From Nonpreemptive to Preemptive Scheduling

From Single-Processor to Multi-Processor*

Mohammad Mahdi Jaghoori
LIACS, Leiden University, The Netherlands
CWI, Amsterdam, The Netherlands
jaghouri@cwi.nl

ABSTRACT

The use of automata for specifying patterns of task generation has broaden the perspective of schedulability analysis; scheduling has moved from periodic or rate-monotonic to aperiodic and non-uniform tasks. The question of schedulability in this setting, however, is not always decidable; with a preemptive scheduler, it is shown to be decidable for very restricted task models, e.g., when a task is modeled merely as a fixed computation time. In this paper, we consider the possibility of specifying tasks using timed automata. We show that, in this more complex setting, decidability holds not only for non-preemptive schedulers but also for preemptive schedulers if a minimum delay is assumed between consecutive preemptions. In practice, this minimum can be a multiple of the CPU clock speed. We show further how to extend from a single processor to multi-processor models with shared and/or separate queues.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management—*Scheduling*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Model Checking*

General Terms

Design, Verification

Keywords

Schedulability Analysis, Decidability, Multi-Processor, Timed Automata

1. INTRODUCTION

Schedulability analysis is checking whether all tasks in a real-time system can be scheduled and accomplished before their deadlines. Traditional approaches to the problem of scheduling consider an a priori given set of tasks with their duration of execution with one or more processors and try to find a possible ordering of tasks, with their assignment to different processors, such that all tasks finish before their deadlines. However, a reactive non-terminating system usually needs to process some (infinitely often) recurring tasks. There have been many works in the areas of scheduling of periodic to rate-monotonic tasks.

The introduction of task automata [6] has broaden the perspective of schedulability analysis to *non-uniformly* recurring tasks. Decidability issues have been studied for different settings of single- and multi-processor systems [6, 12]. It is shown that for nonpreemptive schedulers, this problem is decidable; whereas, for preemptive schedulers in general it is undecidable. In certain cases, for example if a task is merely represented by an exact execution time, the problem becomes decidable. Modeling a task simply by an execution time is not enough in many cases; e.g., if tasks need to generate sub-tasks, if there are dependencies between tasks, or if there are shared resources other than the processor.

In this paper, we study decidable settings of nonpreemptive to preemptive schedulers on single and multiple processor systems while *tasks are specified* themselves as timed automata. This is in line with [9, 10] that use timed automata for specifying task details in actor-based and concurrent objects settings. In this setting, an actor/object has a processor for executing its tasks. A new task on an actor/object is triggered by receiving a message which requires the execution of the corresponding method. The usage of an object is abstractly defined in its behavioral interface, which is in fact an extension of task automata (because tasks are specified here). Jaghoori et al. [9, 10] consider a nonpreemptive single processor setting. Their work has been applied [4, 8] to the concurrent object language Creol [11], where the possibility of voluntarily releasing the processor during execution of a method gives rise to a *cooperative scheduling* paradigm. Cooperative scheduling is in essence nonpreemptive and therefore it is not suitable for a wild environment, e.g., a general purpose operating system, but it is very powerful if used with care for example in embedded systems design; a task is not preempted in an unstable state whereas all tasks are willing to yield the processor when safe.

Our first contribution is extending cooperative scheduling to preemptive with minimum inter-preemption delays. This assumption is in reality not too restrictive because there is always such a delay as a multiple of the processor clock speed. By encoding this in a decidable class of timed automata, we show that this problem is decidable. Since we allow task behavior to be specified in timed automata, in

*This work is supported by EU FP7-231620 project: HATS.

SAC'11 March 21-25, 2011, TaiChung, Taiwan.

principle we need the power of stopwatches to be able to model preemption, which is undecidable in general. To be able to model preemption, we restrict task specification automata to use only one clock which in turn cannot be reset by the task itself; nonetheless, tasks can still generate other tasks and subtasks.

We require that there is a minimum delay *delta* between every two preemptions, and that *delta* should be a natural number. The execution time of tasks however need not be an exact integer value, e.g., a task could be modeled as a timed automaton that finishes between 2 and 3 time units. Furthermore, our model of the scheduler can start the next task immediately; i.e., for the above example, it does not have to wait for 3 time units to start the next task.

As our second contribution, we extend the schedulability analysis to a multiprocessor setting and show that schedulability remains decidable. It covers both cooperative and preemptive scheduling. We consider two settings for multiprocessing: shared and separate queues for different processors. The shared queue setting for nonpreemptive scheduling has been studied by de Boer et al. [5]. Further, we study multiple queues for multiple processors with or without task migration. We show a general technique for modeling a load balancing strategy between different processors.

The rest of the paper is organized as follows. We give the formal definitions of timed automata in Section 2. The basis of the encoding is the way to specify tasks behavior and their generation, explained in Section 3. In section 4 we review cooperative scheduling for single processor systems and then extend it to preemptive scheduling given a fixed minimum inter-preemption delay. Section 5 explains how to migrate from single to multiple processors. We prove decidability by reducing schedulability to the problem of reachability in timed automata. Section 6 concludes the paper.

Related Work. Our work builds on top of task automata [6] and therefore can handle nonuniformly recurring tasks. Krcal et al. [12] extend task automata to multi-processors. These works consider full-fledged preemptive scheduling and show its undecidability in general. They show that with unrestricted preemption, we need to dramatically restrict the task behavior specification to make schedulability analysis decidable, e.g., tasks are mere computation times and cannot affect generation of new tasks. Jaghoori et al. [9] apply schedulability analysis to object-oriented systems where a task corresponds to a method, and therefore it can generate subtasks; but they only consider nonpreemptive scheduling.

In this paper, we take the ideas of Jaghoori et al. in specifying tasks as timed automata and extend it to multiprocessor (with shared/separate queues, with/without task migration) and preemptive scheduling. The shared queue setting for nonpreemptive scheduling has been studied by de Boer et al. [5]. Due to the undecidability results of task automata, we do not handle full-fledged preemption; instead, to allow more flexible task specifications, we restrict the scheduler such that there is always a minimum delay between two preemptions.

Decidable classes of task automata [6] with preemptive schedulers are restricted to priority-based scheduling, e.g., Fixed-Priority Scheduling and Earliest-Deadline First. With the assumption of a minimum delay between preemptions, as by de Boer et. al [5], we can handle any kind of scheduling, e.g., a round-robin preemptive scheduler.

Like the works mentioned above [6, 9, 12], we deal with schedulability analysis of design models (specified in automata), and not directly with analysis of code implemented in programming languages, as [3] do. With the techniques we provide, one can decide about application-specific scheduling strategies at design time. To ensure schedulability of an implemented program one needs to check its conformance with the schedulable design models.

2. PRELIMINARIES

In this section, we define timed automata [1] as it forms the basis of our analyses. Suppose $\mathcal{B}(C)$ is the set of all clock constraints on the set of clocks C. A timed automaton over actions Σ and clocks C is a tuple $\langle L, l_0, \longrightarrow, I\rangle$ representing

- a finite set of locations L (including initial location l_0);
- the set of edges $\longrightarrow \subseteq L \times \mathcal{B}(C) \times \Sigma \times 2^C \times L$; and,
- a function $I : L \mapsto \mathcal{B}(C)$ assigning an invariant to each location.

An edge (l, g, a, r, l') implies that action 'a' may change the location l to l' by resetting the clocks in r, if the clock constraints in g (as well as the invariant of l') hold. Since we use UPPAAL [13], we allow defining variables of type boolean and bounded integers. Variables can appear in guards and updates; clocks can be reset to integer values.

Networks of timed automata. A system may be described as a collection of communicating timed automata. In these automata, the action set is partitioned into input, output and internal actions. The behavior of the system is defined as the parallel composition of those automata $A_1 \parallel \cdots \parallel A_n$. Semantically, the system can delay if all automata can delay and can perform an action if one of the automata can perform an internal action or if two automata can synchronize on complementary actions (inputs and outputs are complementary). In a network of timed automata, variables can be defined locally for one automaton, globally (shared between all automata), or as parameters to the automata.

Marking a location *urgent* indicates that the automaton cannot spend any time in that location. This is equivalent to resetting a fresh clock x in all of its incoming edges and adding an invariant $x \leq 0$ to the location. In a network of timed automata, the enabled transitions from an urgent location may be interleaved with the enabled transitions from other automata (while time is frozen). Like urgent locations, *committed* locations freeze time; furthermore, if any process is in a committed location, the next step must involve an edge from one of the committed locations.

3. TASK GENERATION AND SPECIFICATION

We explain how to model task generation and specifications with timed automata in UPPAAL. This enables verification right away. Task names are defined as unique integer constants so we can refer to task types by their names. A task generation automaton uses the channel invoke to send a message; the name of the message sent is assigned to the global variable *msg* and its deadline is given in the variable *deadline*. Receiving a message triggers a new task. In Figure 1, Task1 and Task2 are generated intermittently, while the generation speed is controlled by the constant SPEED.

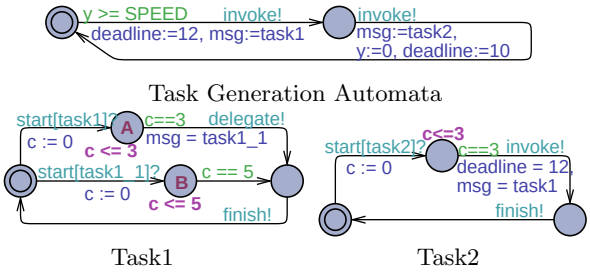

Figure 1: Task Generation and Specification

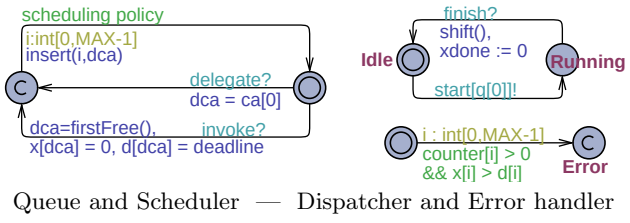

Figure 2: Cooperative scheduling, single processor

An automaton specifying a task (or subtask) behavior starts by waiting for synchronization on the *start* channel and finishes with the *finish* channel; the task behavior is modeled in between. This way we can control a task's execution. The channels start and finish are urgent, i.e., once a synchronization is enabled it is taken without delay; thus context-switch between two tasks takes no time. The finish transition goes back to the initial location to make it possible to execute the task again later. It is important that all locations in task specification automata have a time upper bound as an invariant, otherwise, schedulability is trivially not satisfied.

Task1 and Task2 need 8 and 3 time units for complete execution, respectively. Execution of Task2 generates a new instance of Task1 using the invoke channel during its execution. Task1 is modeled in two parts. First it goes to location A where after 3 time units, it releases the processor, i.e., it *delegates* the rest of the task as a subtask Task1_1 to be scheduled again in the queue and then finishes. The *start* transition for subtask Task1_1 goes to location B which completes the remainder of Task1 in 5 time units.

In each round of the task generation automaton, two instances of Task1 and one instance of Task2 are generated; their execution time adds up to 19. A single processor system is therefore schedulable only if SPEED is at least 19.

4. SINGLE PROCESSOR: FROM COOPERATIVE TO PREEMPTIVE SCHEDULING

It is shown in [9] that a queue length of MAX=$\lceil d_{max}/b_{min} \rceil$ is enough for schedulability analysis of single processor systems: If there are more tasks in the queue at a time, they inevitably require more than d_{max} to finish (because each task takes more than b_{min} to execute), which implies non-schedulability.

We repeat the cooperative scheduling framework of [9] with the difference that we take dispatcher and error handler out of the scheduler automata (see Figure 2). This simplifies our extensions in later parts. To analyze a system in UPPAAL, we make a network of timed automata with instances of task generation and specifications automata and the automata for scheduling, task dispatching and error handling.

4.1 Cooperative Scheduling

Cooperative scheduling means that tasks may release the processor before their completion, e.g., Task1 as specified in Section 3. Assume that Task2 is triggered immediately after Task1. When Task1 releases the processor, Task2 finds a chance for execution before the subtask Task1_1. Otherwise, Task2 would have no chance of meeting its deadline with a nonpreemptive scheduler. Nevertheless, we need an Earliest Deadline First (EDF) scheduler to make sure Task2 is scheduled before Task1_1.

To store tasks we use an array q, which holds the tasks in the order they should be executed ($q[0]$ is currently executing). The function *insert* puts a new task in its right place based on the scheduling policy. As the scheduler is nonpreemptive, it can never put a task at $q[0]$ unless the queue is empty. When the processor is idle, the *dispatcher* automaton starts $q[0]$ (see Figure 2). When a task finishes, the queue q is shifted to remove the finished task. The clock x_{done} is reset whenever a task finishes. x_{done} can be tested in the task generation automata (as is the tradition in Task Automata [6]).

With a queue of length MAX we associate arrays x and d of MAX clocks and integers. The clock assigned to the task $q[i]$ is determined by $ca[i]$. When $q[i]$ is added to the queue $x[ca[i]]$ is reset and $d[ca[i]]$ is set to its deadline; the remaining deadline of $q[i]$ at any later point is calculated by $d[ca[i]] - x[ca[i]]$. When receiving a message on invoke, the 'first free' clock is selected and assigned to the new task. For a subtask generated on delegate channel, the clock assigned to currently running task is reused; thus, the subtask inherits the deadline of the original task. *counter* keeps track of the number of tasks using each clock.

The transition of error handler is taken if a clock (with a nonzero *counter*) exceeds its deadline. The property to check for schedulability analysis is:

E <> ErrHandler.Error || tail > MAX

EDF Strategy. EDF needs to dynamically check clock values of all tasks to compute their remaining deadlines. Therefore it has to be implemented as a guard to select the proper i for inserting the new task. One can specify EDF in UPPAAL by writing the following guard in the queue/scheduler automaton:

```
(i == 0 and tail == 0) or
(i > 0 and i ≤ tail and
 forall (m : int[1,MAX−1])
   (m ≥ i or x[ca[m]] − x[dca] ≥ d[ca[m]] − d[dca])
 and forall (m : int[1,MAX−1])
   (m < i or m ≥ tail or
    x[ca[m]] − x[dca] < d[ca[m]] − d[dca])
)
```

The first line distinguishes the case of an empty queue when 0 is chosen for i. In other cases, it cannot choose 0 which is the running task (the second line). By using the two **forall** expressions, we find the point where the remaining deadline of the new (sub)task, i.e., d[dca]−x[dca], fits. If there are multiple tasks with the same priority (i.e., remaining

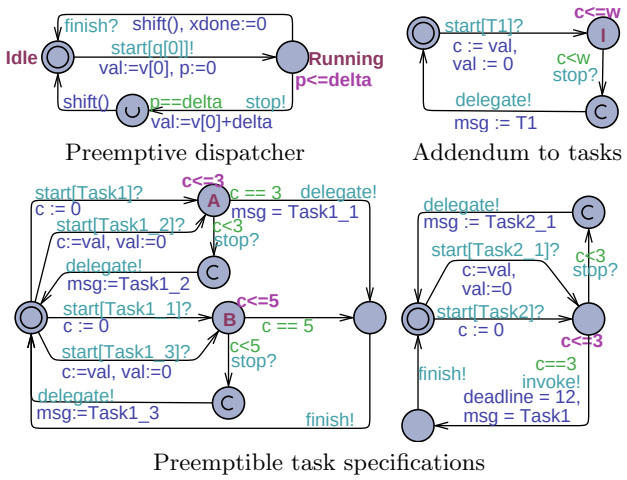

Figure 3: Changes for preemptive scheduling

deadline), they will be handled in a first-come first-served manner; therefore, the choice of i is deterministic.

4.2 Preemptive Scheduling

Preemptive scheduling in general is undecidable for task automata in single-processor [6] and multi-processor settings [12]. Since we can encode the simple model of tasks with feedback used in timed automata in our framework, this undecidability result will carry over to our setting. However, we show below that if a fixed minimum delay is always assumed between every two preemptions, the problem becomes decidable. If we consider the clock speed of the processor as time unit, this delay can be represented as a natural number. We call this delay *delta*.

De Boer et. al [5] reported on a method of integer preemption. Like task automata, they restrict task specifications to mere execution times. Unlike task automata, they handle a round-robin scheduler. In this chapter, we explain how to model integer preemption while tasks are specified as timed automata which may create other tasks or subtasks. We restrict task specification automata such that only one clock is used to keep track of time during task execution and this clock is not reset by the task itself.

Preemptible Task Specifications. Given task specifications in timed automata (as in previous section), we explain below a method for automatically adding transitions to these automata to make them preemptible. We model preemptions that happen after *delta* time units (unless the currently running task is finished before preemption). The 'addendum' in Figure 3 shows three transitions that should be added to every location l of all task specification automata, except for the initial and final locations (before the finish transition). Below we explain these transitions.

- We add a channel called stop. When the task is at location l, it may be preempted by synchronizing on the channel stop. The stop transition is taken only if the clock c is strictly smaller than w, where w is the time upper bound on location l. This is to ensure maximal progress before preemption, i.e., for example, a task is not preempted if it can finish in zero time. The decision when a task should be preempted is taken by the dispatcher automaton explained later.

- When a task is stopped, the rest of it should be put back in the queue as a subtask. This is done using the delegate channel. T1 used in this figure should be replaced with a new subtask name (defined as a unique integer constant in the UPPAAL model). If (the remainder of) the running task has the highest priority in the queue, it will be restarted immediately. This is naturally handled by the delegation mechanism.

- When a task is preempted, in principle its clock has to be stopped. Arbitrarily stopping clocks makes our timed automata models undecidable [1]. Since we only allow preemption at integer points, we show how to simulate this behavior without really stopping the clocks. The idea is that the queue automaton remembers the clock value when it should be stopped; we add the array v to store these integer clock values. When the task is resumed (start [T1]?) we reset the clock to this value ($c := val$). The value of val is set by the dispatcher according to the stored value in the queue. The reason val is reset to zero afterwards is explained with the dispatcher automaton.

Figure 3 shows how the specifications of Task1 and Task2 are updated. The new subtasks for these automata are Task1_2, Task1_3 and Task2_1. As shown in this example, preemptible tasks can be modeled on top of cooperative models; i.e., we can use the cooperative model of Task1 that releases the processor during execution. This gives us more flexibility in scheduling of tasks.

Preemptive Dispatcher. The dispatcher automaton cannot stay in the Running location more than *delta* time units, modeled as an invariant on the new clock p. At this point, it preempts the currently running task, it removes this task from the queue by shifting, and then dispatches the first task in the queue, i.e., $q[0]$.

As explained in the last bullet above, we add an array v to the queue to remember the last value of a task clock at the time of preemption. The insert function assigns the last value of val to the corresponding element of v. For new tasks (triggered on the invoke channel) val is zero; this is ensured by resetting val on the start transition in the addendum to task specifications, explained above. When a task is started, the dispatcher assigns the stored value $v[0]$ to val which is then assigned to the computation clock c of the started task. The task is preempted exactly after *delta* time units; therefore, the dispatcher assigns $v[0] + delta$ to val which is then used by the insert function for the new instance of the subtask. To handle the v element of the queue for each task, the shift function of the queue is updated in a straightforward manner. The structure of the queue/scheduler automaton is not changed.

The choice of urgent and committed locations in the dispatcher and addendum of task specifications is critical. After a synchronization on stop, the synchronization on delegate happens; remember from Figure 2 that afterwards the task is inserted in the queue (due to a committed location in the queue/scheduler automaton). Then, only after all committed locations are processed, the urgent location in the dispatcher is processed and the queue is shifted. This is im-

portant because the insert function still cannot put a task at $q[0]$, which is in turn because the running task cannot be preempted at any arbitrary time. Performing the shift at the end gives the new subtask to be at the head of the queue after shift (if it has the highest priority).

After stopping a task, its automaton will be in initial location (see the addendum in Figure 3). This allows execution of other instances of the same task type if there are many of them in the queue and the scheduling strategy requires so. This approach can be used to implement a round-robin behavior when multiple tasks in the queue have equal priority. In resolving equal priorities, usually the first-come first-served strategy is applied. By using delegation for preemption, the currently running task will be put at the back of other tasks with equal priority.

If one wishes to keep first-come first-served strategy for resolving equal priorities, but does not wish to give a round-robin behavior, i.e., when a high-priority task is started it will not be intervened by tasks of equal priority, one should add another channel for preemption, with similar behavior as delegate, except that it handles equal priorities in a last-come first-served manner.

5. FROM SINGLE TO MULTI-PROCESSOR

We describe below how the task and scheduler models of single-processor systems described above can be extended to a multi-processor setting. We show this extension for the simpler case of cooperative scheduling, however, the extension for preemptive scheduling is the same. We identify two ways of scheduling in multi-processors: A shared queue between the processors or separate queues. The former case is more suitable for a multi-core environment, while the latter is in line with the traditional parallel computing approach. In each setting, there are two possibilities with respect to task migration. Task migration means that different pieces of a task may get executed on different processors. With a shared queue it is natural to allow task migration assuming a negligible penalty for it, while with separate queues it is easier to disallow task migration. We will, however, consider allowing and disallowing task migration in both cases.

In multi-processor with n processors, in principle n tasks can be executed at the same time. This means that the length of schedulable queues should be multiplied by n. In the setting of shared queues, we need to analyze a queue with a length of up to $n \times \lceil d_{max}/b_{min} \rceil$. When there is a queue for each processor, each queue should be analyzed up to a length of $\lceil d_{max}/b_{min} \rceil$.

5.1 Multi-Core: Shared Queue

Since the queue is shared between the cores, there will be one instance of the queue/scheduler automaton. To model a multi-core system, the idea is to replicate the task dispatcher automaton per core. This also requires instantiating the task specification automata once per core. These automata will be parameterized in processor-core identity (cf. Figure 4). Parameterizing the task specification automata requires parameterization of the start and finish channels to be able to interact properly with the respective dispatcher automaton.

The queue is divided in two parts: the running tasks and waiting tasks. There are as many running tasks as there are cores, given by an integer constant CORES. To respect the nonpreemptive nature of the scheduler, it cannot insert a task in the first CORES elements of the queue (corresponding to the running tasks) unless they are empty. This affects the implementation of the insert function. Furthermore, the shift function is updated such that it does not shift the running tasks; instead it shifts the waiting tasks and assigns the next task to the core that has just finished.

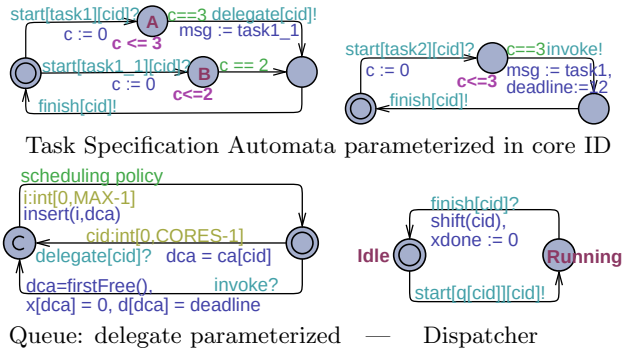

Figure 4: Cooperative scheduling, Multi-core

When a subtask is generated, the delegate channel uses the core ID to identify the running task that triggered the subtask. This helps the scheduler find the clock assigned to the parent task, which should be reused for the subtask (in order to inherit the remaining deadline). Nevertheless, the subtask is simply put back in the queue (i.e., the insert function is not changed); this means that the subtask may be executed on a different processor from the parent task.

To disallow task migration, we can add an extra element to the queue for each task to keep track of the core ID on which it has been assigned before. This value is initially set to a value that does not correspond to any core ID. The insert function will use the core ID reported by the delegate channel to fill this element. The shift function can assign a subtask to a core only if it has the same ID.

5.2 Multi-Processor: Multiple Queues

To model a multiple queues, we need to replicate the queue, in addition to the dispatcher and error handler, for every processor. To this end, the automata must be parameterized in processor identifiers. This enables each queue/processor to define its own local scheduling strategy.

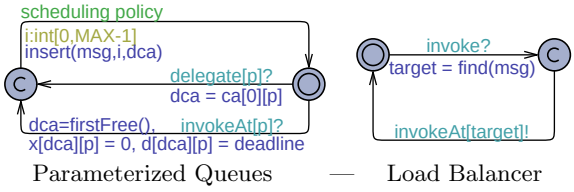

Figure 5: Multi-Processor with Separate Queues

Like in multi-core, task specification automata are parameterized in processor ID, which means that start and finish channels are also parameterized. In the case of subtask generation (along delegate channel) the task will be added to the queue of the same processor (no task migration).

New tasks (that are generated on the invoke channel) should be spread on different queues. One way to achieve this is to require the task generation and specification automata to specify directly the target queue for each task. This means that the invoke channel will be parameterized in processor ID, too.

An alternative and more elegant solution is adding a load balancer automaton. The load-balancer intervenes every message and directs it to the processor with smaller load; or in general, based on its load balancing strategy. To do so, the load balancer receives all communication on the invoke channel. Instead the queue automata receive messages on the new invokeAt channel, which is parameterized in the processor ID. The load balancer forwards the messages along invokeAt channel to the proper target queue (see Figure 5).

The load balancing strategy is implemented in the find function in this figure. The function find could look into the queues of different processors for load balancing, or could simply decide based on the message type. If the load balancing strategy should be based on the remaining deadlines of tasks, which involves checking clock values, the strategy has to be implemented as a guard, similar to the EDF scheduling policy explained in Section 4.1.

To allow task migration across queues, one can define a new channel delegateAt, in much the same way as invokeAt. The load balancer then needs to receive all delegated subtasks and decide whether a new target results in better overall performance.

6. DISCUSSION AND CONCLUSION

In this paper, we focused on the practical side of schedulability analysis, i.e., decidable cases. To escape from undecidability of preemptive scheduling, Fersman et al. [6] simplify the task model. On the contrary, we allow task behaviors to be specified with timed automata and restrict the scheduler: we assume a minimum delay between preemptions, which in reality could be a multiple of the CPU clock speed.

We extended our results to multi-processors, with shared and separate queues, with or without task migration. There are many ways to assign a scheduler: to each queue, to each processor, their combination or hierarchical. For multiple queues, we showed a simple way to implement a load balancer on top of queue-specific schedulers. Hierarchies of schedulers can be implemented by multiple levels of the invoke channel as we showed for the load balancer. It is to be noted that in all settings, the task generation automaton is the same and applies to the whole single or multiple processor system.

We showed decidability by encoding the problem in a decidable class of timed automata [2]. However, problems in the forward analysis algorithm in UPPAAL make it unable to handle the preemptive models. This seems to be related to use of integer clock assignments, checking clock differences (to formulate EDF) and channel synchronizations.

It is interesting to study how a time penalty for task migration in multi-processor systems could affect the schedulability results. This could be implemented with a penalty function in the load balancer automaton when dispatching subtasks using the delegateAt channel.

Our goal in this paper was to show the decidability of the schedulability problem in different settings by reducing it to reachability of timed automata. In our encoding, there is one clock for each queue slot; this is anyway necessary for earliest deadline first strategy. For fixed priority scheduling, one needs less clocks. One can improve the performance by using less clocks as by Fersman et al. [7].

An independent line of future research is to pursue automatic code generation from schedulable design models. Alternatively, one can study conformance of programs to high-level schedulable designs.

7. REFERENCES

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[2] P. Bouyer, C. Dufourd, E. Fleury, and A. Petit. Are timed automata updatable? In *Proc. Computer Aided Verification (CAV'00)*, volume 1855 of *LNCS*, pages 464–479, 2000.

[3] E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venter, D. Weil, and S. Yovine. TAXYS: A tool for the development and verification of real-time embedded systems. In *Proc. CAV'01*, volume 2102 of *LNCS*, pages 391–395. Springer, 2001.

[4] F. de Boer, T. Chothia, and M. M. Jaghoori. Modular schedulability analysis of concurrent objects in Creol. In *Proc. Fundamentals of Software Engineering (FSEN'09)*, volume 5961, pages 212–227, 2009.

[5] F. S. de Boer, I. Grabe, M. M. Jaghoori, A. Stam, and W. Yi. Modeling and analysis of thread-pools in an industrial communication platform. In *Proc. 11th International Conference on Formal Engineering Methods (ICFEM'09)*, volume 5885 of *LNCS*, pages 367–386. Springer, 2009.

[6] E. Fersman, P. Krcal, P. Pettersson, and W. Yi. Task automata: Schedulability, decidability and undecidability. *Information and Computation*, 205(8):1149–1172, 2007.

[7] E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Schedulability analysis using two clocks. In *Proc. Tools and algorithms for the construction and analysis of systems (TACAS'03)*, pages 224–239, Berlin, Heidelberg, 2003. Springer-Verlag.

[8] M. M. Jaghoori and T. Chothia. Timed automata semantics for analyzing Creol. In *Proc. Foundations of Coordination Languages and Software Architectures (FOCLASA'10)*, EPTCS 30, pages 108–122, 2010.

[9] M. M. Jaghoori, F. S. de Boer, T. Chothia, and M. Sirjani. Schedulability of asynchronous real-time concurrent objects. *J. Logic and Alg. Prog.*, 78(5):402 – 416, 2009.

[10] M. M. Jaghoori, D. Longuet, F. S. de Boer, and T. Chothia. Schedulability and compatibility of real time asynchronous objects. In *Proc. RTSS'08*, pages 70–79. IEEE CS, 2008.

[11] E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1-2):23–66, 2006.

[12] P. Krcal, M. Stigge, and W. Yi. Multi-processor schedulability analysis of preemptive real-time tasks with variable execution times. In *Proc. Formal Modeling and Analysis of Timed Systems*, volume 4763 of *LNCS*, pages 274–289, 2007.

[13] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *STTT*, 1(1-2):134–152, 1997.