Electronic Notes in Theoretical Computer Science 42 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume42.html 15 pages

Analysis of Scheduling Behaviour using Generic Timed Automata

Thorsten Gerdsmeier[1,2] and Rachel Cardell-Oliver[3]

Department of Computer Science
University of Essex
Colchester CO4 3SQ, United Kingdom

Abstract

A method for dynamic, automated analysis of the behaviour of real-time programs under different scheduling algorithms is presented. Each scheduling algorithm is defined by a generic timed automaton which can be instantiated with data for a particular set of tasks. The resulting network of instantiated timed automata can be analysed automatically in the model-checker Uppaal, to ensure correctness properties are satisfied. Various scheduling metrics can also be calculated. We assume that data such as worst case execution times, periods, deadlines and priorities have been pre-calculated for each task. In this paper we present generic specifications for uniprocessor scheduling using the immediate ceiling priority protocol (ICPP) and the earliest-deadline-first (EDF) algorithm and analyse the behaviour of a mine pump controller implemented in Ada95 under each scheduling algorithm.

1 Introduction

To analyse programs, rather than specifications, we must consider how they will be scheduled for execution in an implementation environment. Since the 1970s the problem of analysing possible schedules for a given set of tasks under a particular scheduling strategy has been an active research topic. Rate monotonic analysis [12] and its extensions provide conservative analysis of the schedulability of a particular set of periodic tasks. Such predictions are safe (if RMA predicts tasks are schedulable, then they will be schedulable in an implementation), but may be pessimistic (RMA may predict tasks can not be scheduled, when the implementation is in fact schedulable).

A major advantage of the method of RMA, substituting real parameters into

[1] First author is supported by awards from the University of Essex.

[2] Email: gerdst@essex.ac.uk

[3] Email: cardr@essex.ac.uk

formulae, is that it is relatively cheap to apply. A system can also be analysed by implementing it and measuring task execution times. The implementation method is expensive to perform because a system must be implemented before it can be analysed and there is always an uncertainty whether the worst case execution time has been observed.

More recently, several authors have considered automatic, dynamic analysis by exploring all possible behaviours of a set of tasks in interaction when they are executing. This approach promises safe estimates, which are more accurate than rate monotonic analysis but cheaper to perform than implementation and analysis. Dynamic methods for scheduling analysis have been proposed by several authors [2,3,7,6] using timed automata [1] to model tasks and model-checking tools such as Uppaal and HyTech to perform the analysis. The first of these was developed by Corbett [6]. Here a program automaton is explicitly constructed for a multi-tasking program. The automaton is then analysed with the model checker HyTech. The case studies show that the number of locations of the program automaton becomes quite large. Bradley et al.[3] show how to construct a single timed automaton which models the interactions of a mixture of periodic tasks and tasks with message dependencies, executed on multiple processors. Ericsson et al. [7] consider scheduling of tasks with a fixed execution time and deadline which are triggered by events on a single processor. In Braberman and Felder [2] processes are modelled by constructing an automaton for each task. Delays are added to model the interference of tasks.

Each of these dynamic analysis methods share the disadvantage that generating the timed automata model of scheduled tasks is complex, and thus time consuming and prone to error. We have attempted to reduce that cost by introducing *generic* automata to represent the behaviour of a typical task under a particular scheduling strategy. Generic timed automata are timed automata templates that can be instantiated with parameters. Program statements are abstracted to time consuming actions represented by time spent in automaton locations. Scheduling strategies are defined by guards and synchronisations which govern the transitions of the automaton. For each task of the program to be analysed, the timing of program statements, priorities, periods and deadlines are passed as parameters to the generic task automaton. Properties of the scheduled program can then be analysed automatically using a model checker to explore all possible behaviours of its set of tasks.

The main contribution of this paper is to show how different scheduling strategies can conveniently be defined by generic timed automata and that these generic automata can be used to analyse properties of a set of tasks under different scheduling strategies. We focus on the activity of analysing scheduled behaviour of a set of tasks. Our method for dynamic analysis of schedules uses timed automata and the model checker, Uppaal. We assume that code execution times, task periods, deadlines and the timing of any triggering events are pre-calculated.

Section 2 introduces Uppaal timed automata and scheduling strategies. In Section 3, two contrasting scheduling algorithms, *Immediate Ceiling Priority Protocol* (ICPP) (without shared variables) and *Earliest Deadline First* (EDF) (preemptive), are each defined using generic timed automata. We prove that the generic models are correct. For example, that mutual exclusion is ensured (only one process may be running at a time) and priorities respected (no process should start executing whilst there is a higher priority process ready or pre-empted). The ICPP scheduling is simplified in this paper for space reasons. We do not model task communication, but in [8] we demonstrate how to analyse a set of task that are communicating via protected objects. Task synchronization is not considered because we take the Ravenscar profile [4] as guide. Section 4 shows how the automata are instantiated for the analysis of a particular program : Burns and Welling's Ada95 mine pump program (without shared resources). Section 5 shows how the resulting UPPAAL specification can be used to analyse performance properties. In section 6 we discuss our work and explain how shared resources can also be handled in the model.

2 Scheduling and Tasks in Timed Automata

From the point of view of scheduling analysis, a task can be in one of three possible states: ready to run, running or waiting. Initially each task will be in its ready state. Under certain conditions it can change from ready to running, then either from running to ready again if pre-empted or from running to waiting once execution has completed. The task returns from its waiting state to being ready to execute when triggered by a period or event. This is the most general form of the model. Some states may be irrelevant for certain strategies. For example, under round robin scheduling and also in earliest deadline first there is no wait state and tasks alternate between running and ready states only.

A *scheduling algorithm* is defined by the rules which control when and how transitions between states occur. In order to model tasks for scheduling, therefore, we need a specification language with states and transitions governed by time and data constraints. The language of *timed automata* [1] exactly meets these requirements.

There are many versions of timed automata. We use the UPPAAL timed automaton model [11]. The UPPAAL version is a finite state automaton equipped with real valued clocks, integer valued data variables and synchronisation actions. Transitions have guards and reset operations on clock and data variables. At any time an automaton can change its location by following a transition, provided the current clock and data variables satisfy the enabling guard. Transitions occur instantaneously and time only passes in locations. The values of all clocks increase synchronously with time. An invariant is associated with every location which determines valid clock values for the location. Automata synchronise with each other via channels and shared variables. We do not have inegrators in UPPAAL, instead we use simple integer variables. UPPAAL uses a dense time model but for the work reported in this paper a discrete time model checker would also do.

The rules of Uppaal timed automata which govern change of location have three different components. Any of the three rule components may be empty.

guard the constraints on clock values and the values of global variables under which the transition is taken. For example, $myclk == 1, w \leq mypri * 2 - 1$ states the value of my clock is 1 and that the value of global variable w is at most $mypri * 2 - 1$.

synchronisation causes two different tasks to synchronise over a named channel. For example, synchronisation channels are used to ensure that the actions such as

(i) a pre-empted task changing from running to ready state and
(ii) a task with a higher priority task changing from its ready to running state

occur simultaneously.

assignments reset clock variables and update global variables. For example $myclk := 0, w := w + mypri$ describes the addition of a task to the set of waiting tasks and sets the task's clock to 0 ready to measure the timing of the next action.

In recent versions of Uppaal [10], an automata template can be defined using parameters for any of its constants, clocks, data variables and synchronisation channels. A network of automata can then be constructed by instantiating parameterised automata with actual parameters. Process parameters can include arrays of data variables [10].

3 Specifying Scheduling Strategies

We explain our method by introducing generic models for two contrasting scheduling strategies: ICPP and EDF. In both cases we assume a single processor is used. In ICPP scheduling we assume tasks with a fixed period. Task overheads such as context switches, entry and exit times of protected operations, entry and exit times for processing of delay statements, timer interrupts and rescheduling times are included in the execution times. Jitter on clocks is not considered.

3.1 Immediate Ceiling Priority Protocol

For an introduction to ICPP scheduling the reader is referred to [5,9]. In order to schedule tasks under ICPP we need the following data.

For each task:

- *priority of the task,* for priority levels $0 \dots n$ we define priority of a task on priority level i by $mypri = 2^i$. The idle task has priority 0.

- *period of the task*, the time elapsed between each invocation of the task.
- *worst case execution time* for the functional code of each task, given as a certain number of atomic actions.

For the full system:

- *interval between system clock interrupts:* we assume the ideal clock interrupt model in which clock interrupts can occur at any time, not just at certain periodic times. A task which is in state ready can start immediately if no task of higher priority is running.
- *time slice for atomic actions*, for simplicity we shall always assume a granularity of 1 time unit in this paper.

In addition, each task is assigned a local clock, a clock to measure its period, and a counter which keeps track of how many atomic actions have been performed or equivalently how many time slices of the functional code have been executed.

The definiton of a typical task `TaskICPP` under ICPP scheduling can be found in figure 1. The automata `TaskICPP` is in location `ready` if the corresponding task is ready to run, it is in location `run` if the task is executing an atomic action, and in location `wait` if the task is temporarily inactive between completion of execution and its next invocation. The location `runwait` is introduced for technical reasons and corresponds to no state of the real program. The global integer variable `w` is always equal to the sum of all priorities of tasks that are not in location `wait`. A task can change its location from `ready` to `run`, if no task of higher priority is in location `run` or `ready`. This is checked by the guard `w<=2*mypri-1`. If the guard is satisfied the task can start executing. We assume a single processor architecture where only one task can run at a time. Therefore for one task to start executing another task must be pre-empted or finish executing. This is realised by the synchronisation action `change` which causes two transitions to be taken simultaneously: one from each participating task.

Exactly one task is always executing because we include an idle task as well as the program tasks. If a task in location `run` has not finished all `mymax` of its atomic actions it can either re-enter `run` if all tasks of higher priority are inactive or enters the location `ready` if a task of higher priority is active. From location `ready` the task can return to `run` once all tasks of higher priority have become inactive. After a task completes execution of all its `mymax` atomic actions, it makes itself inactive by entering location `wait` until its period triggers the task again and the task automaton returns to its `ready` state to await being scheduled to run again.

Verification of the scheduling strategy

We can now check that the generic automata we have defined are correct in that they ensure the characteristics of ICPP scheduling on a single processor.

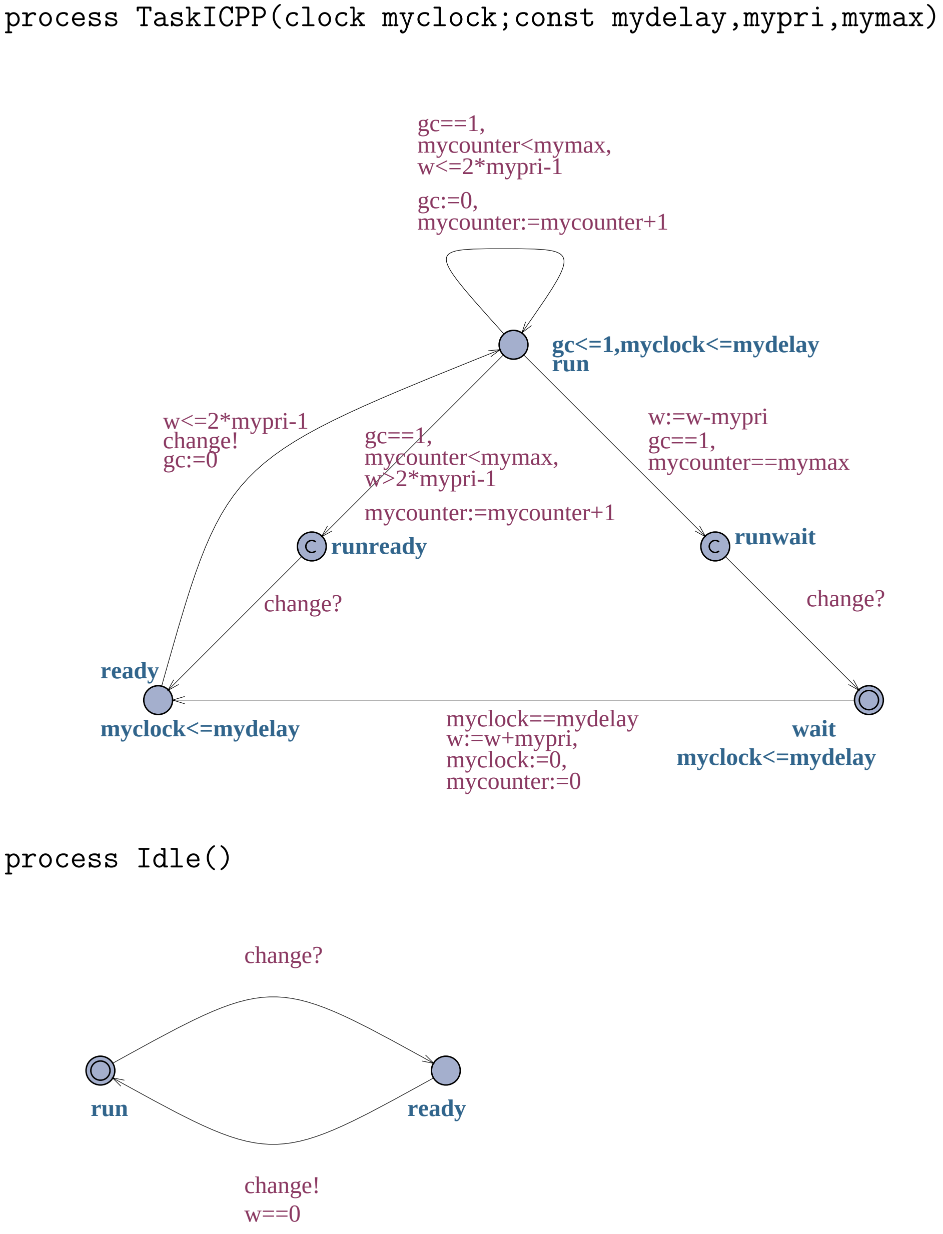

Fig. 1. Generic Automata for ICPP Scheduling

Property 1: Exactly one task is running at any time.
Proof: Let r be the number of running tasks.
a) In the initial state only the idle task is running, so $r = 1$.
b) Whenever a task starts running another task has to stop running. When a task starts running it has to take the transition `ready -> run`. These transitions are synchronized by `change!`. Therefore another task has to take a transition synchronized with `change?`. This is only possible on the transitions `runready -> ready` which immediately follows `run -> runready` and `runwait -> wait` which immediately follows `run -> runwait`. It follows if one task is entering state `run` from state `ready`, another task has to stop run-

ning, so $r := r + 1 - 1$. For all other transitions r is unchanged.
Therefore we have always $r = 1$ as required.
End Proof

Property 2: No task can enter the `run` state if any task with higher priority is in location `ready`.
Proof:
We assume task k with priority 2^k is the task with highest priority in location `ready`. Suppose that task j with lower priority ($j < k$) enters location `run`. Task j can enter run only on transitions `ready -> run` and `run -> run`. These transitions are enabled only if $w \leq 2 * 2^j - 1$. 2^k is in `w`'s sum whenever task k is in ready or run states since it enters that part of the graph with `w:=w+mypri`, never changes `w` and leaves it with `w:=w-mypri`.
Therefore if task k is in location ready : $w > 2 * 2^j - 1$,for all $j < k$ because : $w \geq 2^k >= 2^{j+1} > 2^{j+1} - 1 = 2 * 2^j - 1$. Therefore no task j can enter `run` if a higher priority task `k` is in location `ready`.
End Proof

Property 3: For n+1 tasks $0..n$ it is always true: $w \leq 2^{n+1} - 1$.
Proof:
Induction base case: n=0
One idle task and one TaskICPP process on task level 0. We only increase `w` on the transition `wait -> ready`. From ready there is no trace to wait without taking the transion `run -> runwait` where we decrease `w` by the same value. It follows $w \leq 2^{0+1} - 1 = 1$.
Induction step case: `n -> n+1`
We add one task with task on priority level (n+1). The tasks on level $0 \ldots n$ can never change the value of `w` to a value bigger than $2^{n+1} - 1$ as we assume. For the task on level n+1 we only increase `w` on the transition `wait -> ready`. On this transition we increase `w` by 2^{n+1}. From ready there is no trace to wait without taking the transion `run->runwait` and decreasing `w` by 2^{n+1}. It follows the process can never increase `w` by more than 2^{n+1}.
We conclude : $w <= 2^{n+1} - 1 + 2^{n+1} = 2^{n+2} - 1$.
End Proof

Property 4: The task with the highest priority is never pre-empted.
Proof:
The task with highest priority, say 2^n can only be preempted by taking the transition `run -> runready` followed by `runready -> ready` . This transition can only be taken if $w > 2 * 2^n - 1$. But this is not possible because from **Property 3** always $w \leq 2^{n+1} - 1$.
End Proof

3.2 Earliest Deadline First

In order to schedule tasks under EDF we need the following information about each task:

- *time slice used for atomic actions:* time slice is 1 in the examples given in this paper,
- *worst case and best case execution times* for the functional code of each task, given as a multiple of the time slice,
- *deadline to complete each task*

The generic automata for earliest deadline scheduling is shown in Figure 2. The task automaton has only two locations, `run` and `ready` and these locations have the same meaning as in the ICPP automata. We assume every task has a constant deadline *mydead*. The aim of earliest deadline scheduling is to ensure that no task ever exceeds it's deadline. Thus, for a task with deadline `maxwait` and maximum execution time `execmax`, `myclock` should never exceed `maxwait` in location `ready`.

In fact, we add that condition as invariant on the ready state. Should a particular instantiation not be schedulable, the model-checker Uppaal will alert the user that a deadlock arises. In this case, new deadlines must be chosen until the set of tasks is schedulable. Adding the invariant is safe because it does not introduce any new behaviours except a possible deadlock.

After every time slice (that is when clock `gc` is one) it is checked again what task is running next. The task selected must be that with the nearest deadline: that is k with deadline $deadk$ and clock ck so that $deadk - ck$ is minimal over all user tasks 1 to n. Since the language of data expressions in Uppaal does not allow the direct calculation of this function, we introduce a supporting automata `Calculate` which is called by a task automata whenever the earliest deadline calculation is required. If `mycounter` is in the range of `execmin` and `execmac`, `mycounter` and `myclock` is reset and location `ready` is entered. This corresponds to finishing execution of the task. The range of possible values of `mycounter` when this transition can be taken introduces some non-determinism.
If `mycounter` is lower than `execmac`, `mycounter` is increased by one and location `ready` is entered. A transition from location `ready` to location `run` is possible if the task has the earliest deadline, that is `earliest==myid`.

Verification of the scheduling strategy

We check two properties of EDF on a single processor.

Property 1: Never more than one user task is running.
Proof:
A user task is running when it is in location `run`. Initially all tasks are in location `ready` (*r=0*), `S0-go!-S2` increases *r:=r+1* leaves *r=1* then `S2-calc?-S1`

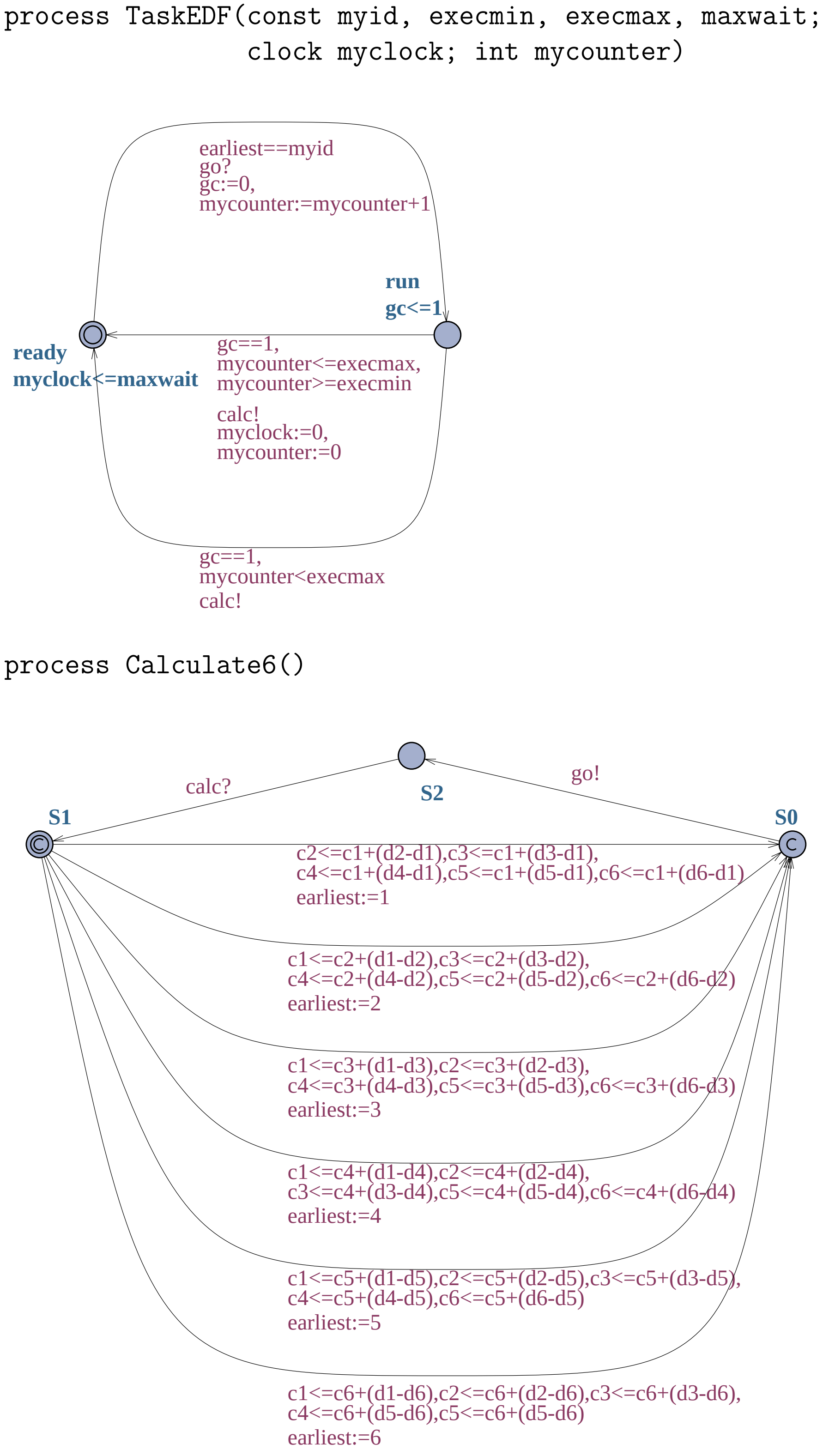
process TaskEDF(const myid, execmin, execmax, maxwait;
clock myclock; int mycounter)
earliest==myid
go?
gc:=0,
mycounter:=mycounter+1
run
gc<=1
ready
myclock<=maxwait
gc==1,
mycounter<=execmax,
mycounter>=execmin
calc!
myclock:=0,
mycounter:=0
gc==1,
mycounter<execmax
calc!
process Calculate6()
calc?
S2
go!
S1
S0
c2<=c1+(d2-d1),c3<=c1+(d3-d1),
c4<=c1+(d4-d1),c5<=c1+(d5-d1),c6<=c1+(d6-d1)
earliest:=1
c1<=c2+(d1-d2),c3<=c2+(d3-d2),
c4<=c2+(d4-d2),c5<=c2+(d5-d2),c6<=c2+(d6-d2)
earliest:=2
c1<=c3+(d1-d3),c2<=c3+(d2-d3),
c4<=c3+(d4-d3),c5<=c3+(d5-d3),c6<=c3+(d6-d3)
earliest:=3
c1<=c4+(d1-d4),c2<=c4+(d2-d4),
c3<=c4+(d3-d4),c5<=c4+(d5-d4),c6<=c4+(d6-d4)
earliest:=4
c1<=c5+(d1-d5),c2<=c5+(d2-d5),c3<=c5+(d3-d5),
c4<=c5+(d4-d5),c6<=c5+(d6-d5)
earliest:=5
c1<=c6+(d1-d6),c2<=c6+(d2-d6),c3<=c6+(d3-d6),
c4<=c6+(d5-d6),c5<=c6+(d5-d6)
earliest:=6

Fig. 2. Generic Automata for EDF Scheduling of six tasks

decreases *r:=r-1* leaves *r=0* before any other task can become ready.
End Proof

Property 2: The task that starts executing has the earliest deadline.
Proof:
The variable `earliest` is equal to i, if task i has the earliest deadline since `earliest` $:= i \Leftrightarrow i = min_j(d_j - c_j)$. (We call `myclock` of task j c_j). A user task can only start running, if `earliest` is equal to its task identity.
End Proof

4 Scheduling an Ada95 Mine Pump Program

We can now analyse the behaviour of a given implementation for a particular scheduling strategy. We illustrate our method using a mine-pump program of Burns and Wellings implemented in Ada 95 [9][Chapter 3].

4.1 Mine Pump with ICPP Scheduling

The mine pump program [9] consists of six user tasks: `Methane_Monitor`, `Air_Monitor`, `Co_Monitor` and `Safety_Checker` are periodic tasks, `High_Sensor` and `Low_Sensor` are sporadic tasks. As is standard in rate monotonic analysis, the two sporadic processes, `High_Sensor` and `Low_Sensor` are represented in [9] by a periodic process. The task data required to analyse this program, in time units of 100 microseconds, has been pre-calculated in [9] reproduced in Table 1. The period of the sporadic tasks `Low_Sensor` and `High_Sensor` has been reduced from 10000 in the original to 1000 here. This is pessimistic but safe and reduces the computing time for model checking. Clock interrupts occur every 50 time units. However, in the following we assume ideal clock interrupts: processes can be interrupted or begun at any integer time. The WCET of every task is *C1*. The WCET plus context switches (that are made by the environment) is *C2* and it is this time we use as task execution time for our scheduling analysis. Each task's execution time includes two context switches. *C1* and *C2* include time spent in protected objects. The *Deadline* is the time in which the computation of a task has to be finshed.

In Uppaal, the full specification is given by the generic automata shown in Figure 1 together with the following declarations.

```
clock c1,c2,c3,c4,c5,c6,gc;
int  [0,63] w:=0;
urgent chan change;

Methane_Monitor :=TaskICPP(c1, 200, 32, 58);
Air_Monitor :=TaskICPP(c2, 300, 16, 37);
```

Task Name	Priority	Period	C1	C2	Deadline
Methane Monitor	32	200	54	58	100
Air Monitor	16	300	33	37	200
CO Monitor	8	300	33	37	200
Safety Checker	4	350	35	39	300
Low Sensor	2	1000	29	33	750
High Sensor	1	1000	29	33	1000

Table 1
ICPP scheduling data for Ada95 Mine Pump program

```
CO_Monitor :=TaskICPP(c3, 300,  8, 37);
Safety_Checker :=TaskICPP(c4, 350,  4, 39);
Low_Sensor :=TaskICPP(c5, 1000, 2, 33);
High_Sensor :=TaskICPP(c6, 1000, 1, 33);

system  Methane_Monitor, Air_Monitor, CO_Monitor,
Safety_Checker, High_Sensor, Low_Sensor, Idle;
```

4.2 Mine Pump with EDF Scheduling

The task parameters relevant for EDF scheduling, are shown in Table 2. Since we had no data for best case execution times, we have used WCET for both. We have selected EDF deadlines published in [9].

Task Name	TaskID	Execution	Deadline
Methane Monitor	1	58	200
Air Monitor	2	37	250
CO Monitor	3	37	300
Safety Checker	4	39	350
Low Sensor	5	33	800
High Sensor	6	33	1000

Table 2
EDF scheduling data for Ada95 Mine Pump program

In Uppaal, the full specification is given by the automata of Figure 2 together with the system information below. As before, we assume ideal clock interrupts that can happen at any time.

```
const d1 200;
const d2 250;
const d3 300;
const d4 350;
const d5 800;
const d6 1000;

clock c1,c2,c3,c4,c5,c6,gc;
int earliest,
    mycounter1,mycounter2,mycounter3,mycounter4,mycounter5,mycounter5;
chan calc,go;

Methane_Monitor :=TaskEDF(1, 58, 58, d1, c1 );
Air_Monitor :=TaskEDF(2, 37, 37, d2, c2);
Co_Monitor :=TaskEDF(3, 37, 37, d3, c3);
Safety_Checker :=TaskEDF(4, 39, 39, d4, c4);
HighSensor :=TaskEDF(5, 33, 33, d5, c5);
LowSensor :=TaskEDF(6, 33, 33, d6, c6);

system  Methane_Monitor, Air_Monitor, Co_Monitor,
Safety_Checker, HighSensor, LowSensor, Calculate6;
```

5 Analysis of Scheduling

In section 3 we proved the correctness of our generic specifications of ICPP and EDF scheduling using inductive reasoning. In this section we demonstrate how model checking can be used to analyse the performance of a given set of tasks under a particular scheduling strategy.

A set of tasks is schedulable unless the deadline constraints are violated. These constraints for schedulability are stated as state invariants of generic specifications. Deadline violation is flagged as a deadlock by the Uppaal model-checker and if no deadlocks are flagged, then the set of tasks is schedulable. The mine pump program with parameters of Tables 1 and 2 have been checked in Uppaal to be schedulable under ICPP and EDF respectively.

We can use task clocks to derive certain performance bounds. A task's response time is the time taken from when the task is first ready until it completes execution. The worst case response time of ICPP scheduling can be found by selecting a `R` for every `TaskX` so that:

```
A[] (TaskX.runwait) imply myclock <= R-1 fails
A[] (TaskX.runwait) imply myclock <= R succeeds
```

The best case reponse time of ICPP scheduling can be found similarly by choosing an `S` so that

```
A[] TaskX.runwait imply myclock >= S+1 fails
A[] TaskX.runwait imply myclock >= S succeeds
```

The maximum waiting times of tasks scheduled under EDF can be found similarily:

```
A[] (TaskX.ready imply myclock <= W-1) fails
A[] (TaskX.ready imply myclock <= W) succeeds
```

Results of response time analysis for the mine pump under ICPP and EDF are given in Tables 3 and 4. It can be seen that all tasks meet their dealines. In the mine pump example the knowledge about response times gives us information about the age of measured sensor data.

Task Name	Best Case	Worst Case	Task Response
	Response	Response	Deadline
Methane Monitor	58	59	100
Air Monitor	37	96	200
CO Monitor	74	133	200
Safety Checker	39	172	300
Low Sensor	91	263	750
High Sensor	124	295	1000

Table 3
ICPP response times

Task Name	Maximum Wait	Task
	Observed	Deadline
Methane Monitor	201	200
Air Monitor	203	250
CO Monitor	298	300
Safety Checker	335	350
Low Sensor	769	800
High Sensor	959	1000

Table 4
EDF response times

6 Discussion

We have introduced generic timed automata for analysing ICPP and EDF scheduling strategies. The automata can be instantiated with parameters such as WCET that characterize the program under analysis. We have demonstrated the technique by analysing an Ada95 mine pump program using published deadlines and WCETs. We have not used shared resources.

In [8] we assume a scheduler based on the Ravenscar [4] where task communication is possible via protected objects and not via rendevous : the model introduced in this paper is extended with protected objects. We suggest every protected object is modelled by an automaton. A task calls a protected object by synchronization with the corresponding automaton. The synchronization takes part when the value of `mycounter` of an automaton `TaskEDF` or `TaskICPP` has certain values (the time when the protected object is called).

Our model is mostly deterministic, so that the state space does not explode quickly. We used a Pentium III with 800 Mhz and 768 MByte RAM for analysis. The requirements of Table 3 and 4 have been checked in less than a minute each. The model in [8] checked for the same requirements takes also less than a minute each. If we change the periods of both `LowSensor` and `HighSensor` from 1000 to 10000, it takes around one hour to check each requirement.

We can, however, use several techniques to reduce the state space and thus reduce the model checking time. In our example we assumed an ideal clock checked every millisecond. This is ideal but not realistic. In realistic systems one could expect a check each 10 milliseconds. This coarser time grain could be modelled by dividing all times in our example by 10. Second, we could combine our model with scheduling theory by calculating some response times of tasks with functional analysis and integrate these results in our model.

A third way to reduce state space is to use harmonic values (periods with common factors) as delay times of tasks as done by Corbett and Bradley et al. [6,3].

In this paper we analysed task response times for the mine pump program. Similar techniques could be used to check further properties of the scheduled tasks. Two typical requirements of the mine pump program are

- Turning the pump on when the water is high has a bound of 100 milliseconds.
- Emergency shut down following a high methane value reading has a bound of 30 milliseconds.

In order to verify these properties, first identify the start and end points of interest (points may be in different tasks), introduce a new clock, reset the clock on leaving the start location and check that the clock always is below its required maximum when we reach the end point. Other properties such as schedule length and how often a task is interrupted can be determined similarly. Finally, it will be interesting to experiment with new generic timed automata specifications for different scheduling algorithms.

References

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

[2] Braberman, V. A. and M. Felder, *Verification of real-time designs: Combining scheduling theory with automatic formal verification*, in: O. Nierstrasz and M. Lemoine, editors, *Software Engineering-ESEC/FSE '99*, Lecture Notes in Computer Science **1687**, 1999, pp. 494–510.

[3] Bradley, S., W. Henderson and D. Kendall, *Using timed automata for response time analysis of distributed real-time systems*, in: A. H. Frigeri, W. A. Halang and S. H. Son, editors, *24th IFAC/IFIP Workshop on Real-Time Programming WRTP '99*, 1999, pp. 143–148.

[4] Burns, A., B. Dobbing and G. Romanski, *The Ravenscar tasking profile for high integrity real-time programs*, in: *Reliable Software Technologies-Ada-Europe '98*, Lecture Notes in Computer Science **1411** (1998), pp. 263–275.

[5] Burns, A. and A. Wellings, "Real-Time Systems and Programming Languages," Addison-Wesley, 1997.

[6] Corbett, J. C., *Timing analysis of Ada tasking programs*, IEEE Transactions on Software Engineering **22** (1996), pp. 1–23.

[7] Ericsson, C., A. Wall and W. Yi, *Timed automata as task models for event-driven systems*, in: *Proceedings of RTSCA 99* (1999).

[8] Gerdsmeier, T. and R. Cardell-Oliver, *Analysis of scheduling behaviour with timed automata - protected objects*, Technical Report CSM-339, Department of Computer Science, University of Essex (October 2000).

[9] Joseph, M., editor, "Real-time Systems Specification, Verification and Analysis," Prentice Hall, 1996.

[10] Larsen, K. G. and P. Pettersson, *Uppaal2k*, Newsletter 10, BRICS (1999).

[11] Larsen, K. G., P. Pettersson and W. Yi, Uppaal *in a Nutshell*, Int. Journal on Software Tools for Technology Transfer **1** (1997), pp. 134–152.

[12] Liu, C. L. and J. Layland, *Scheduling algorithms for multiprogramming in a hard real-time environment.*, Journal of the ACM **20** (1973), pp. 46–61.