Model-Based Analysis of Contract-Based Real-Time Scheduling

Georgiana Macariu and Vladimir Creţu

Computer Science and Engineering Department
Politehnica University of Timisoara
Timisoara, Romania

Abstract. We apply automata theory to analyze the schedulability of real-time component-based applications running on uniform multi-processor platforms. The resource requirements of each application or application component are specified in a service contract resulting a hierarchy of contracts. As we are interested in determining the schedulability of such applications, this hierarchy of contracts is mapped to a hierarchical scheduling strategy. We use model checking and transform the schedulability analysis problem into a reachability checking of a timed automata model of the service contracts.

1 Introduction

In the last years, real-time embedded software development has focused more and more on building flexible and extensible applications. Component-based software systems achieve these objectives by gluing individually designed, developed and tested software components, each component having different timing requirements. Therefore, when building such a component-based system one must ensure that components can coexist without jeopardizing each other's execution.

One of the solutions for temporal isolation of applications running on uniprocessor systems has been provided by utilizing hierarchical scheduling based on execution time servers [1]. 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 [2]. In Harbour's model, every application or application component may have a set of service contracts describing its minimum resource requirement. These contracts are used in online or offline negotiations to determine if the resource requirements can be guaranteed or not.

Recently, hierarchical scheduling has been used in a multi-processor scheduling framework for integrating applications with hard, soft and non-real-time requirements [3]. Also research is undertaken for extending the service contract model for component-based multi-processor real-time systems. Chang et. al [4] has proposed a two-level resource contract model. First, each application has a contract specifying the resources to be reserved for its execution. This is called an *external contract*. Next, every component of the application has its own contract, called *internal contract*, describing the portion of the resources specified in

S. Lee and P. Narasimhan (Eds.): SEUS 2009, LNCS 5860, pp. 227–239, 2009.
© IFIP International Federation for Information Processing 2009

the external contract that must be distributed to the component. Each component consists of one or more tasks which may require parallel execution. Internal contracts are mapped to abstract servers which are further divided in execution time sub-servers (called just servers in what follows) in order to support parallel execution of the components. On the other hand, external contracts are mapped to multi-processor time partitions [5]. As each application will be mapped to a separate time partition, a specific scheduling policy may be associated with it.

Starting from the hierarchical scheduling solution proposed in [4], we apply timed automata theory [6] to specifying the service contracts for components and applications. A component contract describes the tasks of the component and the arrival patterns of these tasks, modeled by a timed automaton. An application contract will refer to the servers for all components in the application and to the arrival patterns of these servers, modeled also as a timed automaton. We allow a different scheduling policy for each component and application, with no restriction on task preemption. Furthermore, we present a compositional approach based on timed automata for schedulability analysis of component-based real-time applications running on uniform multi-processor platforms. For each application, schedulability (i.e. checking that all application components can be executed such that all their tasks meet their deadlines) can be analyzed separately. In the timed automata formalism, schedulability analysis is reduced to reachability and can be performed using a tool like UPPAAL [7].

Schedulability analysis of real-time systems using timed automata has been proved decidable and applied successfully for non-preemptive scheduling policies of tasks. However, in timed automata models as defined in [6] time elapses at the same rate for all components and therefore they cannot be used for preemptive scheduling policies where execution of tasks can be suspended and resumed later. Stopwatch automata [8], a subclass of Linear Hybrid Automata, have been proposed as a solution for modeling preemptible tasks. However, since the reachability of Linear Hybrid Automata has been proved undecidable [9], this proof extends also to the stopwatch automata. An over-approximation method based on Difference Bound Matrixes has been applied in [8] for a coarse reachability analysis of stopwatch automata. Even so, the schedulability checking problem has been shown to be decidable for non-uniformly recurring tasks triggered by events. [10] introduces timed automata extended with tasks, a class of timed automata with subtraction where clocks may be updated by subtraction in a bounded zone, and proves that the schedulability checking relative to a preemptive scheduling policy is decidable for this class of automata. This result has been extended for multi-processor real-time systems in [11] where it is shown that the schedulability problem is decidable for preemptive scheduling policies with fixed execution time tasks. However, they do not allow task migration meaning that a task instance is bound with one processor until it finishes. In our framework a task instance may execute on any processor depending on the availability of the execution time servers and the configuration of the multi-processor time partition.

The global multi-processor schedulability analysis using model-checking has been investigated for tasks with static priorities in [12]. The models in [12] allow

restricted and full migration of task instances. Every task is modeled separately and the schedulability of tasks is checked in decreasing order of their priority which limits the applicability of the analysis to static scheduling policies. This also 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 addresses both static and dynamic scheduling policies. Moreover it requires just a single run of the model checking for the entire task set using a single clock in a setting with resources that are not continuously available and multiple levels of scheduling.

This paper is organized as follows. Section 2 introduces our formal model for contract-based scheduling and Section 3 gives details on the timed automata used in the system model. We present performance evaluation results in Section 4. Section 5 concludes this paper.

2 The Contract-Based Scheduling Model

This section presents the formal model of the service contracts. As explained in Section 1 there are two levels of such contracts. The first level specifies the resource requirements of a single application while the second level describes the requirements of each individual component of the application. Corresponding to the two levels of contracts there are two scheduling levels. At the upper level, each component of an application has a scheduler for scheduling its tasks, while at the lower level there is an application scheduler which manages the servers associated with each component of the application.

2.1 Component Contracts

A component C consists of a finite set of n tasks $\mathcal{T}$ and a timed automaton $\mathcal{A}_C$ where:

- a component task $\tau_i \in \mathcal{T}$ is a tuple $\tau_i = (w_i, p_i, o_i, d_i)$, with w_i being the worst case execution time of the task, p_i the inter-arrival time between different instances of the same task, o_i the first release of the task and d_i is the deadline of the task where $w_i \leq d_i \leq p_i$,
- tasks may execute in parallel and are independent of each other,
- $\mathcal{A}_C$ models the execution of tasks in set $\mathcal{T}$ by taking transitions labeled with actions $tReady_i$, $tFinish_i$ and $tOverrun_i$, $\forall\, 1 \leq i \leq n$ representing the release and ending of task τ_i, and actions tGo_i and $tPreempt_i$ through which the component scheduler notifies task execution start/restart and suspension.

The tasks of the component will be executed according to a component specific scheduling policy implemented by a scheduler associated with the component. The parameters of the tasks along with the task arrival pattern determine the resource requirements for the component. These resource requirements can be supported using one or more execution time servers, depending whether the tasks

must execute in parallel or not. The period, deadline and budget of the servers associated with a component are specified in the component contract.

A server is defined by a tuple (q, p, o) where q is the capacity of the server, p is its replenishment period (i.e. the server becomes active every p time units) and o is the time of its first release. Each server may also have a deadline equal to its period. It is assumed there is a finite set of servers $\mathcal{S}$ containing the servers for all the components of an application.

Definition 1 (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 automata $\mathcal{A}_{Cc}$ over the set of actions $\Sigma_\mathcal{C}$ such that:*

- *$\mathcal{A}_{Cc}$ specifies the activation pattern of servers $\sigma_i \in \mathcal{S}_C$, $1 \leq i \leq n_s$.*
- *$\Sigma_\mathcal{C}$ is split in two sets:*
  - *output actions: $\Sigma_\mathcal{C}^O = \{sReady_i, sFinish_i, sOverrun_i, sActive_i, sInactive_i \mid 1 \leq i \leq n_s\}$*
  - *input actions: $\Sigma_\mathcal{C}^I = \{sGo_i, sPreempt_i \mid 1 \leq i \leq n_s\}$.*

The $\mathcal{A}_{Cc}$ automaton sends the output action $sReady_i$ to the scheduler associated with the application as soon as server σ_i is ready for execution and sends $sFinish_i$ or $sOverrun_i$ to the same scheduler to notify it that the server has finished its execution, respectively missed its deadline. As a response to its actions $\mathcal{A}_{Cc}$ can receive from the application scheduler sGo_i, telling it that server σ_i can start its execution, or $sPreempt_i$ which results in server σ_i being suspended from execution until the next sGo_i action. Actions $sActive_i$ and $sInactive_i$ are used to announce the component scheduler that server σ_i has consumed all its budget, respectively that it has replenished its budget and can be used again to execute tasks.

2.2 Application Contracts

As proposed in [4] the application contracts are supported by a multi-processor time partition model. Each application is associated with a time partition which has a local scheduler to execute the execution time servers assigned to the components of the application.

In a uni-processor system a time partition is implemented as a fixed-length major time frame composed of several scheduling windows. A scheduling window is defined by its offset to the beginning of the partition major time frame and by its length. The scheduling scheme of the major time frame repeats during the execution of the system such that all scheduling windows are essentially periodic. In a multi-processor system, we assume there is a major time frame for each processor, but frames on all processors will have equal length and will be synchronized. The scheduling windows of frames on different processors can be different.

From the above specification we derive next a formal definition of the multi-processor time partition.

Definition 2 (Time partition). *A time partition $\mathcal{TP}$ in multi-processor system is described by a set of major time frames $\{\mathcal{F}_i \,|\, 1 \leq i \leq m, length(\mathcal{F}_i) = L\}$, one for each of the m processors in the system, where $\mathcal{F}_i$ is a set of scheduling windows with periods that are an exact divisor of L.*

In our setting the time partition is used to facilitate application contracts. In a simple scenario, the application contract could specify a few pairs of period and length values which upon successful negotiation of the contract could be mapped to a set of scheduling windows.

Definition 3 (Application contract). *An application contract $\mathcal{C}_\mathcal{A}$ is a pair $(\mathcal{TP}, \mathcal{A}_{Ca})$ where:*

- *$\mathcal{TP}$ is the multi-processor time partition provided by the contract, and*
- *$\mathcal{A}_{Ca}$ is a timed automaton over the action set Σ_{SW} modeling the scheduling scheme of the major time frame:*
  - *$\Sigma_{SW} = \{swActive_k, swInactive_k\}$, where k is a scheduling window in $\mathcal{TP}$.*
  - *action $swActive_k$ signals to the application scheduler that the scheduling window k is now active, while $swInactive_k$ signals its deactivation.*

3 The Timed Automata Models

As shown in the previous section both component and application levels include three automata - one for generating tasks or servers according to a given release pattern, one for generating the resources (servers or scheduling windows) on which the tasks and servers, respectively shall be executing and one for scheduling. Notice that servers can be both schedulable entities (i.e. when referring to the application scheduler) and resources (i.e. for the component scheduler). For this reason in the rest of this section they are referred simply as tasks and, respectively as resources. Also, this plurality of roles implies that the timed automaton generating tasks for the application level is the same with the one generating resources for the component level. Therefore, this automata can be deduced immediately from the task generator and the resource generator automaton types. The rest of the section is dedicated to given detailed descriptions of each of the three types of automata. In addition to the three types of automata, the model also includes a *Timer* automaton which uses a single continuous clock t and each time this clock ticks sends a *tick* signal to the task generator and the resource generator automata.

We first introduce some notations. Let $W(i)$, $P(i)$, $D(i)$, $R(i)$ and $E(i)$ denote the worst case execution time, the period, the deadline, the next release time and the current execution time, respectively for each task τ_i. For each task τ_i it is defined a status variable $status(i)$ that is initialized to *idle* meaning that a task instance has not been released yet. The value $status(i) = ready$ is used to denote that a task instance of τ_i is ready for execution (i.e. it has just been released or was preempted). Let $status(i) = running$ stand for the fact that

a task instance of τ_i is currently running on one of the active resources. To denote that an instance of task τ_i has finished or has missed its deadline we use $status(i) = finish$ and $status(i) = overrun$, respectively.

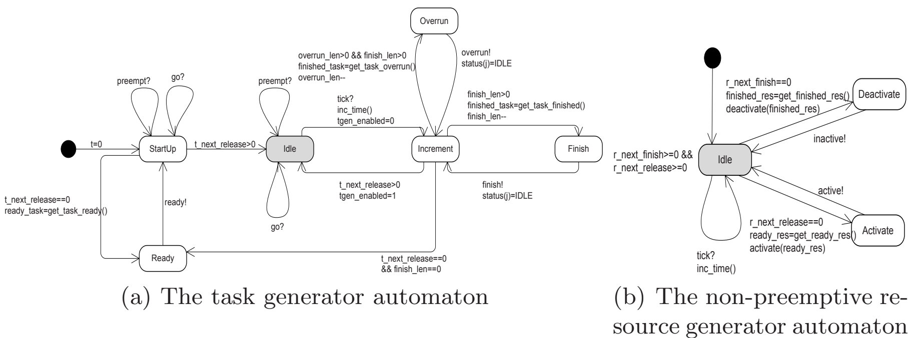
3.1 Task Generator Automaton

Model checking of preemptive scheduling algorithms could be done using a stopwatch model but it has been proved that schedulability of these models is undecidable. Therefore, in order to address task preemption a discrete time formalism is adopted for the model proposed in this paper. This leads to a limitation as all task parameters (i.e. worst case execution time, period, deadline, release time) must have integer values.

In order to be able to determine the actual execution time of a task, a variable $E(i)$ is used for keeping track of the time task τ_i has executed since its last release. Each time the task is released $E(i)$ is set to 0 while $R(i)$ is set to the time of its next release. When the task generator automaton receives a *tick* signal from the *Timer* automaton it increases $E(i)$ for tasks with $status(i) = running$ and decreases $R(i)$ for all tasks with a value MIN representing the minimum between the time for the next release of a task or of a resource and the time for the next termination of a task or deactivation of a resource. In other words, $E(i)$ acts like a discrete clock which can be suspended and resumed.

Instead of using a task generator for releasing all n tasks of a component according to some pattern, it would have been possible to define a timed automaton for each of the n tasks, each automaton with a clock, leading to a total of n clocks. Since the state space of timed automata grows exponentially with the number of clocks in the model, the approach taken in this paper is superior to this one.

Figure 1(a) shows the main locations and transitions in the task generator automaton, leaving out some self-loop transitions. All white locations in the figure have the semantics that the system cannot delay in those locations and the next transition must involve an outgoing edge from one of them.



(a) The task generator automaton

(b) The non-preemptive resource generator automaton

Fig. 1. Task and resource generators

The task generator automaton uses a variable *next_release* to remember the time until the next task is released. At start-up this variable is initialized with the smallest $R(i)$ and if after that $next_release = 0$ the automaton goes to the *Ready* location and selects a task τ_i for which $R(i) = 0$, updates *next_release*, sets the shared variable $ready_task = i$ and sends the *ready* signal to the scheduler automaton. Once *next_release* becomes greater than 0, the generator moves to the *Idle* location where it waits for the next tick of the *Timer*. When the *tick* signal arrives the transition to the *Increment* location is taken and *inc_time*() updates the values $status(i)$, $E(i)$ and $R(i)$ as follows:

- for all tasks τ_i with $status(i) = running$ $E(i) = E(i) + MIN$ and if $E(i) = W(i)$ then $status(i) = finished$,
- for all tasks τ_i $R(i) = R(i) - MIN$ and $next_release = min(R(i))$,
- for all tasks τ_i running or ready for execution with $E(i) < W(i)$ and $P(i) - D(i) = R(i)$ sets $status(i) = overrun$d.

Next, for all tasks τ_j that have finished the variable $finished_task$ is set to j and the $finished$ signal is sent to the scheduler which will free the resources used by these tasks. If any task τ_j has missed its deadline an *overrun* signal notifies the scheduler which as a result will go to an *Error* location. After signaling all task finish events the generator checks to see if there is any task ready for execution and goes back to the *Ready* location.

3.2 Resource Generator Automaton

The task generator automaton presented above can be used to generate servers which act as resources for the component level. By adding just two signals - *active* and *inactive* - to notify the scheduler about the availability of the resources the task generator automaton becomes a resource generator automaton with the property that those resource are preemptible. If resources are not preemptible (i.e. the scheduling windows of a time partition) the resource generator automaton is a simplified version of the task generator.

Figure 1(b) presents the non-preemptive version of the resource generator automaton. The automaton keeps a discrete clock $RE(k)$ for each resource r_k. Also $RR(k)$ is used to remember the time until the next activation of resource r_k and two variables named *next_release* and $next_finish$ hold the time until the next resource activation and, respectively deactivation. When resource r_k is activated $RE(k) = L(k)$ where by $L(k)$ we denote the length of the resource's activation period. At every *tick* signal received from the *Timer* for all active resources r_k $RE(k)$ is decreased with the value MIN and variables *next_release* and $next_finish$ are also decreased with the same value. When *next_release* reaches 0 all resources r_k with $RR(k) = 0$ are activated. If $next_finish$ becomes 0 than all resources r_k with $RE(k) = 0$ are deactivated.

3.3 Scheduler Automaton

As it can be seen from the definitions in the previous sections, the component scheduler and the application scheduler have rather similar behavior. Both of

them must schedule a set of periodic tasks/servers with deadlines less or equal to their period. The component tasks are scheduled on execution time servers which may be active or inactive. It is possible for two or more servers to be active simultaneously which implies that two or more tasks may run in parallel. For the application scheduler the tasks to be scheduled are actually the servers used by the component scheduler as resources. The servers are scheduled for execution on the scheduling windows of a time partition. The scheduling windows represent the resources allocated to the application by the system. As more scheduling windows can be active simultaneously parallel execution of the servers is also possible.

A scheduler automaton for a service (i.e. application or component) contract has the following characteristics:

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

To record the status of a resource, let $rt_map(j)$ be a map where $rt_map(j) = inactive$ denotes that resource j is inactive, $rt_map(j) = active$ means that resource j is active but no task is executing on it, and $rt_map(j) = i$ denotes that resource j is active and is currently used by task τ_i.

Figure 2 shows the scheduler automaton. The locations of the automaton have the following interpretations:

1. *Idle* - denotes the situation when no task is ready for execution or no resources are active,
2. *Prepare* - a task has been released and a resource is active after a period during which either there were no tasks to schedule or no active resources,
3. *Running* - at least one task is currently executing,

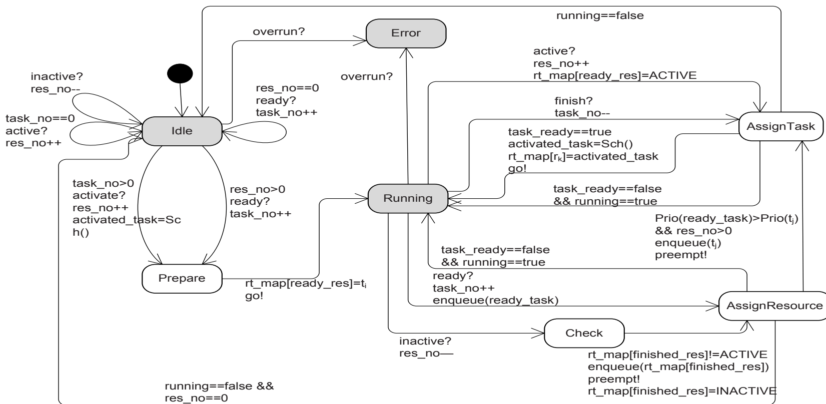


Fig. 2. The scheduler automaton

4. *AssignTask* - a task has just finished and as a result an active resource can be used to schedule another ready task,
5. *AssignResource* - a task has just been released or a resource has just become inactive leaving its assigned task with no resource on which to execute; consequently the task has to be enqueued and if it has the highest priority in the queue according to *Sch* then an active resource is assigned to it,
6. *Check* - a resource has become inactive,
7. *Error* - the task set is not schedulable with *Sch*.

The scheduler enters the *Idle* location when either there are no ready tasks, no active resources or both of these conditions hold. As long as new tasks are released for execution but there are no active resources on which the tasks to be executed (i.e. $task_no > 0$ *and* $res_no == 0$) or as long as there are available resources but no ready tasks (i.e. $task_no == 0$ *and* $res_no > 0$) the scheduler stays in the *Idle* location. If the scheduler receives a *ready* signal meaning that task τ_{ready_task} has been released and $res_no > 0$ the scheduler goes to the *Prepare* location. Leaving the *Prepare* location for the *Running* location, it assigns the task to one of the active resources by setting $rt_map(j) = i$, sets the variable $activated_task = i$ and sends a *go* signal to announce the task generator automaton that task τ_i is running. After the scheduler has reached the *Running* location, it will leave this location if one of the following situations happen:

- the resource r_k becomes active (signaled by the *active* signal and *activated_re-source* $= k$): this is marked by updating $rt_map[k] = ACTIVE$ on the transition to the *AssignTask* location. If tasks are ready for execution than the scheduler will assign the highest priority task τ_j to resource r_k by setting $rt_map[k] = j$ and will notify the task generator with the signal *go* on a transition back to the *Running* location.
- a new task τ_i has been released (signaled by the *ready* signal and $ready_task = i$): the task is enqueued by setting $status(i)$ to *ready* on the transition to the *AssignResource* location. If task τ_i is the highest priority released task and there are active resources then τ_i must start executing. If there is a free active resource then task τ_i is assigned to it otherwise the lowest priority task is chosen from the running tasks, preempted and the automaton goes to the *AssignTask* location. On the transition from *AssignTask* to *Running* the resource is assigned to τ_i and a *go* signal is sent to the task generator to notify it that task τ_i has started running.
- the resource r_k becomes inactive (signaled by the *inactive* signal and $deactivated_resource = k$): this is marked by updating $rt_map[k] = INACTIVE$ on the transition to the *Check* location. If the deactivated resource was free and there are still running tasks but no tasks in the queue then the transition back to *Running* location is taken. If a task τ_i was using resource r_k then the scheduler must set $status(i) = ready$ and go to *AssignResource* location. Should the resource r_k be the last active resource the scheduler would simply preempt task τ_i and go back to the *Idle* location, otherwise an active resource is searched analog to the situation when a new task is released.

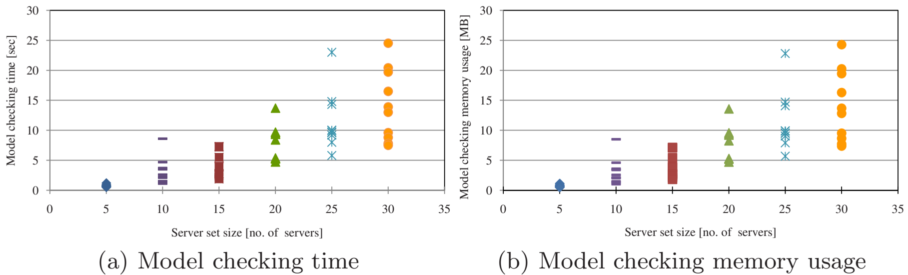
- the task τ_j finishes (signaled by the $finish$ signal and $finished_task = j$): the resource used until now by τ_j can be assigned to the highest priority task waiting in the queue, if there is such a task.
- the task τ_i misses it deadline (signaled by $overrun$): the scheduler automaton goes into the *Error* location.

4 Performance Analysis

This section presents an evaluation of the performance and scalability of model checking the contract-based scheduling model. The experiments were run on a machine with Intel Core 2 Quad 2.40 GHz processor and 4 GB RAM running Ubuntu. The analysis of the model was automated using UPPAAL and the utility program memtime [13] was used for measuring the model checking time and memory usage. Although the proposed model addresses scheduling at two levels, namely task level and server level, experiments were conducted only for the server level as we consider the analysis of the task level is just a replica of the server level due to the similarities between the two levels. In all experiments, to verify schedulability we checked if property `A[] not Error` holds.

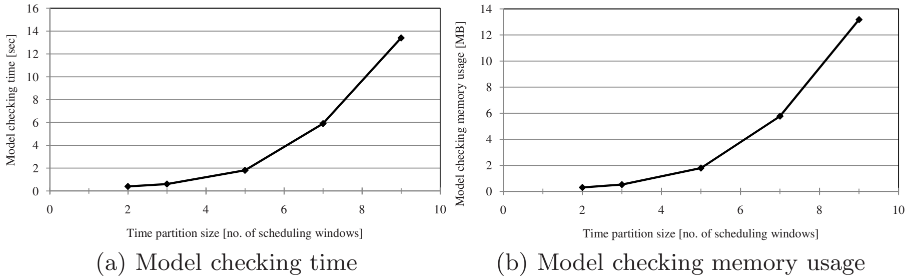
In order to observe the behavior of the model for different number of application servers we have used randomly generated sets of servers with periods in the range $[10, 100]$ and utilizations (i.e. $budget/period$) generated with a uniform distribution in the range $[0.05, 1]$. The offset of each server was set to a value equal to the period multiplied with a randomly generated number in the interval $[0, 0.3]$. Also, the servers sets were accommodated by a time partition with 9 scheduling windows and a total utilization of 4.5. Figure 3 shows how the model checking time and memory usage increase with the number of servers in the set. Also it can be noticed that for the same size of the server set the performance of the model checking can vary between rather larger limits (e.g. for sets of 30 servers the model checking time grows from 7 seconds to approximatively 25 seconds). This is due to the size of the hyper-period of the server sets, larger the hyper-period larger the model checking time and memory consumption.

Next, we analyzed the scalability and performance of model checking when the number of scheduling windows in the time partition accommodating the servers varies. For this, sets with 25 servers each and parameters in the same



(a) Model checking time (b) Model checking memory usage

Fig. 3. Influence of server set size on model checking performance



(a) Model checking time (b) Model checking memory usage

Fig. 4. Influence of time partition size on model checking performance

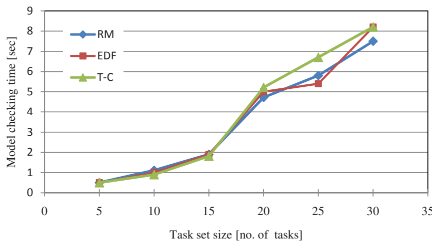


Fig. 5. Influence of scheduling policy on model checking time

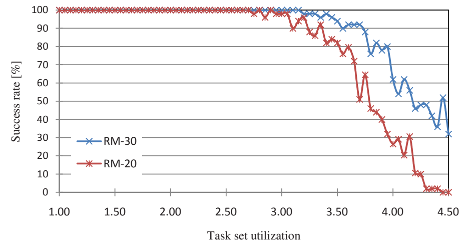


Fig. 6. Schedulability of task sets

limits as for the first experiment were generated and time partitions with 2, 3, 5, 7 and 9 scheduling windows were tested. In Figure 4 it can be seen that both the time for checking the model and the memory usage grow with the number of scheduling windows in the time partition.

In the first two experiments the server sets were scheduled using the Rate Monotonic (RM) priority scheduling policy. The goal of our next experiment is to determine the impact of the scheduling policy on the model checking time and peak memory usage. The same time partition configuration as in the first experiment was used and sets of 5, 10, 15, 20, 25 and 30 servers were scheduled using both the Rate Monotonic, the Earliest Deadline First (EDF) and the (T-C) (i.e. the higher the difference between the period and the budget of a server the lower its priority) scheduling policies. As can be seen in Figure 5 the scheduling policy has little influence on the performance of the model checking.

In the last experiment we are interested in seeing what is the influence of the task set utilization on the schedulability analysis. We have used the same time partition as in the first experiment with a total utilization of 4.5 and task sets of 20 and 30 tasks with utilizations between 1 and 4.5 scheduled using the Rate Monotonic policy. Figure 6 depicts the number of schedulable task sets identified by our analysis. It can be noticed that even if the total utilization of a task set is maximal with respect to the available resources, our analysis is able to determine its schedulability, which is a clear advantage over the pessimist schedulability bounds presented in [4].

5 Conclusions

In this paper we have presented a compositional approach using the timed automata formalism for schedulability analysis of component-based real-time applications which utilize multi-processor resource partitions. Starting with the assumption that the resource requirements for each application and component are stipulated in a service contract we have defined a timed automata model for specifying the contracts and shown how to use model checking as a technique for analyzing the preemptive schedulability of an hierarchy of such contracts. The performance analysis of our technique using the UPPAAL model checker showed that even with just one real-time clock used for the entire model, the applicability of the technique is limited by the state-explosion problem.

Acknowledgment

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. Lipari, G., Bini, E.: A methodology for designing hierarchical scheduling systems. Journal of Embedded Computing 1(2), 257–269 (2005)
2. Harbour, M.G.: Architecture and contract model for processors and networks. Technical Report D-AC1, Universidad de Cantabria (2006)
3. Brandenburg, B.B., Anderson, J.H.: Integrating hard/soft real-time tasks and best-effort jobs on multiprocessors. In: ECRTS 2007: Proceedings of the 19th Euromicro Conference on Real-Time Systems, Washington, DC, USA, pp. 61–70. IEEE Computer Society, Los Alamitos (2007)
4. Chang, Y., Davis, R., Wellings, A.: Schedulability analysis for a real-time multiprocessor system based on service contracts and resource partitioning. Technical Report YCS-2008-432, Computer Science Department, University of York (2008)
5. Kaiser, R.: Combining partitioning and virtualization for safety-critical systems. White Paper, SYSGO AG (2007)
6. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
7. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. International Journal on Software Tools for Technology Transfer 2(1), 134–152 (1997)
8. Cassez, F., Larsen, K.G.: The impressive power of stopwatches. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 138–152. Springer, Heidelberg (2000)
9. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's decidable about hybrid automata? In: STOC 1995: Proceedings of the 27th annual ACM symposium on Theory of computing, pp. 373–382. ACM, New York (1995)
10. Fersman, E., Krcal, P., Pettersson, P., Yi, W.: Task automata: Schedulability, decidability and undecidability. Information and Computation 205(8), 1149–1172 (2007)

11. Krcal, P., Stigge, M., Yi, W.: Multi-processor schedulability analysis of preemptive real-time tasks with variable execution times. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 274–289. Springer, Heidelberg (2007)
12. Guan, N., Gu, Z., Deng, Q., Gao, S., Yu, G.: Exact schedulability analysis for static-priority global multiprocessor scheduling using model-checking. In: Obermaisser, R., Nah, Y., Puschner, P., Rammig, F.J. (eds.) SEUS 2007. LNCS, vol. 4761, pp. 263–272. Springer, Heidelberg (2007)
13. Memtime utility, `http://freshmeat.net/projects/memtime/`