on multi-core platforms becomes even more important as these components consist of interacting tasks and each component can have a different scheduling strategy. In this paper we address the formal verification of schedulability of real-time multi-core component-based systems assuming that the components consist of preemptable periodic tasks. These tasks can be independent or there may exist various precedence constraints between them, represented as task graphs. Stopwatch automata [3] have been proposed for modeling of preemptive tasks, but reachability of composition of these automata is undecidable [4]. Moreover, it has been shown that many preemptive multiprocessor scheduling algorithms for periodic tasks suffer of scheduling anomalies when there is a change in their execution time [5]. Therefore, an accurate analysis of systems using these algorithms must consider variable task execution times. This poses another problem to our formal verification of schedulability since it was proved [6] that the schedulability problem on task graphs is undecidable if the following conditions are both met: (1) *the scheduling strategy is preemptive* and, (2) *tasks have variable execution times ranging over a continuous interval*.

We propose a formal method for checking the schedulability of real-time component-based applications running on multi-core platforms. Our method uses the timed automata [7] formalism, for which the reachability problem is decidable. Our timed automata model is actually the model of a level in a multi-core scheduling hierarchy. The proposed method uses a discrete time formalism but is also able to capture continuous task execution times by approximating the stopwatch automata model. We show this is true by proving that the formal language accepted by the timed automata model is included in the language accepted by the stopwatch automata and we evaluate the approximation errors. Also, we show how our model can be applied iteratively to check the entire scheduling hierarchy.

Related work. Formal verification of component-based systems is addressed by several frameworks for various purposes. The Save Integrated Development Environment (SAVE-IDE) [8] offers support not only for design of component based systems, but also allows specification of the behavior of each component using timed automata. Using UPPAAL [9] and timed automata models, it is possible to check if the components satisfy their specification. However, the verification features of the IDE do not allow specification of component-level scheduling strategies based only on component interfaces. Ke et. al. [10] also propose a methodology for formal verification of the timing and reactive behavior of component-based systems. Unlike the work presented in this paper, their approach assumes that tasks associated to a component execute on a single processor and each task is modeled by a separate timed automaton. In our case, the timed automata network which models a component uses a different approach in which a

single automaton is used for all tasks of the component and thus better performance can be achieved. An approach to modeling real-time systems resembling ours is taken in the TIMES tool [11] but until now the tool only offers support for analyzing uniprocessor systems.

Multi-processor schedulability analysis using model-checking has been investigated in [12]. The models in [12] allow restricted and full migration of task instances. Their work, like the one described in this paper also uses a discrete time formalism, but tasks are assumed to be independent and every task is modeled separately. Task schedulability is checked in decreasing order of task priority which implies that for a task set with N tasks, model checking has to be performed N times in order to determine the schedulability of the entire set. With this approach a maximal number of $N+1$ clocks are necessary for a task set of size N. Unlike this model checking solution, our proposal requires just a single run of the model checking for the entire task set using a single clock.

Madl et al. [13] introduce model checking for schedulability analysis of preemptive event-driven asynchronous distributed real-time systems with execution intervals. The analysis method proposed here starts from the same essential idea as the work in [13] but there are several significant differences. First, we assume a hierarchical scheduling model which means that the execution of tasks is constrained by the availability of the temporal partitions. Second, unlike the work in [13] which assumes tasks are partitioned between processors, task migration is allowed in our model. Last, instead of modeling just the tasks of an application individually, we model a whole scheduling level.

II. Preliminaries on Timed Automata

In this section we give basic descriptions and definitions related to the timed automata used in our work.

Formal syntax. Assume a finite set of real-valued clocks $\mathcal{C}$ and $\mathcal{B}(\mathcal{C})$ the set of constraints on the clocks in $\mathcal{C}$. The clock constraints (guards) are conjunctions of expressions of the form $x \bowtie N$ and $x - y \bowtie N$ where $x, y \in \mathcal{C}$, $N \in \mathbb{N}$ and $\bowtie \in \{<, \leq, =, \geq, >\}$. A timed automaton over the set of clocks $\mathcal{C}$ is a tuple $\langle L, l_0, \Sigma, \mathcal{C}, I, E \rangle$ where

- L is a set of finite locations,
- l_0 is the initial location,
- Σ is a set of actions,
- $\mathcal{C}$ is the set of clock variables,
- $I : L \to \mathcal{B}(\mathcal{C})$ associates invariants to locations,
- $E \subseteq L \times \mathcal{B}(\mathcal{C}) \times \Sigma \times 2^{\mathcal{C}} \times L$ is the set of transitions, where transition $\langle l, g, a, r, l' \rangle$ from location l to location l', labeled with action a is executed only if guard g is true and resets clocks in $r \subseteq \mathcal{C}$.

All timed automata models presented in this paper are based on the UPPAAL [9] model of timed automata which is extended with constructs such as constants, integers, committed and urgent constraints on locations, networks of timed automata and events transmitted between automata. An *urgent location* is similar to a location with all incoming transitions resetting a clock x and having associated an invariant $x \leq 0$ (i.e. time cannot pass while the automaton is in an urgent location).

Semantics. For a timed automaton we can define a clock valuation function $v : \mathcal{C} \to \mathbb{R}_+$ assigning positive real values to clocks in $\mathcal{C}$. A state s in the timed automaton is a pair (l, v) where $l \in L$ and v is a clock valuation. The automaton can stay in state s as long as the invariant associated to l is true or can execute transitions outgoing from l when the guard of these transitions is true. Therefore, two types of transitions can be defined:

- delay transitions: $(l, v) \xrightarrow{d} (l, v')$ where $v'(x) = v(x) + d, \forall x \in \mathcal{C}$ and v' preserves the invariant of location l,
- action transitions: $(l, v) \xrightarrow{a} (l', v')$ if there exists a transition $\langle l, g, a, r, l' \rangle \in E$ and guard g is true for clock valuation v and v' is obtained from v by resetting all clocks in $r \subseteq \mathcal{C}$ and leaving all others unchanged.

Networks of timed automata. A network of n timed automata $A_i = \langle L_i, l_i^0, \Sigma, \mathcal{C}, I_i, E_i \rangle$, $1 \leq i \leq n$ over a common set of clocks and actions is a parallel composition of A_i, describing a timed automaton obtained from its component automata. Semantically, the network of timed automata requires joint execution of delay transitions and synchronization over complementary action transitions.

For networks of timed automata, UPPAAL introduces the concept of committed locations. A committed location is more restrictive than an urgent location, as a state containing a committed location cannot delay and the next transition of the system must involve an outgoing edge from one of the committed locations in the state.

III. System Model

A. Real-Time Components and Component Contracts

In this paper we assume that real-time software applications consist of independent components. Further, each component consists of a set of independent *multi-threaded tasks* (MTTs), where each MTT is made up of periodic tasks with execution costs defined as continuous intervals but with a common period and deadline. The tasks in an MTT may run in parallel and it is also possible to define precedence constraints between them. The execution patterns of all tasks of a component are modeled using a timed automaton as it is described in the next sections.

Definition 1 (Component). *A component C consists of a finite set $\mathcal{MT}$ of n MTTs where:*

- *a MTT $\Theta_i \in \mathcal{MT}$, with $1 \leq i \leq n$, is a tuple $\Theta_i = (\mathcal{T}_i, p_i, d_i, r_i)$, where $\mathcal{T}_i$ is the set of t_i tasks in the MTT, p_i represents the inter-arrival time between different instances of the same MTT, d_i is the deadline by which all tasks in $\mathcal{T}_i$ should finish and r_i represents the time of the first release of Θ_i,*

- *each task $\tau_j \in \mathcal{T}_i$, $1 \leq j \leq t_i$, is characterized by a tuple $(bcet_j, wcet_j, prio_j)$ where $bcet_j$ and $wcet_j$ are integer values that specify the limits of the continuous execution interval of task τ_j and $prio_j$ is the priority of the task.*

All numeric parameters in Definition 1 are considered integer numbers.

The tasks belonging to each component are scheduled separately using a component-specific preemptive scheduling policy. Therefore, when building an application based on such components one must ensure that the tasks of each component are schedulable independent of the execution of any other component in the application.

One of the solutions for ensuring temporal isolation of components running on uni-processor or multi-processor real-time systems is provided by hierarchical scheduling schemes based on execution time servers [14]. In hierarchical scheduling each application has its own scheduler and can use the scheduling policy that best suits its needs. Based on such a hierarchical scheduling scheme, Harbour has introduced the concept of service contracts [15]. In Harbour's model, every application or application component may have a set of service contracts describing its minimum resource requirement. Similarly, we define service contracts to capture the resource requirements of our components. Next, each service contract is mapped to a set of execution time servers which mark the limits of the resources allocated by the application to the child component. An execution time server in a multi-core system, as considered in this paper, is characterized by a tuple (Q, P) meaning that the component will receive Q units of execution every P units of time. Additionally, we consider a third parameter o representing the time when the server is first released. It is assumed there is a finite set of servers $\mathcal{S}$ containing the servers for all components of an application.

In terms of the timed automata formalism we define a component contract as follows:

Definition 2 (Component contract)**.** *A component contract $\mathcal{C}_C$ providing a set of n_s execution servers $\mathcal{S}_C \subseteq \mathcal{S}$ is a timed automaton $\mathcal{A}_{Cc}$ over the set of actions $\Sigma_{\mathcal{C}_c}$ such that:*

- *$\mathcal{A}_{Cc}$ specifies the activation pattern of servers $\sigma_i = (Q_i, P_i, o_i) \in \mathcal{S}_C$, $1 \leq i \leq n_s$,*
- *$\Sigma_{\mathcal{C}_c} = \{active, inactive\}$, where action $active$ signals to the component scheduler that a server $\sigma_i \in \mathcal{S}_C$ has just become active (i.e. the processor is now available to be used by the component), while $inactive$ signals deactivation of the server.*

We consider that parameters Q_i, P_i and o_i have integer values.

The component also has associated a scheduler which will schedule for execution the tasks of the MTTs in a component according to a preemptive scheduling policy. We consider that a global scheduling policy is used and, as such, a task can run on any processing unit. As a consequence, task migration may occur whenever a task is preempted or suspended. The scheduler of the component is modeled by a timed automaton with the following characteristics:

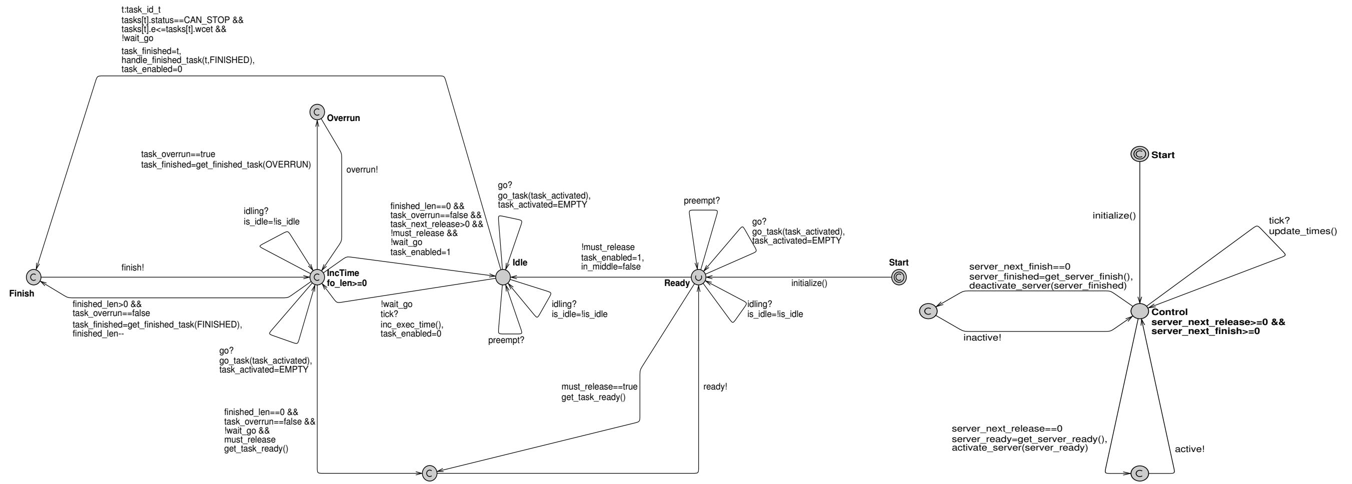
- has a queue holding the tasks ready for execution,
- implements a preemptive scheduling policy representing a sorting function for the task queue,
- maintains a map between active execution servers and tasks using the servers, and
- has an `Error` location which is reached when a task misses its deadline.

A component consisting of n MTTs could have been modeled also using a timed automaton for each of the tasks of the MTTs, each automaton with its own clock. Since the state space of timed automata models grows exponentially with the number of clocks in the model, we decided to build a single timed automaton which models the execution patterns of all n MTTs and reduce the number of clocks to one as it will be shown in the following subsection. Moreover, each task could have been modeled using stopwatch automata but the reachability analysis of composed stopwatch automata is undecidable [4]. The same observation applies for modeling the component as a single stopwatch automaton and consequently, we propose an approximation of a stopwatch model using timed automata with discrete clocks to keep track of the execution time of each task.

B. Timed Automata Model for Real-Time Components

In the timed automata formalism a component of a real-time application is the network of timed automata obtained through parallel composition of the automaton which models the execution pattern of the MTTs of the component, the component scheduler automaton and the timed automaton modeling the activation and deactivation patterns of the execution time servers (i.e. the $\mathcal{A}_{Cc}$ automaton in Definition 2). In what follows we will use the names *Task Generator (TGT)* to denote the timed automaton which models the MTTs' execution and *Server Generator (SG)* for the one modeling the servers. Apart from these, the network also includes a *Timer* automaton which uses a single continuous clock t and each time this clock ticks the automaton sends a *tick* signal to the *TGT* and the *SG* automata.

Before explaining in more detail the timed automata model we introduce some notations. For each MTT Θ_i we use a variable $R(i)$ to hold the time of the next release of Θ_i. In order to determine the actual execution time of each task τ_j, we use a variable $E(j)$ to keep track of the time task τ_j has executed since its last release. Basically, $E(j)$ acts like a discrete clock which can be suspended and resumed. Also, for each task τ_j a variable $status(j)$ indicates its current status and is initialized to *idle* meaning that a task instance has not been released yet. The value $status(j) = ready$ is used to denote that a task instance of



(a) The *Task Generator* timed automaton. (b) The *Server Generator* timed automaton.
Figure 1. Task and Server generators.

τ_j is ready for execution (i.e. it has just been released or was preempted). If τ_j is waiting for one of its predecessor tasks to finish then $status(j) = waiting$. If an instance of task τ_j is running then $status(j) = running$. A task τ_j which has executed for $bcet_j$ time units but for less than $wcet_j$ units will have $status(j) = can_stop$. To denote that an instance of task τ_j has finished or has missed its deadline we use $status(j) = finished$ and $status(j) = overrun$, respectively. The discrete clock $E(j)$ keeps track of the overall time for which $status(j) = running$.

The *TGT* automaton presented in Figure 1(a) uses a variable $task_next_release$ to remember the time until one of its MTTs must be released and a variable $task_next_finish$ to keep the earliest time when one of the currently running tasks should finish. At start-up, $task_next_release$ is initialized to $min(R(i))$, $i = 1, 2, .., n$ and the automaton goes to the *Ready* location. From this location, if $task_next_release = 0$ and there is at least one task τ_j with $status(j) = idle$ (i.e. *TGT* must release a task), the automaton executes the transition with the guard $must_release = true$ and in function $get_ready_task()$ elects the task τ_j, $j = 1, 2, .., t_i$, for which $status(j) = idle$, updates $task_next_release$ (only if this is the first task τ_j in the MTT that is released in the current period) and $task_next_finish$, sets a shared variable $task_ready$ to $(i - 1) \cdot n + j$ (a global identifier of task τ_j of the MTT Θ_j) and then sends the $ready$ signal to the scheduler automaton which will read the $task_ready$ variable and will add task τ_j to its queue. The process is repeated until $task_next_release$ becomes greater than 0 and there are no idle tasks for any MTT that has just been released. At this point the automaton goes to the *Idle* location where it waits for a *tick* signal from the *Timer* automaton. On each release of the MTT Θ_i, $R(i)$ is postponed with p_i.

The *SG* automaton presented in Figure 1(b) works in a similar way with the distinction that generated active servers are continuous (not preemptable). For each server σ_k, $1 \leq k \leq n_s$, there is a discrete clock $RE(k)$ analogous to $E(j)$ and a variable $RR(k)$ is used to keep the time until the next activation of σ_k. Two additional variables, $server_next_release$ and $server_next_finish$ hold the time until the earliest start time of a server σ_k and the earliest finish time, respectively. When $server_next_release = 0$ a processing unit becomes available for the component (i.e. some σ_k starts) and the *SG* takes the transition guarded with $server_next_release = 0$. In function $get_sever_ready()$ *SG* determines the server σ_k which became active, updates $server_next_release$ and $server_next_finish$ and sets a shared variable $server_ready$ to k. Afterwards, the $active$ signal is sent to the scheduler of the component to announce the activation of server σ_k. Also, for the server that just started, $RR(k)$ is set to P_k. When σ_k finishes and the processing unit is no longer available, the automaton takes the transition guarded with $server_next_finish = 0$ and, similar to the previous scenario, sets the shared variable $server_finished = k$ and sends the $inactive$ signal to the scheduler.

On every tick of the timer, *TGT* leaves the *Idle* location and goes to the *IncTime* location. During this transition, in function $inc_exec_time()$, the current execution time $E(j)$ of all tasks τ_j running (with status set to $running$ or can_stop) at that time are increased with a value MIN representing the minimum between $task_next_release$, $task_next_finish$, $server_next_release$ and $server_next_finish$. If, as a result of this update, there are tasks for which $E(j)$ reached $bcet_j$ then we set $status(j) = can_stop$ and if $E(j) = wcet_j$ then the task has finished its execution, $status(j)$ becomes $finished$ and a variable $finished_len$ counting the finished tasks is incremented. At the same time, we identify any task τ_j that missed its deadline and set $status(j) = overrun$. Also, as time passes the time $R(i)$

of the next release of each MTT Θ_i is decreased with MIN and the values $task_next_release$ and $task_next_finish$ are updated. When the *SG* receives the *tick* signal from the *Timer*, in function $update_times()$, it increases the current activation length $RE(k)$ of all active servers σ_k with MIN and decreases $RR(k)$ of all servers with the same value. Also the values of the variables $server_next_release$ and $server_next_finish$ are updated.

If some $R(i)$ reaches 0 then a new instance of the MTT is released (*TGT* sends the *ready* signal to the scheduler as explained before). When the scheduler (see Figure 2) receives notification of a new task being released, it checks if a server on which to schedule the task is available and, if so, sends the go signal to the *TGT* automaton and sets the entry in its server-task map accordingly. If the priority of the newly released task is higher than the priority of one of the running tasks and no active servers are idle, the scheduler will preempt the lower priority task and will give the server to the higher priority task. If no server is available or the server is deactivated while a task is running on it, the task is either scheduled on another server (if its priority allows it) or is queued. On every tick the *TGT* automaton searches for all tasks τ_j that finished their execution or that missed their deadline and sends $finish$ or $overrun$ signals to the scheduler. If the server used by a finished task is still active and there are ready tasks waiting in the scheduler's queue, a new task is started and the go signal is sent to *TGT*. Moreover, when an active server σ_k finishes, the scheduler will attempt to reschedule the task that was using the server associated with it on some other free server available to the component. If no active server is free, then the lowest priority running task may be preempted. Between all automata, data (e.g. task identifier or resource identifier) is transmitted using shared variables. A more detailed description of the scheduler timed automaton can also be found in [16].

In order to be able to capture task execution intervals in continuous time, when the *TGT* automaton is in the *Idle* location, if there is at least one task τ_j with $status(j) = can_stop$, the automaton may decide non-deterministically to finish the task. A remark that must be made is that whenever there is at least one task with status can_stop the value of MIN is set to 1. This implies that at the next *tick* signal, the discrete clocks presented above are increased with a single time unit. If a task τ_j finishes at some fraction of the time unit, another task τ_l that was previously preempted or is ready to be released may take the place of τ_j. However, because in this case we cannot keep track in the discrete clock $E(l)$ of the time task τ_l is executing until the first tick after it has been started/restarted, the value in $E(l)$ is only an approximation of the real execution time of τ_l. Although, this approach represents just an approximation model of the real system, we will show in the next section that the model preserves the properties of the system and any component that is deemed schedulable with our model is indeed schedulable.

It is important to notice that once each component of an application is proved to be schedulable, by using reachability analysis on our model we can also check the schedulability of the entire application as follows. Each execution time server σ_k is basically a periodic task with hard deadlines and fixed execution requirement Q_k. Therefore it can be considered as an MTT consisting of a single task with $bcet = wcet$ and the whole application can be seen as just another component with its own scheduler and whose MTTs are the execution servers corresponding to the service contracts of its components. If we consider that the application also has a service contract mapped to another set of execution server we can again check the proposed model by changing only the parameters of the MTTs and of the execution servers to reflect the new scheduling level represented by the parent application.

IV. Analysis of the Timed Automata Model Approximation

A. Stopwatch Automata as a Model for Real-Time Components

Stopwatch automata [3] can be defined as timed automata for which clocks can be stopped and later resumed with the same value. These clocks are called stopwatches and provide a simple way for modeling preemptive real-time tasks. Syntactically, a stopwatch automaton SWA is a tuple $\langle L, l_0, \Sigma, \mathcal{C}, I, E, A\rangle$ where $L, l_0, \Sigma, \mathcal{C}, I, E$ have the same meaning as for timed automata (see Section II) and $A : L \times \mathcal{C} \rightarrow \{0, 1\}$ is a function that defines the rates of clocks $c_i \in \mathcal{C}$ in locations as differential functions $\dot{v}(c_i) = k_i$ where $k_i \in \{0, 1\}$.

From a semantical point of view, the element that distinguishes the SWA from the timed automaton is the clock valuation function $v : \mathcal{C} \rightarrow \mathbb{R}_+$ assigning positive real values to clocks in $\mathcal{C}$. In a SWA the value of a clock variable during a delay transition $(l, v) \xrightarrow{d} (l, v')$ is updated to $v'(c_i) = v(c_i) + A(l, c_i) \cdot d, \forall c_i \in \mathcal{C}$.

In our case, since the tasks belonging to the MTTs of a component are scheduled using a preemptive scheduling policy we could have chosen to model the component using the stopwatch automaton in Figure 3. The execution time of each task in each MTT is represented as a stopwatch clock ec_j, $\forall j \in \{1, 2, ..., n \cdot t_i\}$ with $1 \leq i \leq n$. With each MTT we associate a clock dc_i, $\forall i \in \{1, 2, .., n\}$ which will keep track of the MTT's deadline.

In the stopwatch version of our components we only need to replace the *TGT* automaton with a *Task Generator* stopwatch automaton (TGS). The *TGS* presented in Figure 3 still uses the variable $task_next_release$ to remember the time until one of its MTTs must be released but it is not necessary to keep the variable $task_next_finish$. When the system starts, $task_next_release$ is initialized to

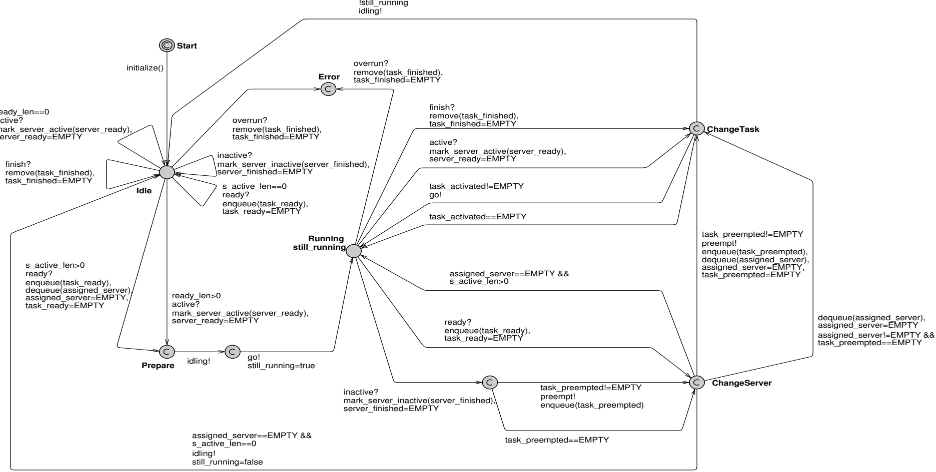


Figure 2. Timed automaton model for the scheduler of the component.

$min(R(i))$, $i = 1, 2, .., n$ and the automaton goes to the *Ready* location. From this location, it can either go the *Idle* location if there are no tasks ready for release or, if there is at least one task τ_j with $status(j) = idle$, the automaton executes the transition with the guard $must_release = true$ and, in function $get_ready_task()$ elects the task τ_j, $j = 1, 2, .., t_i$, which is idle, updates $task_next_release$, puts the task global identifier $(i-1) \cdot n + j$ in the shared variable $task_ready$ and sends the $ready$ signal to the scheduler automaton. The scheduler will read the $task_ready$ variable and will add task τ_j to its queue. On each release of the MTT Θ_i, $R(i)$ is postponed with p_i. After all tasks that are ready to start are released the stopwatch automaton goes to the *Idle* location where it waits for a *tick* signal from the *Timer* automaton or for a running task τ_j to finish. Another event which may take the automaton out of the *Idle* location is a missed deadline of any MTT. The rates of the clocks ec_j and dc_i are specified in the guard of the *Idle* location: $dc'_i = 1$ for all MTTs which contain at least one task that is not finished yet, otherwise $dc'_i = 0$ and $ec'_j = 1$ for all tasks that have $status(j) = running$ but $ec'_j = 0$ for the other tasks.

In the stopwatch automaton the execution time of each task τ_j is measured by stopwatch ec_j started at the release of the task, when the automaton sends the *ready* signal to the scheduler automaton while the variable $task_ready = j$, until the task finishes and the *finish* signal is sent with variable $task_finished = j$. ec_j does not include the time while the task was preempted. Therefore, for any task τ_j belonging to the multi-threaded task Θ_i the following constraints should be true such that we can say that τ_j has not missed its deadline:

$$0 \leq ec_j \leq wcet_j,\ 0 \leq ec_j \leq dc_i,\ 0 \leq bcet_j \leq wcet_j \quad (1)$$

Definition 3. *A multi-threaded task $\Theta_i = (\mathcal{T}_i, p_i, d_i, r_i)$ is schedulable iff all its tasks $\tau_j = (bcet_j, wcet_j, prio_j) \in \mathcal{T}_i$ finish execution before the deadline of Θ_i: $dc_i \leq d_i$ when $ec_j = wcet_j$, $\forall j \in \{1, 2, ..., t_i\}$.*

Definition 4. *A component is schedulable iff all its multi-threaded tasks are schedulable.*

The set of actions of the *TGS* is $\Sigma = \{ready, go, preempt, finish, overrun, idling, tick\}$. The *idling* signal is sent by the scheduler automaton when it goes in or out of the *Idle* location. *TGS* stays in the *Idle* location as long as either there is no server active or there are no ready tasks to be scheduled or both of these conditions are true. The *go* and *preempt* signals are controlled also by the scheduler. The stopwatch automaton will send *ready* for every new release of a task instance and *finish* at its end.

A timed word over the alphabet Σ is a pair (ρ, θ) where $\rho = \rho_1, \rho_2, ..$ is an infinite sequence of events in Σ and $\theta = \theta_1, \theta_2, ..$ is a timed sequence denoting the timestamps of the events in ρ. A timed language over Σ is a set of timed words over Σ. The timed language $L(S)$ accepted by the stopwatch automaton is the union of the timed languages $L_j(S)$ where the words in each language $L_j(S)$ refer to valid event sequences generated during the execution of task τ_j. We consider that $L(S) = \bigcup_{1 \leq i \leq n} \bigcup_{1 \leq j \leq t_i} L_j(S)$ because the semantics of task related events in Σ are established only in correspondence with a shared variable indicating the task to which the event refers. The untimed words in all $L_j(S)$, and consequently in $L(S)$, are described by the following regular expression:

$$E_S = (ready, go, (preempt, go)^*, finish) \quad (2)$$

In our case the timestamps of all events $\{ready, go, preempt, finish\}$ acceptable by the stopwatch automaton have to be less than the deadline of the MTT containing the task for which the event appeared (i.e. the task is indicated in a shared variable). This implies

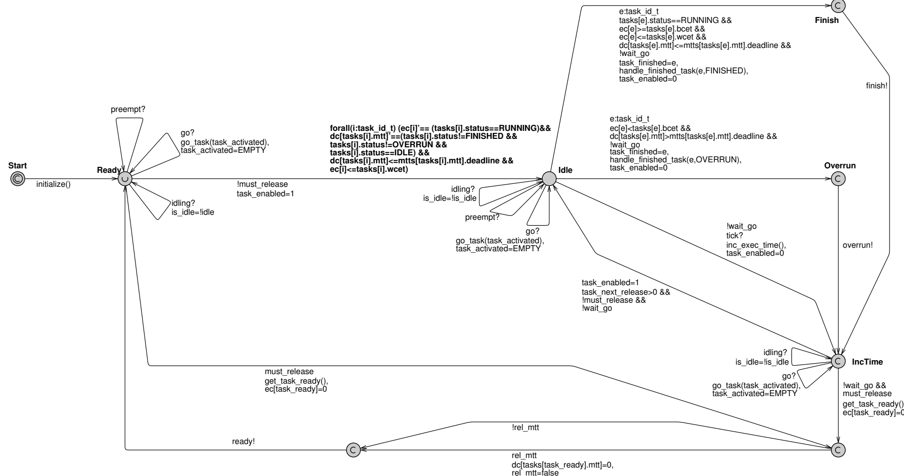


Figure 3. Stopwatch automata model of a real-time component.

that the time a task τ_j in Θ_i spends in the $ready$ state, denoted from now as T_{ready_j}, must be lower or equal than $d_i - wcet_j - T_{wait_j}$, where T_{wait_j} is the time the task has to wait for its predecessors to finish.

Problem formulation. The *TGS* is not an initialized stopwatch automaton and consequently reachability is undecidable for it [17]. Moreover, it has been demonstrated in [6] that the schedulability analysis problem for multiprocessor systems is undecidable if (1) the tasks are scheduled using a preemptive scheduling strategy, and (2) the tasks have execution times ranging over a continuous time interval. Next, we show that the timed automata model for component-based applications proposed by us and described in Section III provides a decidable method for verification of real-time component-based applications which approximates the stopwatch automata model with discrete clocks but at the same time is able to capture continuous execution time intervals.

B. Approximation of Components using Timed Automata

In this section we show what are the approximation errors implied by the proposed method for schedulability analysis of real-time component-based applications. We also prove that any component declared schedulable by our method it would also be declared schedulable by the stopwatch model. We do this by showing that the language $L(T)$ of *TGT* is also accepted by *TGS*.

The alphabet of the *TGT* is similar to the one of the stopwatch automaton. The *tick* and *idling* events in Σ are not directly related to the execution of the task and only help in modeling the discrete time. These two signals are kept in *TGS* since the activation time of the servers is still measured with discrete clocks (i.e. we see them as non-preemptable tasks with integer parameters). Keeping the discrete clocks also in *TGS*, instead of replacing them with real clocks, has no influence on the accuracy of the model and on the schedulability analysis. Therefore, in what follows we will refer mostly to the task-related events in Σ: $ready, go, preempt, finish$ ($overrun$ means the system is not schedulable and, as we analyze the conditions under which the system is schedulable, we assume this event does not appear).

For the proposed discrete time model we have chosen to consider a time unit equal to 1. The approximation errors in our model arise from the following situations:

(1) a task τ_j starts its execution at some subdivision of the time unit and
(2) a task τ_j resumes its execution (after it has been preempted) at some subdivision of the time unit.

Since we cannot measure the time from the start/restart point of τ_j until the beginning of the next time unit, the subunit of execution time will not be reflected by $E(j)$.

A task τ_j can be preempted either when a higher priority task τ_l becomes ready for execution or the server used by task τ_j is deactivated and there is no other active and free server. Activation and deactivation of servers is observed by *TGT* only through task preemptions and resumptions. As for all MTTs the release time is an integer value, a higher priority task τ_l can be released for execution while τ_j is executing only at discrete moments. If τ_l does not depend on any task it means that, in this case, τ_l can preempt τ_j only at discrete moments of time and the clocks $E(j)$ and $E(l)$ will behave just like the clocks ec_j and ec_l. If τ_l depends on some task τ_k and τ_k finishes at some time between two consecutive discrete moments then this makes it possible for τ_l to preempt τ_j. In the later case, the clocks $E(j)$ and $E(l)$ will not be increased at next discrete time point. If task τ_j resumes is execution during the same fraction of time in which it was preempted, clock $E(j)$ will still not be increased as in this case τ_j's role is similar to that of τ_l in the previous case. This implies that no matter how many

times the task is preempted between two successive discrete time points, in the computation of $E(j)$ this will have the same effect as a single preemption. Since all tasks of an MTT Θ_i have the same deadline d_i and are released at the same time then for all tasks in Θ_i at most m_j preemptions may influence the value of the discrete clock $E(*)$ without a deadline miss occurring, where

$$1 \leq m_j \leq d_i - T_{wait_j}, m_j \in \mathbb{N} \quad (3)$$

The alphabet of *TGT* is the alphabet of the *TGS*, namely $\Sigma = \{ready, go, preempt, finish, overrun, idling, tick\}$. Just like for the stopwatch automaton, in this case also we are interested only in the events in Σ related to task execution. Therefore, for each task τ_j, the timed automata has to accept timed words following the syntax of the untimed regular expression:

$$E_T = (ready, go, (preempt, go)^*, finish) \quad (4)$$

We see that $E_T = E_S$.

In what follows we use v_{sw_j} to denote the valuation of the stopwatch clock ec_j in *TGS* and v_t for the valuation of the continuous clock in *TGT*, where $v_t \in [0, 1]$. We also consider a valuation $v_j = E(j) + v_t$ for each task τ_j. This helps in measuring the approximation error of the proposed model. Note that v_j is an approximation of v_{sw_j}. As at most m_j preemptions of a task τ_j can influence the value of $E(j)$ it results that $v_{sw_j} - v_j \leq m_j$. Also, we use v_{dc_i} to denote the valuation of the clock dc_i in *TGS* and we consider the valuation $v_{\Theta_i} = p_i - R(i) + v_t$ which measures the time since the release of the MTT Θ_i. The valuation v_{Θ_i} will grow with the same slope as the valuation v_{dc_i} and consequently $v_{\Theta_i} = v_{dc_i}$ at any time between the release of Θ_i and its finish.

In order to establish the relationship between *TGT* and *TGS*, we must compare the timed words that follow the syntax of E_T. We assume the timestamps of these words analyzed in relation to the valuations v_{Θ_i} for *TGT* and v_{dc_i} for *TGS*, respectively, are the same.

Theorem 1. *For any timed word that follows the syntax of E_T and simulates the execution trace of a task τ_j on both* TGT *and* TGS *automata, $v_{sw_j} - m_j \leq v_j \leq v_{sw_j}$ holds from the release of the task until its ending, $\forall j \in \{1, 2, ..., t_i\}$ and $\forall i \in \{1, 2, ..., n\}$.*

Proof: Both *TGS* and *TGT* receive events related to a specific task in the same order and with the same timestamps (related to v_{Θ_i} and v_{dc_i}, respectively). Whenever the *TGS* receives a *go* signal it sets the status of the task to *running* and when it receives the *preempt* signal the status of the task is set to *ready*. The behavior of the *TGS* upon receiving the *go* and *preempt* events is the same.

When the status of the task is *running*, v_{sw_j} grows with slope 1 and stays constant when the task has status *ready*. The valuation v_j also stays constant while the task is in the *ready* state and grows with slope 1 when the task is running due to the v_t component. If the task is preempted only at discrete time points (e.g. when the server that the task was using finished its available execution units) then at the end of the task $v_j = v_{sw_j}$. However, if a task preemption happens between two successive distinct time points, the valuation v_t is not added to v_j, whereas v_{sw_j} will contain also this fraction of time unit, and therefore $v_j \leq v_{sw_j}$. Since the v_j can decrease for at most m_j times, each time with at most 1 time unit, it follows that $v_{sw_j} - m_j \leq v_j$. ■

The inequality in Theorem 1 shows that during the simulation of the same word on both the *TGT* and *TGS* automata, it will take at least the same amount of time for v_j to reach a specific value as it will take to v_{sw_j}.

Next we analyze possible timestamps of the *finish* event. As this event is related to the guards in the automata that contain the best case execution time $bcet_j$ of a task τ_j and its worst case execution time $wcet_j$ we will determine what is the relation between the actual best execution time t_{bcet_j} and the actual worst execution time t_{wcet_j} of τ_j and the valuations $v_j = bcet_j$ and $v_j = wcet_j$. Note that v_j does not include those fractions of time units that we cannot measure in the *TGT*.

Theorem 2. *For any timed word that follows the syntax of E_T and simulates the execution trace of a task τ_j on* TGT, *if $v_j = bcet_j$ then $t_{bcet_j} \leq bcet_j$, $\forall j \in \{1, 2, ..., t_i\}$ and $\forall i \in \{1, 2, ..., n\}$.*

Proof: If task τ_j is not preempted during its execution or is preempted only at discrete time points then $t_{bcet_j} = bcet_j$. If the task is preempted between two consecutive discrete time points then the valuation v_j when the task will resume its execution will not contain the subunit of time that it had executed before it was preempted and only an integer number of time units. In contrast, t_{bcet_j} will contain those time fractions and, therefore it will reach $bcet_j$ faster than v_j which means that $t_{bcet_j} < bcet_j$. ■

Theorem 3. *For any timed word that follows the syntax of E_T and simulates the execution trace of a task τ_j on* TGT, *if $v_j = wcet_j$ then $wcet_j \leq t_{wcet_j}$, $\forall j \in \{1, 2, ..., t_i\}$ and $\forall i \in \{1, 2, ..., n\}$.*

Proof: If task τ_j is not preempted during its execution or is preempted only at discrete time points then $t_{wcet_j} = wcet_j$. If the task is preempted between two consecutive discrete time points then the valuation v_j when the task will resume its execution will not contain the subunit of time that it had executed between before it was preempted and only an integer number of time units. In contrast, t_{wcet} will contain those time fractions and, therefore by the time v_j will reach $wcet_j$ t_{wcet_j} will be greater than $wcet_j$. ■

From theorems 2 and 3 it follows that, if a task finishes its execution before its deadline in *TGT* it will always meet

its deadline in *TGS* which means that if the task is proven schedulable in *TGT* then it will also be schedulable in *TGS*.

For a task τ_j belonging to a MTT Θ_i to be schedulable it is also required for it to finish before its deadline. If T_{ready_j} is the time for which the task has the *ready* status then we say that the task is schedulable if the following condition is satisfied:

$$T_{ready_j} + m_j \leq d_i - wcet_j - T_{wait_j} \quad (5)$$

The condition 5 says that task τ_j can be preempted for at most $d_i - wcet - T_{wait_j} - m_j$ before its deadline. The m_j term appears due to the imprecision of the model.

Next we prove that *TGS* accepts the timed language over Σ that *TGT* accepts. Specifically we prove that $L(T) \subseteq L(S)$ by checking the intersection $L(T) \bigcap \overline{L(S)} = \emptyset$. We have already shown that the syntax of the timed language $L(S) = \bigcup_{1\leq i\leq n} \bigcup_{1\leq j\leq t_i} L_j(S)$, where $L_j(S)$ is the language with words referring to valid event sequences generated during the execution of task τ_j. By analogy, $L(T) = \bigcup_{1\leq i\leq n} \bigcup_{1\leq j\leq t_i} L_j(T)$. Therefore, if $L_j(T) \subseteq L_j(S)$, $\forall 1 \leq j \leq t_i$ and $\forall 1 \leq i \leq n$ then also $L(T) \subseteq L(S)$. With this objective we prove that $L_j(T) \bigcap \overline{L_j(S)} = \emptyset$, $\forall 1 \leq j \leq t_i$ and $\forall 1 \leq i \leq n$. For a task τ_j to be schedulable all words in $L_j(T)$ must satisfy condition 5. Similarly, all words in $L_j(S)$ must satisfy the condition $T_{ready_j} \leq d_i - wcet_j - T_{sw_wait_j}$, where $T_{sw_wait_j}$ is the task waiting time in *TGS* and $T_{sw_wait_j} \leq T_{wait_j}$. Then the condition $L_j(T) \bigcap \overline{L_j(S)} = \emptyset$ becomes ($T_{ready_j} + m_j \leq d_i - wcet_j - T_{wait_j}$ and $T_{ready_j} > d_i - wcet_j - T_{sw_wait_j}$). As from condition 3 we know that $m_j \geq 1$ and T_{ready_j} cannot be at the same time higher than a value α and smaller than a value β with $\alpha > \beta$, it follows $L_j(T) \bigcap \overline{L_j(S)} = \emptyset$ is true for any task τ_j and, therefore $L(T) \bigcap \overline{L(S)} = \emptyset$ holds.

V. The H.264 Decoder: a Case Study

Using the proposed method for schedulability analysis, in this section we present a series of experiments in which we apply the proposed schedulability method to analyze the multimedia H.264 decoder [18]. We model the decoder as a component consisting of several MTTs. The tasks in each MTT and the precedence constraints between them are established from the workflow of the decoder as follows. During decoding each video frame is divided into slices and each slice is further split in blocks called macro-blocks (MB). The decoding process can be applied to several slices in parallel and consists of a well-defined set of steps. We define a MTT for each slice in a frame and each slice processing step is mapped to a task of the MTT. In the first stage of the process, numerical values are recovered from the binary codes of the compressed video (Entropy Decoding). Since a part of the data in the encoded video was computed through prediction, in the next stage (Dequantization and Inverse Transform DQIT) the differences between the predicted data and the real data are recovered. Next, in the motion compensation (or Inter-prediction) or Intra-prediction stage, each MB in the frame is decoded based on predicted data from previous frames or other MBs in the frame. Finally, the MBs of each slice are put together (Reconstruction) and a filtering stage is applied to improve quality of the decoded slice. Correspondingly, each MTT will have five tasks, one for each of the above stages. The MTT corresponding to the decoding process is presented in Figure 4. We consider that for each frame the same MTT will process the same slice.

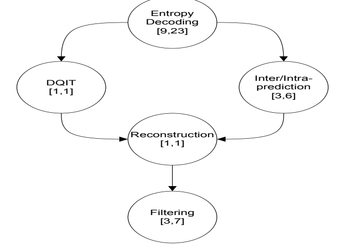


Figure 4. The MTT for slice decoding.

To see how we can apply our method on the H.264 decoder, we extracted the execution parameters of the tasks in a MTT through profiling of the FFMPEG[1] encoder using three real video files[2] with increasing level of spatial details and amount of movement. After profiling we have obtained the $[bcet, wcet]$ intervals shown in Figure 4.

We have checked the schedulability of the component model using the UPPAAL model checker by issuing the verification of the `A[] not Error` property. We ran experiments considering that for each frame of the video 2 (experiment 2-S), 3 (3-S), 4 (4-S) or 5 (5-S) slices are processed in parallel, where for each slice we have a MTT. The number of actual tasks in each experiment is five times the number of slices used for the experiment. First, for each task in a MTT, we have considered the time intervals in Figure 4 and for the period and deadline of the MTT we have chosen the value 100. Since we noticed that the difference between the $wcet$ and the $bcet$ of each task and between execution times of different tasks influences the scalability of our method (the maximum number of supported MTTs), we then ran experiments in which all these time values were doubled (experiments 2-SD, 3-SD, 4-SD, 5-SD). In the last series of experiments (2-SD1, 3-SD1, 4-SD1, 5-SD1) we doubled only the execution time of the first task in the MTTs. We have chosen only this task since it has the greatest execution time, which varies along the largest interval. For all experiments we have used a set of three execution servers, one on each processor, with periods equal to 50 and a total processor utilization of 2.0 meaning that every 50 time units the servers provide 100 execution units to the decoder. The experiments were executed on a machine with Intel Core 2 Quad 2.40 GHz processor and 4 GB RAM running Ubuntu. The model checking time of all experiments is presented in Figure 5.

We can see in Figure 5 that although the number of tasks gets up to 25 the model checking time is rather

[1] http://ffmpeg.org/

[2] We have used the QCIF Akiyo, Foreman and Mobile videos from http://trace.eas.asu.edu/yuv

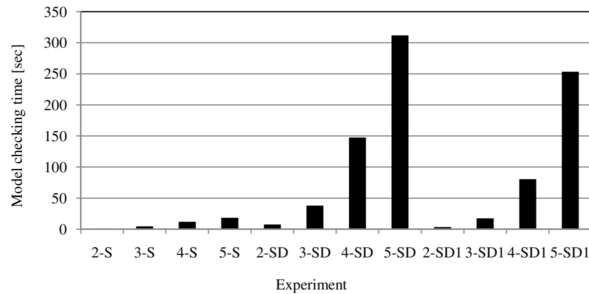


Figure 5. Model checking time

small (maximum 310 seconds). Also it can be seen that the complexity is a factor of the number of tasks, but is influenced also by the difference between the best and worst case execution time. Also, from the last four experiments (2-SD1 - 5-SD1) it can be observed that by increasing the difference between parameters of different tasks, the model checking time also grows even more than when we double all task parameters. We believe this is due to the fact that the change in these experiments has increased the non-determinism of the model.

VI. Conclusions

In this paper, we have proposed an approximation method for analysis of schedulability of component-based applications using model checking. Using timed automata we have provided a decidable method for schedulability verification in multi-core systems. The method captures dependencies between preemptable tasks and is able to capture continuous task execution times using a discrete time formalism. We have shown that the approximations in our method give a sufficient condition to determine the schedulability of a component and, by using iteratively the analysis on the parent application we can also prove the schedulability of the application. The applicability of our method was shown on a H.264 decoder.

Acknowledgments

This research is supported by eMuCo, a European project supported by the European Union under the Seventh Framework Programme (FP7) for research and technological development.

References

[1] A. Easwaran, I. Shin, and I. Lee, "Optimal virtual cluster-based multiprocessor scheduling," *Real-Time Systems*, vol. 43, no. 1, 2009.

[2] B. B. Brandenburg and J. H. Anderson, "Integrating hard/soft real-time tasks and best-effort jobs on multiprocessors," in *ECRTS '07: Proceedings of the 19th Euromicro Conference on Real-Time Systems*, 2007.

[3] F. Cassez and K. G. Larsen, "The impressive power of stopwatches," in *CONCUR '00: Proceedings of the 11th International Conference on Concurrency Theory*, 2000.

[4] Y. Kesten, A. Pnueli, J. Sifakis, and S. Yovine, "Decidable integration graphs," *Information and Computation*, vol. 150, no. 2, 1999.

[5] R. Graham, "Bounds on the performance of scheduling algorithm," *SIAM Journal of Applied Mathematics*, vol. 17, no. 2, pp. 416–429, 1969.

[6] P. Krcal, M. Stigge, and W. Yi, "Multi-processor schedulability analysis of preemptive real-time tasks with variable execution times," in *FORMATS '07: Proceedings of the 5th International Conference on Formal Modeling and Analysis of Timed Systems*, vol. 4763, October 2007, pp. 274–289.

[7] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, 1994.

[8] S. Sentilles, A. Pettersson, D. Nystrom, T. Nolte, P. Pettersson, and I. Crnkovic, "Save-IDE - A tool for design, analysis and implementation of component-based embedded systems," in *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, 2009.

[9] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a nutshell," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 1, pp. 134–152, 1997.

[10] X. Ke, K. Sierszecki, and C. Angelov, "COMDES-II: A component-based framework for generative development of distributed real-time control systems," in *RTCSA '07: Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2007.

[11] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi, "TIMES - a tool for modelling and implementation of embedded systems," in *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2002.

[12] N. Guan, Z. Gu, Q. Deng, S. Gao, and G. Yu, "Exact schedulability analysis for static-priority global multiprocessor scheduling using model-checking," in *SEUS'07: Software Technologies for Embedded and Ubiquitous Systems*, vol. 4761, September 2007, pp. 263–272.

[13] G. Madl, N. Dutt, and S. Abdelwahed, "A conservative approximation method for the verification of preemptive scheduling using timed automata," in *RTAS '09: Proceedings of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2009.

[14] G. Lipari and E. Bini, "A methodology for designing hierarchical scheduling systems," *Journal of Embedded Computing*, vol. 1, no. 2, 2005.

[15] M. G. Harbour, "Architecture and contract model for processors and networks," Universidad de Cantabria, Technical Report D-AC1, 2006.

[16] G. Macariu and V. Cretu, "Model-based analysis of contract-based real-time scheduling," in *SEUS'09: Software Technologies for Embedded and Ubiquitous Systems*, vol. 5860, November 2009, pp. 227–239.

[17] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya, "What's decidable about hybrid automata?" in *STOC '95: Proceedings of the 27th annual ACM Symposium on Theory of Computing*, 1995.

[18] "ISO/IEC 14496-10. coding of audio-visual objects. part 10: Advanced Video Coding."