A Generic Approach to Schedulability Analysis of Real-Time Tasks

Elena Fersman and Wang Yi
Uppsala University
Department of Information Technology
P.O. Box 337, S-751 05 Uppsala, Sweden
Email: {elenaf,yi}@it.uu.se

Abstract. In off-line schedulability tests for real time systems, tasks are usually assumed to be periodic, i.e. they are released with fixed rates. To relax the assumption of complete knowledge on arrival times, we propose to use timed automata to describe task arrival patterns. In a recent work, it is shown that for fixed priority scheduling strategy and tasks with only timing constraints (i.e. execution time and deadline), the schedulability of such models can be checked by reachability analysis on timed automata with *two* clocks.
In this paper, we extend the above result to deal with precedence and resource constraints. This yields a unified task model, which is expressive enough to describe concurrency, synchronization, and tasks that may be periodic, aperiodic, preemptive or non-preemptive with (or without) combinations of timing, precedence, and resource constraints. We present an operational semantics for the model, and show that the related schedulability analysis problem can be solved efficiently using the same technique. The presented results have been implemented in the TIMES tool for automated schedulability analysis.

1 Introduction

In the development of real time systems, there are typically three types of constraints specified on tasks: *timing* constraints such as (relative) deadlines, *precedence* constraints specifying a (partial) execution order of a task set, and *resource* constraints given as critical sections in which mutually exclusive access to shared data must be guaranteed. For many applications, we may have to deal with combinations of these constraints, and guarantee that the constraints are satisfied (known a priori to the final system implementation).

In scheduling theory, the general approach to resolving tasks constraints is through *task models* modeling the abstract behaviours of tasks. Roughly speaking, task models are task arrival patterns such as periodic or sporadic, describing how the tasks are released or triggered. Task models also provide information on resource requirements such as worst case computing times, shared semaphores etc. Based on task models, scheduling algorithms (or scheduling strategies) can be synthesized to schedule the tasks execution so that the given constraints are satisfied. Alternatively, given a scheduling strategy such as FCFS, using task models, we can verify whether a task set

is schedulable, that is, the given constraints can be met or not. The later is known as *schedulability analysis*. It has been a central notion in research on real time systems.

For periodic tasks, i.e. tasks that are released and computed with fixed rates periodically, there are well-developed techniques for schedulability analysis. We mention Liu and Layland's pioneer work on rate-monotonic analysis [17], and Joseph and Pandya's response-time analysis [15]. In the past years, these classic works have been extended to deal with more complex constraints e.g. offset analysis [18] and unfolding [6] for precedence constraints and priority ceiling protocols [20, 19] for shared resources.

The limitation of simple task models like periodic ones is obvious. Besides the restricted expressiveness for modeling, the analysis for schedulability is often based on the worst case scenarios, and therefore may be pessimistic in many cases. In recent years, several automata-theoretic approaches to modeling, scheduling and controller synthesis have been presented, e.g. [2, 8, 3, 11, 10]. To relax the stringent constraints on task arrival times, such as fixed periods, we have proposed to use timed automata to describe task arrival patterns [11], which provides a generic task model. The model is expressive enough to describe concurrency, synchronization, and real time tasks which may be periodic, sporadic, preemptive or non-preemptive. In a recent work [10], it is shown that for fixed priority scheduling strategy, the schedulability analysis problem for the task model *without precedence and resource constraints* can be efficiently solved by reachability analysis on timed automata using only 2 extra clock variables.

In this paper, we extend our previous results to deal with precedence and resource constraints. As the main result of this paper, we show that schedulability analysis problem for the automata-based task model in connection with combinations of timing, precedence, and resource constraints can be solved efficiently using timed automata technology. Again, the number of extra clock variables needed in the analysis is 2 just like our previous result [10] for models without precedence and resource constraints. We present algorithms and their implementation in the TIMES tool for automated schedulability analysis. Indeed, the analysis can be done in a similar manner to response time analysis in classic Rate-Monotonic Analysis. In addition, other techniques such as model checking to verify logical as well as temporal correctness can be applied within the same framework prior to schedulability analysis.

The rest of this paper is organized as follows: in the next section we present a generic task model combining standard task parameters and constraints with automata-theoretic approaches to system modelling. In section 3, we present operation semantics of the model. In section 4 we show how to check the model for schedulabilty using timed automata. In section 5 we describe the implementation of the presented results in the TIMES tool. Section 6 concludes the paper.

2 A Generic Task Model

In this section, we present a unified task model combining the standard notion of tasks, task parameters and tasks constraints with automata-based approaches to system modelling.

2.1 Tasks Parameters and Constraints

Task A *task* (or task type) is an executable program. A real time system may contain a set of tasks scheduled to run over a limited number of shared resources. We shall distinguish task type and task instance. A *task type* may have different *task instances* that are copies of the same program with different inputs. When it is understood from the context, we shall use the term *task* for task type or task instance. A task may have task parameters such as fixed priority, execution time (for a given platform), deadline etc. Some of these parameters will be considered as task constraints in the following.

Now, let $\mathcal{P}$ ranged over by P_1, P_2 etc, denote a finite set of task types. The task instances will be released according to pre-specified patterns such as periodic tasks etc. In the next subsection, we describe task arrival patterns based on timed automata. The released tasks will be inserted in the task queue and scheduled according to given task constraints. Following the literature [7], we consider three types of task constraints.

Timing Constraints A typical timing constraint on a task is deadline, i.e. the time point before which the task should complete its execution. We assume that the *worst case execution times* and *hard deadlines* of tasks in $\mathcal{P}$ are known (or pre-specified). Thus, each task P is characterized as a pair of natural numbers denoted (C, D) with $C \leq D$, where C is the execution time of P, D is the relative deadline for P. The deadline D is a relative deadline meaning that when task P is released, it should finish within D time units. We shall use

- $C(P)$ to denote the execution time of P and
- $D(P)$ to denote the relative deadline of P.

Note that in addition to deadlines, we also view execution times as (timing) constraints meaning that the tasks can not consume more than the given execution times.

Precedence Constraints In many applications, tasks may have to respect some precedence relations to express for example, input and output relations between tasks, and data dependencies in data-flow diagrams. These relations are usually described through a precedence graph, which induces a partial order on a task set.

A precedence graph is a directed acyclic graph in which nodes represent tasks and edges represent precedence relation, e.g. an edge from P to Q (denoted $P \rightarrow Q$) requires that task P must be completed before Q may start to execute. We may allow the more general form of precedence constraints e.g. the AND/OR-precedence graphs [12]. But for the presentation, we shall only consider partial orders in the form of $P_i \rightarrow P_j$ over the task set.

Resource Constraints We distinguish two types of resources: hardware resources and logical resources. The only hardware resource we consider is processor time, and constraints related to processor time are normally known as timing constraints as described above. Logical resources are in general software resources i.e. shared data; however they can be hardware devices accessed through semaphores. We use semaphores to represent logical resources.

Assume a set of semaphores S ranged over by s. representing the set of shared resources for all tasks. We assume that each semaphore s is associated with a set of shared variables $Var(s)$ (protected by the semaphore). Naturally, we require $Var(s) \cap Var(s') = \emptyset$ for $s \neq s'$.

The *semaphore access pattern* of a task P is a sequence of triples: $\{(\mathsf{T_i}, \mathsf{OP_i}, \mathsf{A_i})\}$ where

- $\mathsf{OP_i}$ is the semaphore operation to be performed on a semaphore. Let $\mathsf{OP_i} = \mathsf{p(s)}$ mean locking of s and $\mathsf{OP_i} = \mathsf{v(s)}$ mean unlocking of s.
- $\mathsf{T_i}$ is the computation time for performing $\mathsf{A_i}$.
- $\mathsf{A_i}$ is a sequence of assignments updating the set of variables associated with semaphores s when $\mathsf{OP_i} = \mathsf{v(s)}$, and the updating should be committed at the same time point as when the v-operation is taken.

We shall simply view a semaphore access pattern as a task; clearly, the execution time of such a task is the sum of computing times for all the triples i.e. $\sum \mathsf{T_i}$. We require that that the patterns of semaphore operations (p and v) should be well-formed in the sense that no semaphores are left locked, or unlocked without a preceding locking operation. More precisely for each s, we require that any occurrence of $(p(s), \mathsf{T_i}, \mathsf{A_i})$ should be followed by $(v(s), \mathsf{T_j}, \mathsf{A_j})$ in a semaphore access pattern; however in between of the two, there may be operations on other semaphores.

Given a semaphore access pattern for each task of a task set, the problem is how to schedule the task set to access the semaphores properly so that no one is blocked unbounded number of times (and thus delayed forever). In the literature, various solutions for this problem using resource access protocols have been developed [7]. In the following sections, we shall describe the precise semantics of these protocols and present a uniformed approach to schedulability analysis for these protocols.

2.2 Timed Automata as Task Arrival Patterns

In the literature, most of the published works in real time scheduling are based on the simple task arrival pattern: tasks are assumed to be released with fixed rates periodically though a few variants e.g. sporadic tasks [7] and periodic tasks with release jitters [21] have been studied based on slightly relaxed periodicity assumption. The obvious advantage is that we have simple and efficient analysis procedures for periodic behaviours. But the analysis results may be pessimistic in many cases.

In [8] and [11], we have proposed to use timed automata as task arrival patterns. The advantage is that we may describe real time tasks that can be triggered and released according to a much more expressive structure i.e. state machines with clock constraints. The idea is to annotate each transition of a timed automaton with a task that will be triggered when the transition is taken. The triggered tasks will be scheduled to run according to a given scheduling strategy.

However, in these previous work, we only allow tasks with timing constraints i.e. execution times and deadlines. In this paper, we adopt a more expressive version of timed automata with tasks. We shall also allow tasks imposed with precedence and resource constraints. The later also implies that tasks may have shared variables. However, access

to shared data should follow the resource constraints as described previously. In addition, we also allow an automaton and its annotated tasks to have shared variables. The shared variables may be updated by the execution of a task (but not the automaton) and their values may be read by the automaton and effect the behaviour of the automaton.

Timed Automata. Assume a finite set of actions Act and a finite set of real-valued variables $\mathcal{C}$ for clocks. We use a, b, τ etc. to range over Act, where τ denotes a distinct internal action, and x_1, x_2 etc to range over $\mathcal{C}$. We use $\mathcal{B}(\mathcal{C})$ to denote the set of conjunctive formulas of atomic constraints in the form: $x_i \sim C$ or $x_i - x_j \sim D$ where $x_i, x_j \in \mathcal{C}$ are clocks, $\sim \in \{\leq, <, =, \geq, >\}$, and C, D are natural numbers. We use $\mathcal{B}_I(\mathcal{C})$ for the subset of $\mathcal{B}(\mathcal{C})$ where all atomic constraints are of the form $x \prec C$ and $\prec \in \{<, \leq\}$. The elements of $\mathcal{B}(\mathcal{C})$ are called *clock constraints*.

Definition 1. *A timed automaton extended with tasks, over actions Act, clocks $\mathcal{C}$ and tasks $\mathcal{P}$ is a tuple $\langle N, l_0, E, I, M\rangle$ where*

- *$\langle N, l_0, E, I\rangle$ is a timed automaton where*
  - *N is a finite set of locations ranged over by l, m, n,*
  - *$l_0 \in N$ is the initial location, and*
  - *$E \subseteq N \times \mathcal{B}(\mathcal{C}) \times Act \times 2^{\mathcal{C}} \times N$ is the set of edges.*
  - *$I : N \mapsto \mathcal{B}(\mathcal{C})$ is a function assigning each location with a clock constraint (a location invariant).*
- *$M : N \hookrightarrow \mathcal{P}$ is a partial function assigning locations with tasks* [1].

In the next section, we shall present the precise meaning of an automaton. Intuitively, a discrete transition in an automaton denotes an event triggering a task and the guard (clock constraints) on the transition specifies all the possible arrival times of the event (or the associated task). Whenever a task P is triggered, it will be put in the scheduling (or task) queue for execution (corresponding to the ready queue in operating systems). The scheduler should make sure that all the task constraints are satisfied in scheduling the tasks in the queue.

To handle concurrency and synchronisation, parallel composition of extended timed automata may be introduced in the same way as for ordinary timed automata (e.g. see [16]) using the notion of synchronisation function [13]. For example, consider the parallel composition $A||B$ of A and B over the same set of actions Act. The set of nodes of $A||B$ is simply the product of A's and B's nodes, the set of clocks is the (disjoint) union of A's and B's clocks, the edges are based on synchronisable A's and B's edges with enabling conditions conjuncted and reset-sets unioned. Note that due to the notion of synchronisation function [13] , the action set of the parallel composition will be Act and thus the task assignment function for $A||B$ is the same as for A and B.

[1] Note that M is a partial function meaning that some of the locations may have no task. Note also that we may associate a location with a set of tasks instead of a single one. It will not cause technical difficulties.

3 Operational Semantics

Semantically, an extended timed automaton may perform two types of transitions just as standard timed automata. But the difference is that delay transitions correspond to the execution of running tasks with the highest priority (or earliest deadline) and idling for the other tasks waiting to run. Discrete transitions correspond to the arrival of new task instances.

We use valuation to denote the values of variables. Formally a valuation is a function mapping clock variables to the non–negative reals and data variables to the data domain. We denote by $\mathcal{V}$ the set of valuations ranged over by σ. Naturally, a semantic state of an automaton is a triple (l, σ, q) where l is the current control location, σ denotes the current values of variables, and q is the current task queue. We assume that the task queue takes the form: $[P_1, P_2 ... P_n]$ where P_i denotes a task instance. We use $c(P_i)$ and $d(P_i)$ to denote the remaining computing time and relative deadline of P_i. Note that for tasks with timing constraints only, each P_i is simply a pair $(c(P_i), d(P_i))$ and for tasks with resource constraints, P_i is a list of triples i.e. a semaphore access pattern as described earlier.

Assume that there is a processor running the released task instances according to a certain scheduling strategy Sch e.g. FPS (fixed priority scheduling) or EDF (earliest deadline first) which sorts the task queue whenever a new task arrives according to task parameters e.g. deadlines. In general, we assume that a scheduling strategy is a sorting function which may change the ordering of the queue elements only. Thus an action transition will result in a sorted queue including the newly released tasks by the transition. A delay transition with t time units is to execute the task in the first position of the queue for t time units. Thus the delay transition will decrease the computing time of the first task by t. If its computation time becomes 0, the task should be removed from the queue (shrinking).

- Sch is a sorting function for task queues (or lists), that may change the ordering of the queue elements only. For example, $\mathsf{EDF}([P(3.1, 10), Q(4, 5.3)]) = [Q(4, 5.3), P(3.1, 10)]$. We call such sorting functions scheduling strategies that may be preemptive or non-preemptive.
- Run is a function which given a real number t and a task queue q returns the resulted task queue after t time units of execution according to available computing resources. For simplicity, we assume that only one processor is available. Then the meaning of $\mathsf{Run}(q, t)$ should be obvious and it can be defined inductively. For example, let $q = [Q(4, 5), P(3, 10)]$. Then $\mathsf{Run}(q, 6) = [P(1, 4)]$ in which the first task is finished and the second has been executed for 2 time units.

Further, for a real number $t \in \mathcal{R}_{\geq 0}$, we use $\sigma + t$ to denote the valuation which updates each clock x with $\sigma(x) + t$, $\sigma \models g$ to denote that the valuation σ satisfies the guard g and $\sigma[r]$ to denote the valuation which maps each variable α to the value of $\mathcal{E}$ evaluated in σ if $\alpha := \mathcal{E} \in r$ (note that $\mathcal{E}$ is zero if α is a clock) and agrees with σ for the other variables.

Timing constraints Now we present the transition rules for automata extended with tasks imposed with timing constraints (only):

Definition 2. *Given a scheduling strategy* Sch[2]*, the semantics of an extended timed automaton* $\langle N, l_0, E, I, M\rangle$ *with initial state* (l_0, u_0, q_0) *is a transition system defined by the following rules:*

- $(l, u, q) \xrightarrow{a}_{\mathsf{Sch}} (m, u[r \mapsto 0], \mathsf{Sch}(M(m) :: q))$ *if* $l \xrightarrow{g,a,r} m$ *and* $u \models g$
- $(l, u, q) \xrightarrow{t}_{\mathsf{Sch}} (l, u + t, \mathsf{Run}(q, t))$ *if* $(u + t) \models I(l)$

where $M(m) :: q$ *denotes the queue with* $M(m)$ *inserted in* q*.*

The first rule defines that a discrete transition can be taken if there exists an enabled edge, i.e. the guard of the edge is satisfied. When the transition is taken, the variables are updated according to the resets of the edge, and the tasks of the action (if any) are inserted into the queue.

In the second rule, the delay transition corresponds to that the first task in the queue is executed. The transition is enabled when the invariants are satisfied.

We shall omit Sch from the transition relation whenever it is understood from the context.

Precedence constraints Assume that a set of precedence constraints in the form of $P_i \rightarrow P_j$ which define a partial order over the task set. To impose the precedence constraints on the executions of tasks, i.e. the delay transitions, we introduce a boolean variable $\mathsf{G_{i,j}}$ for every pair P_i, P_j such that $P_i \rightarrow P_j$, initialized with 0. $\mathsf{G_{i,j}} = 1$ means that P_i is completed and enables P_j. Thus when P_i is completed, $\mathsf{G_{i,j}}$ is switched to 1 and 0 again when P_j is completed. We shall use the predicate $\mathsf{ready(P)}$ to denote that P_i is ready to run according to the precedence constraints. Naturally $\mathsf{ready(P_i)}$ holds if $\mathsf{G_{k,i}} = 0$ for no P_k. Note that when $\mathsf{ready(P_i)}$ holds, there may be several instances of P_i, that are released. In that case, $\mathsf{ready(P_i)}$ means that the first instance of P_i is ready to run.

Now we formalize the semantics for task models with precedence constraints. To keep track on the booleans, we extend the variable valuation. The action rule remains the same as before. We have new rules for delay transition:

- $(l, \sigma, q) \xrightarrow{t}_{\mathsf{Sch}} (l, \sigma + t, \mathsf{Run(q, t)})$ if $(\sigma + t) \models I(l)$ and $t < c(\mathsf{Hd(ready(q))})$
- $(l, \sigma, q) \xrightarrow{t}_{\mathsf{Sch}} (l, \sigma' + t, \mathsf{Run(q, t)})$ if $(\sigma + t) \models I(l)$ and $t = c(\mathsf{Hd(ready(q))})$

where

- $\mathsf{ready(q)}$ is the sub-queue of q containing all elements P of q such that $\mathsf{ready(P)}$.
- σ' is the new valuation defined as follows: $\sigma'(\mathsf{G_{i,j}}) = 1$ if $P_i = \mathsf{Hd(ready(q))}$, $\sigma'(\mathsf{G_{i,j}}) = 0$ if $P_j = \mathsf{Hd(ready(q))}$ and $\sigma'(\mathsf{G_{i,j}}) = \sigma(\mathsf{G_{i,j}})$ otherwise.

In case the task queue is empty we interpret the conditions $t < c(\mathsf{Hd}(q))$ and $t = c(\mathsf{Hd}(q))$ as true. Hence the transition rules become equal and correspond to a delay transition in ordinary timed automata.

[2] Note that we fix Run to be the function that represents a one-processor system.

Resource constraints To resolve resource constraints and avoid unbounded priority inversion, various resource access protocols have been developed (see e.g. [7]). The protocols are based on fixed priorities for tasks, and the idea of priority ceiling; most of them are more or less simplified versions of the so called Priority Ceiling Protocol (PCP) [20].

In the following, we present a formal approach to describing the precise meaning of the resource access protocols. We shall use one of the simplified versions of PCP, the highest locker protocol (HLP) [19] to illustrate our approach. The protocol is deadlock-free, it prevents unbounded priority inversion, and it is widely used in practice due to its simple implementation.

To represent the priority ceilings of semaphores we use even numbers for the fixed priorities of tasks. Let 0 be the highest priority. Assume that for each task P, we have a semaphore access pattern which is a list $\{(\mathsf{T_i}, \mathsf{OP_i}, \mathsf{A_i})\}$, and the list is ordered according to $\mathsf{T_i}$'s.

Without losing generality, we assume that $\mathsf{OP_i}$ are operations on binary semaphores S_i with $S_i = 1$ meaning that it is free and locked otherwise. We shall use $\mathsf{sem(P)}$ to denote a set of semaphores currently locked by P.

From the access patterns, we can calculate the ceiling $\mathsf{C(s)}$ for each semaphore s, which is the highest priority of the tasks accessing s increased by 1. Thus, the ceilings are all odd numbers. Informally the HLP works as follows: whenever a task succeeds in locking a semaphore s, its priority $\mathsf{Pr}(P)$ should be replaced with $\mathsf{C(s)}$ if $\mathsf{Pr}(P) < \mathsf{C(s)}$.

Now we are ready to present the transition rules. Again the transition rule for actions remains the same. We extend the variable valuation to data variables and active priorities which will be changed according to HLP.

- $(l, \sigma, ((\mathsf{T}, \mathsf{OP}, \mathsf{A}) :: \mathsf{l}, \mathsf{d}) :: \mathsf{q}) \xrightarrow{\mathsf{t}}_{\mathsf{Sch}}$
  $(\mathsf{l}, \sigma, ((\mathsf{T} - \mathsf{t}, \mathsf{OP}, \mathsf{A}) :: \mathsf{l}, \mathsf{d} - \mathsf{t}) :: \mathsf{q})$ if $t \leq \mathsf{T}$
- $(l, \sigma, ((0, p(s), \mathsf{A}) :: \mathsf{l}, \mathsf{d}) :: \mathsf{q}) \xrightarrow{0}_{\mathsf{Sch}}$
  $(\mathsf{l}, \sigma', \mathsf{q})$ if $\sigma(s) = 1$
  where $\sigma' = \sigma[s := 0$,
  $\mathsf{sem(P)} := \mathsf{sem(P)} \cup \{\mathsf{s}\}, \mathsf{Pr(P)} := \mathsf{newPr(P, s)}]$
- $(l, \sigma, ((0, v(s), \mathsf{A}) :: \mathsf{l}, \mathsf{d}) :: \mathsf{q}) \xrightarrow{0}_{\mathsf{Sch}} (\mathsf{l}, \sigma', \mathsf{q})$
  where $\sigma' = \sigma[s := 1$,
  $\mathsf{sem(P)} := \mathsf{sem(P)} - \{\mathsf{s}\}, \mathsf{A}, \mathsf{Pr(P)} := \mathsf{oldPr(P)}]$

where

- $\mathsf{oldPr(P)}$ is the maximal priority of $\mathsf{C(s)}$ for all $s \in \mathsf{sem(P)}$.
- $\mathsf{newPr(P, s)} = \mathsf{C(s)}$ if $\sigma(\mathsf{Pr}(P)) < C(s)$ and $\sigma(\mathsf{Pr}(P)))$ otherwise.

4 Schedulability Analysis

Given a set of tasks and their arrival patterns described as a timed automaton, and a scheduling strategy, the related schedulability analysis problem is to check whether there exists a reachable state (l, u, q) of the automaton where the task queue q contains a task which misses its given deadline.

Definition 3. *(Schedulability) A state (l, u, q) is an error state denoted (l, u, Error) if there exists a task $P \in q$ such that $d(P) = 0$ and $c(P) > 0$. Naturally an automaton A with initial state (l_0, u_0, q_0) is non-schedulable with Sch iff $(l_0, u_0, q_0)(\rightarrow_{\mathsf{Sch}})^*(l, u, \mathsf{Error})$ for some l and u. Otherwise, we say that A is schedulable with Sch. More generally, we say that A is schedulable iff there exists a scheduling strategy Sch with which A is schedulable.*

For task sets with only timing constraints i.e. execution times and deadline, it is known that the problem is decidable [11]. In particular, for the optimal scheduling strategy, EDF (Earliest Deadline First), it is a reachability problem for timed automata extended with subtraction on clocks. From a recent work [10], we know that for fixed-priority scheduling strategies, the problem can be solved efficiently using ordinary timed automata with only two extra clocks.

In this section, we show that these results can be extended to deal with task sets imposed with the two other categories of task constraints i.e. precedence and resource constraints. Firstly we outline the general idea. For a given automaton with tasks A and a given scheduling strategy Sch, we construct two timed automata $E(\mathsf{Sch})$, and $E(A)$, and check for reachability of a predefined error state in the product automaton of the two. If the error state is reachable, automaton A is not schedulable with Sch. To construct the $E(A)$, the automaton A is annotated with distinct synchronization actions $\mathsf{release}_i$ on all edges leading to locations labeled with the task name P_i. The actions will allow the scheduler to observe when tasks are released for execution in A. The automaton $E(\mathsf{Sch})$ (the scheduler), encodes the task queue, the scheduling strategy Sch and the *task constraints*.

4.1 Timing Constraints

For task sets with execution times and deadlines as constraints, the structure of $E(\mathsf{Sch})$ is shown in Figure 1.

In the encoding, the task queue q is represented as a vector containing pairs of clocks (c_i, d_i) for every released task instance, called execution time and deadline clock respectively. In practice fewer clocks can be used for the encoding.

The intuitive interpretations of the locations in $E(\mathsf{Sch})$ are as follows:

- Idle - the task queue is empty,
- $\mathsf{Arrived(P_i)}$ - the task instance P_i has arrived,
- $\mathsf{Run(P_j)}$ - the task instance P_j is running,
- Finished - a task instance has finished,
- Error - the task queue is non-schedulable.

Locations $\mathsf{Arrived(P_i)}$ and Finished are marked as committed, which means that they are being left directly after entering.

We use the predicate $\mathsf{nonschedulable}(q)$ to denote the situation when the task queue becomes non-schedulable and naturally there is a transition labeled with the predicate leading to the error-state. The predicate is encoded as follows: $\exists P_i \in q$ such that $d_i > D_i$.

We use Sch in the encoding as a name holder for a scheduling strategy to sort the tasks queue. A given scheduling strategy is represented by the predicate: $P_i = \mathsf{Hd(Sch(q))}$. For example, Sch can be:

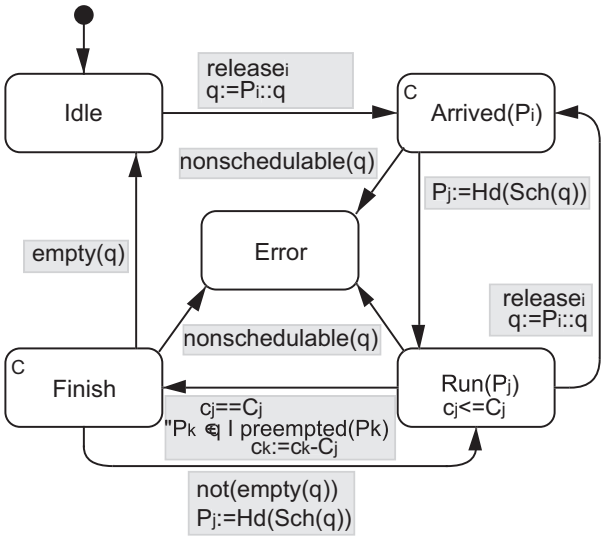

Fig. 1. Scheduler automaton for tasks with timing constraints.

- Highest priority first (FPS):
  $P_i \in q, \forall P_k \in q\ \mathsf{Pr}(P_i) \leq \mathsf{Pr}(P_k)$ where $\mathsf{Pr}(P)$ denotes the fixed priority of P.
- First come first served (FCFS):
  $P_i \in q, \forall P_k \in q\ d_i \geq d_k$
- Earliest deadline first (EDF):
  $P_i \in q, \forall P_k \in q\ D_i - d_i \leq D_k - d_k$
- Least laxity first (LLF):
  $P_i \in q, \forall P_k \in q\ c_i - d_i + D_i - C_i \leq c_k - d_k + D_k - C_k$

For more detailed description of the automaton $E(\mathsf{Sch})$, see [11]. Intuitively, whenever a new task P_i arrives, it is added to the task queue, denoted $q := P_i :: q$, and its deadline clock d_i is reset. Later, when it is chosen for execution by $\mathsf{Hd}(\mathsf{Sch}(\mathsf{q}))$, the automaton changes its state to $\mathsf{Run}(\mathsf{P_i})$, and the execution time clock c_i is reset. When the value of c_i reaches C_i in location $\mathsf{Run}(\mathsf{P_i})$, the task P_i finishes its execution and is being removed from the queue, the automaton changes location to $\mathsf{Finished}$, and either chooses a task with the highest priority from the queue to be executed, or takes the transition to the Idle state if the task queue is empty. On the transition from $\mathsf{Run}(\mathsf{P_i})$ to $\mathsf{Finished}$ execution time clocks of all preempted tasks are being reduced by C_i. It is necessary to keep the correct difference between C_k and c_k, i.e. time necessary to finish the execution. If P_i is preempted by a task P_j $E(\mathsf{Sch})$ changes its state to $\mathsf{Run}(\mathsf{P_j})$. We use the predicate $\mathsf{preempted}(P_i)$ to denote that P_i is preempted.

4.2 Precedence Constraints

Schedulability analysis for task sets with precedence constraints is similar to the case for timing constraints, as described in the previous subsection. The difference is that the task to be executed should be chosen not from the whole task queue but from a subset of tasks satisfying precedence constraints, that is the subset $\mathsf{ready(q)}$.

Recall from previous section, the boolean variable $\mathsf{G_{i,j}}$ denotes that an instance of task P_i is preceded by an instance of task P_j. Then the precedence constraints for each task P_k is simply the conjunction of booleans $\mathsf{G_{i,k}}$ for all edges $P_i \rightarrow P_k$ in the given precedence graph. Let $\mathsf{Ready(P_k)}$ denote the conjunctive formula. Then the union of $\mathsf{Ready(P_k)}$ for all P_k's represents the subset $\mathsf{ready(q)}$.

The construction of scheduler automaton taking precedence constraints into account is shown in Figure 2.

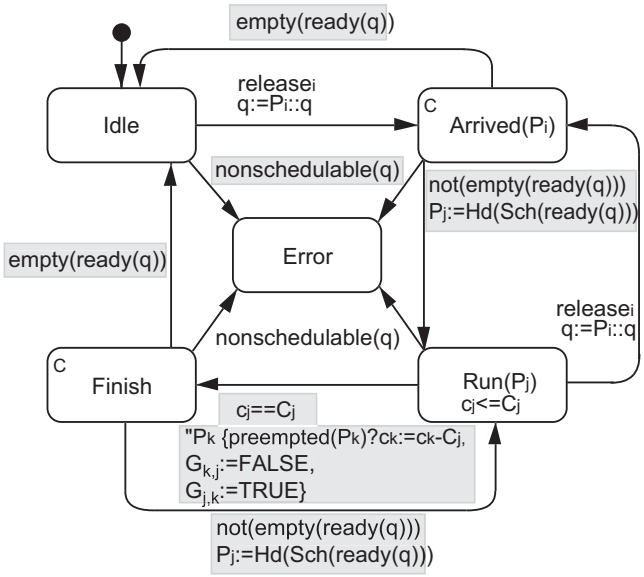

Fig. 2. Scheduler automaton handling precedence constraints. Changes from the encoding without precedence constraints are shown in the grey rectangles.

Modifications made in the scheduler described in the previous subsection to handle precedence constraints are the following transitions:

- $\mathsf{Idle} \rightarrow \mathsf{Error}$ is added since now idling of the processor does not imply an empty queue.
- $\mathsf{Arrived(P_i)} \rightarrow \mathsf{Idle}$ is taken when none of the tasks in the queue (including P_i) satisfies their precedence constraints.

– Finished→Idle is taken when a subset of the queue with satisfied precedence constraints becomes empty.
– Finished→Run(P_j) and Arrived(P_i)→Run(P_j) are taken when ready(q) is non-empty and then the highest priority task is chosen for execution from this subset.
– Run(P_j)→Finished modifies the booleans for precedence constraints. As before, execution time clocks of all currently preempted tasks are being subtracted by the given execution time (which is a constant) of the finished task, $C(P_j)$. All booleans $G_{j,k}$ are set to 1 (i.e. true), and $G_{k,j}$ set to 0. Intuitively it means that the task P_j has consumed all the inputs by the preceding tasks and are waiting again for its predecessors to present new inputs.

4.3 Resource Constraints

In this subsection we address schedulability analysis for task sets using semaphores to access shared resources. The main approaches to avoiding unbounded blocking due to semaphore access is to use resource access protocols to ”schedule” the semaphore locking operations issued by tasks. We shall simply consider the protocols as a scheduler. We shall only study the highest locker protocol (HLP) and it should be easy to see that the approach is applicable to the other protocols. The scheduler automaton describing HLP is shown in Figure 3.

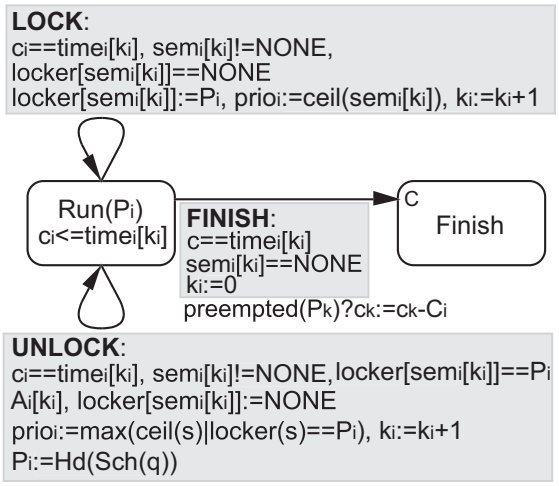

Fig. 3. Encoding of highest locker protocol.

The following variables and functions are used in the encoding:

– Function $ceil(s)$ returns a ceiling of semaphore s.

– Arrays $time_i$, sem_i, and A_i are used to keep a list of semaphore access patterns for each task type P_i. $time_i$ is a sorted array that contains time points when semaphore lock/unlock should occur, sem_i contains the corresponding semaphores, and A_i contains corresponding updates of shared variables. The last element of $time_i$ is the worst case execution time C_i, and the last element of sem_i is a constant NONE, which has a value -1, indicating that P_i has finished its execution. The arrays are static and are a part of the system description.
– Counter k_i is being used to go through all the elements of $time_i$, sem_i, and A_i while P_i is executing, for locking and unlocking the semaphores. The value of this counter is bounded to the length of arrays $time_i$ and sem_i. At any time during the execution of P_i a triple $(time_i[k_i], sem_i[k_i], A_i[k_i])$ represents the next time point, the semaphore to be locked or unlocked, and the update that has to be committed at the same time when semaphore is being unlocked.
– Integer $prio_i$ denotes active priority of task type P_i. Initially $prio_i$ is set to the given fixed priority of P_i, and during the execution of P_i its value can increase. According to the protocol, tasks can have intermediate priority levels, i.e. the number of active priority levels is a doubled number of original priority levels (number of task types). This number is a bound on $prio_i$.
– Array of integers $locker$ is used to keep track of the current status of the semaphores. At any time point the value of $locker[s]$ equals to index of a task that currently is using semaphore s. If s is not locked the value of $locker[s]$ is NONE. Length of the array equals to the number of semaphores in the system, and its elements are bounded with the number of task types in the system.

To extend schedulers from the previous sections with highest locker protocol, three types of transitions have to be added to the encodings, as shown in Figure 3. Initially the counter k_i is set to 0, and while the automaton is in the Run(P_i) state it will take LOCK or UNLOCK transition for each element of the array $time_i$ and increment the counter after each such transition. Invariant $c_i <= time_i[k_i]$ in the location Run(P_i) is used to force the automaton to take a transition as soon as its guard becomes true. Note, that the invariant will change dynamically with the incrementation of k_i.

– LOCK : when execution clock c_i of task P_i reaches time point $time_i[k_i]$, and the corresponding semaphore $sem_i[k_i]$ is unlocked, the task locks the semaphore by setting $locker[sem_i[k_i]]$ to i. The active priority $prio_i$ is set to the ceiling of the semaphore being locked, and the counter k_i is incremented.
– UNLOCK : when execution clock c_i of task P_i reaches time point $time_i[\ k_i]$, and the corresponding semaphore $sem_i[k_i]$ is locked, the task performs the update $A_i[k_i]$, and unlocks the semaphore by setting $locker[\ sem_i[\ k_i]]$ to NONE. The active priority $prio_i$ is set to priority that P_i had before locking the semaphore, which is the maximal ceiling of all semaphores currently locked by P_i, and the counter k_i is incremented. After unlocking of a semaphore the task priority can decrease, therefore at this time point a different task from the ready queue can be scheduled for execution, which is denoted by $i :=$ Hd(Sch(q)). Note that if the scheduler which takes precedence constraints into account is being extended then the highest priority task has to be chosen from the subset of tasks with satisfied precedence constraints, i.e. $i :=$ Hd(Sch(ready(q))).

– FINISH : this transition is an extended version of transitions used to denote the finish of task execution in the previous encodings. It is taken when k_i points to the last elements of $time_i$ and sem_i, i.e. C_i and NONE respectively. k_i is set to 0, and execution time clocks of all preempted tasks are decremented by C_i as in the previous encodings. If precedence constraints are being used, certain bits have to be updated as in encoding from the previous subsection.

Note, that in the highest locker protocol whenever a task P_i starts to execute, i.e. scheduler changes its state to Run(P_i) it implies that all resources that task will need during its execution are available, i.e. blocking never happens. In original priority ceiling protocol running task can be blocked by other task which means that the fourth type of transitions would have to be added to the encoding where $locker[sem_i[k_i]]! = NONE$ and $locker[sem_i[k_i]]! = i$.

5 Implementation

The results presented in this paper have been implemented in TIMES, a tool for modeling and schedulability analysis of embedded real-time systems [4]. The tool currently supports simulation, schedulability analysis, calculation of response times, checking of safety and liveness properties, and synthesis of executable C-code [5] for sets of tasks with arrival times specified by timed automata, under precedence and resource constraints. Graphical editor of the tool allows to specify timed automata extended with tasks, AND/OR precedence graphs, resource access patterns, as well as other task parameters. A simulator view of the TIMES tool analysing a simple system described in the example below is shown in Figure 4. To the left, lists of enabled transitions and variable valuations are displayed, and to the right, message sequence chart and Gantt chart for process execution are shown. A window with calculated worst-case response times is also presented.

Example 1. Consider the automaton in Figure 5. Tasks P and Q with worst-case execution times and deadlines shown in brackets are associated to locations $RelP$ and $RelQ$ respectively. Clock x and integer n are used to specify task release moments. Assume that preemptive deadline monotonic strategy is used to schedule the task queue. Then the automaton with initial state $(Idle, [x = 0, n = 0], [])$ may demonstrate the following sequence of transitions:

$$
\begin{aligned}
&(Idle, [x = 0, n = 0], []) \xrightarrow{0} \\
&(RelP, [x = 0, n = 0], [P(2, 8)]) \xrightarrow{1} \\
&(RelQ, [x = 1, n = 0], [P(1, 7), Q(2, 20)]) \xrightarrow{2} \\
&(Idle, [x = 3, n = 1], [Q(1, 18)]) \xrightarrow{0} \\
&(RelP, [x = 0, n = 1], [P(2, 8), Q(1, 18)]) \xrightarrow{1} \\
&(RelQ, [x = 1, n = 1], [P(1, 7), Q(1, 17), Q(2, 20)])
\end{aligned}
$$

Task schedule that corresponds to this sequence of transitions is shown in Figure 6(a). Note that at time 4 there are two instances of task Q in the queue.

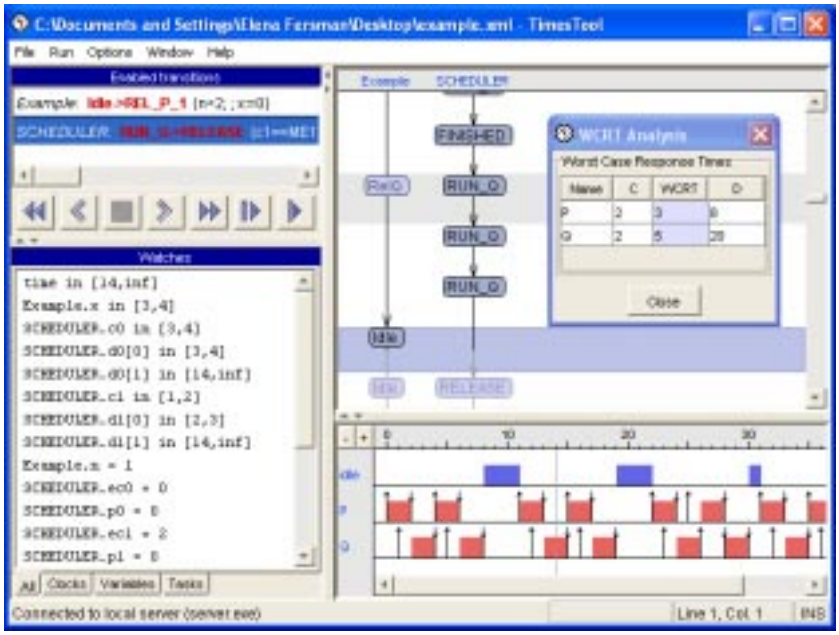

Fig. 4. A screen-shot of the TIMES tool.

When a precedence precedence graph shown in Figure 7 is applied, the task execution order is changed, which is illustrated in Figure 6(b). Each instance of task P has to wait until an instance of Q is completed, and therefore in the beginning of execution, P is not allowed to run.

Now assume that tasks P and Q have two shared resources, protected by semaphores s_1 and s_2. Task P locks semaphore s_1 for the first half of its execution and s_2 for the second half. Task Q locks both semaphores for the whole its execution. Task schedule that corresponds to this sequence of transitions is shown in Figure 6(c). Note that at time 3, both semaphores are locked by Q, and therefore P is being blocked.

Worst-case response times for tasks P and Q, calculated by TIMES tool, are summarised in Table 1. For all these cases the tasks are schedulable because they complete before their deadlines.

Type of constraint	WCRT(P)	WCRT(Q)
Timing only	2	5
Precedence	6	3
Resource	3	5
Precedence and resource	6	3

Table 1. WCRT for tasks with different constraints.

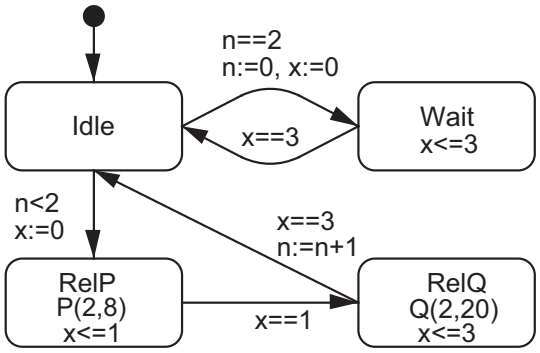

Fig. 5. Timed automaton with tasks P and Q.

6 Conclusions and Related Work

In scheduling theory [7], off-line schedulability tests for real time systems, usually assume that real time tasks are periodic (or sporadic with known minimal inter-arrival times), i.e. they will be released with fixed rates. The assumption of complete knowledge on task uniform arrival times may cause unacceptable resource requirements due to the pessimistic analysis. We have proposed to use timed automata to describe task arrival patterns. In a recent work [10], it is shown that for fixed priority scheduling strategy and tasks with only timing constraints (i.e. execution time and deadline), the schedulability of such models can be checked by reachability analysis on timed automata with *two* clocks.

In this paper, we have extended the above result to deal with tasks that have not only timing constraints but also precedence and resource constraints. We have presented a unified model for finite control structures, concurrency, synchronization, and tasks with combinations of timing, precedence and resource constraints. We have shown that the schedulability analysis problem for the extended model can be solved efficiently using the same technique as in [10].

The presented results have been implemented in the TIMES tool for automated schedulability analysis. We believe that timed automata and our contributions provide a bridge between scheduling theory and automata-theoretic approaches to system modeling and verification for real time systems. Indeed, the analysis can be done in a similar manner as response time analysis in classic Rate-Monotonic Analysis, and also other techniques such as model checking for logical correctness verification can be applied within the same framework.

Related work. Scheduling is a well-established area. Various analysis methods have been published in the literature. For systems restricted to periodic tasks, algorithms such as rate monotonic scheduling are widely used and efficient methods for schedulability checking exist, see e.g. [7]. In the past years, these classic works have been extended to deal with more complex constraints e.g. offset analysis [18] and unfolding [6] for prece-

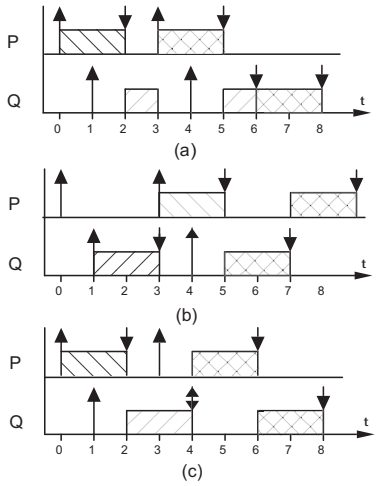

Fig. 6. Gantt charts for tasks with (a) only timing, (b) precedence $Q \rightarrow P$, (c) resource constraints.

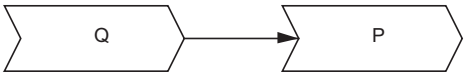

Fig. 7. Precedence graph.

dence constraints and priority ceiling protocols [20, 19] for shared resources. However these works are all within the framework of periodic tasks. Our work is more related to work on timed systems and scheduling. Timed automata has been used to solve non-preemptive scheduling problems mainly for job-shop scheduling[1, 9, 14]. These techniques specify pre-defined locations of an automaton as goals to achieve by scheduling and use reachability analysis to construct traces leading to the goal locations. The traces are used as schedules. A work on relating classic scheduling theory to timed systems is the controller synthesis approach [2, 3]. The idea is to achieve schedulability by construction. A general framework to characterize scheduling constraints as invariants and synthesize scheduled systems by decomposition of constraints is presented in [3]. However, algorithmic aspects are not discussed in this work.

Acknowledgment: We would like to thank the members of the UPPAAL group at Uppsala University for discussions.

References

1. Y. Abdeddam and O. Maler. Job-shop scheduling using timed automata. In *Proc. of CAV'01*, 2001.

2. K. Altisen, G. Gößler, A. Pnueli, J. Sifakis, S. Tripakis, and S. Yovine. A framework for scheduler synthesis. In *Proc. of IEEE RTSS'99*, pages 154–163. IEEE Computer Society Press, 1999.
3. K. Altisen, G. Gößler, and J. Sifakis. A methodology for the construction of scheduled systems. In *Proc. of FTRTFT'2000, LNCS 1926, pp.106-120*, 2000.
4. T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Times - a tool for modelling and implementation of embedded systems. In *Proc. of TACAS'02*, volume 2280 of *LNCS*, pages 460–464. Springer, 2002.
5. T. Amnell, E. Fersman, P. Pettersson, H. Sun, and W. Yi. Code synthesis for timed automata. *Nordic Journal of Computing*, 9(4):269–300, 2002.
6. F. Balarin, L. Lavagno, P. Murthy, and A. Sangiovanni-vincentelli. Scheduling for embedded real-time systems. *IEEE Design & Test of Computers*, 15(1):71–82, 1998.
7. G. C. Buttazzo. *Hard Real-Time Computing Systems. Predictable Scheduling Algorithms and Applications*. Kulwer Academic Publishers, 1997.
8. C. Ericsson, A. Wall, and W. Yi. Timed automata as task models for event-driven systems. In *Proceedings of Nordic Workshop on Programming Theory*, 1998.
9. A. Fehnker. Scheduling a steel plant with timed automata. In *Proc. of RTCSA'99*. IEEE Computer Society Press, 1999.
10. E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Schedulability analysis using two clocks. In *Proc. of TACAS'03*, volume 2619 of *LNCS*, pages 224–239. Springer, 2003.
11. E. Fersman, P. Pettersson, and W. Yi. Timed automata with asynchronous processes: Schedulability and decidability. In *Proc. of TACAS'02*, volume 2280 of *LNCS*, pages 67–82. Springer, 2002.
12. D. W. Gillies and J. W.-S. Liu. Scheduling tasks with and/or precedence constraints. *SIAM Journal on Computing*, 24(4):797–810, 1995.
13. H. Hüttel and K. G. Larsen. The use of static constructs in a modal process logic. *Logic at Botik*, 363:163–180, 1989.
14. T. Hune, K. G. Larsen, and P. Pettersson. Guided Synthesis of Control Programs using UPPAAL. *Nordic Journal of Computing*, 8(1):43–64, 2001.
15. M. Joseph and P. Pandya. Finding response times in a real-time system. *BSC Computer Journal*, 29(5):390–395, October 1986.
16. K. G. Larsen, P. Pettersson, and W. Yi. Compositional and symbolic model-checking of real-time systems. In *IEEE Real-Time Systems Symposium*, pages 76–89, 1995.
17. C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. In *Journal of the ACM*, volume 20(1), pages 46–61, 1973.
18. J.C. Palencia and M. Gonzlez Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proc. of RTSS'98*, pages 26–38. IEEE Computer Society Press, 1998.
19. R. Rajkumar, L. Sha, and J.P. Lehoczky. An experimental investigation of synchronisation protocols. In *Proceedings 6th IEEE Workshop on Real-Time Operating Systems and Software*, pages 11–17. IEEE Computer Society Press, 1998.
20. L. Sha, R. Rajkumar, and J. Lehozky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. on Computers*, 39(9):1175–1185, 1990.
21. K. Tindell. *Fixed Priority Scheduling for Hard Real-Time Systems*. Ph.D. Thesis, Department of Computer Science, University of York, 1993.