ELENA FERSMAN

A Generic Approach to Schedulability Analysis of Real-Time Systems

ABSTRACT


This thesis presents a framework for design, analysis, and implementation of embedded systems. We adopt a model of timed automata extended with asynchronous processes i.e. tasks triggered by events. A task is an executable program characterized by its worst-case execution time and deadline, and possibly other parameters such as priorities etc. for scheduling. The main idea is to associate each location of an automaton with a task (or a set of tasks). A transition leading to a location denotes an event triggering the tasks and the clock constraint on the transition specifies the possible arrival times of the event. This yields a model for real-time systems expressive enough to describe concurrency and synchronization, and tasks with (or without) combinations of timing, precedence and resource constraints, which may be periodic, sporadic, preemptive and (or) non-preemptive. We believe that the model may serve as a bridge between scheduling theory and automata-theoretic approaches to system modelling and analysis.

Our main result is that the schedulability checking problem for this model is decidable. To our knowledge, this is the first general decidability result on dense-time models for real time scheduling without assuming that preemptions occur only at integer time points. The proof is based on a decidable class of updatable automata: timed automata with subtraction in which clocks may be updated by subtractions within a bounded zone. As the second contribution, we show that for fixed priority scheduling strategy, the schedulability checking problem can be solved by reachability analysis on standard timed automata using only two extra clocks in addition to the clocks used in the original model to describe task arrival times. The analysis can be done in a similar manner to response time analysis in classic Rate-Monotonic Scheduling. We believe that this is the optimal solution to the problem. The third contribution is an extension of the above results to deal with 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 techniques. Finally, to demonstrate the applicability of the framework, we have modelled, analysed, and synthesised the control software for a production cell. The presented results have been implemented in the TIMES tool for automated schedulability analysis and code synthesis.

Elena Fersman, Department of Information Technology, Uppsala University, Box 337, SE-751 05 Uppsala, Sweden.

© Elena Fersman 2003

ISSN 1104-2516
ISBN 91-554-5774-6
Printed in Sweden by Elanders Gotab, Stockholm 2003
Distributor: Uppsala University Library, Box 510, SE-751 20 Uppsala, Sweden
To my parents
Acknowledgements

First of all I would like to thank my supervisor, Professor Wang Yi, for taking me on as one of his graduate students, and for his excellent guiding and support during the years of my graduate studies. He has never been short on research ideas that he generously shares with his students. Wang helped me at the beginning with many practical problems, when I was a ‘stranger’ in a new world. I want to thank him also for always being able to discuss ideas and answer my questions, including those not related to real-time systems.

I would like to thank current and former members of the UPPAAL group at Uppsala University: Paul Pettersson, Leonid Mokrushin, Tobias Amnell, Alexandre David, Johan Bengtsson, Fredrik Larsson, John Håkansson, Annika Karlsson, and Pavel Krčál for scientific discussions and great company. Specially I would like to thank Paul for his encouraging support and great sense of humor.

I co-authored my first paper together with Professor Bengt Jonsson. I wish to thank him for sharing his knowledge with me. I would also like to thank everyone else at DoCS for making the department such a friendly environment. Special thanks to Anne-Marie Nilsson, Inga-Lisa Ericsson and Marianne Ahrne for help with practical issues at the department.

I am grateful to Professor Igor Chernorutsky and Professor Oleg Tsybin at St. Petersburg State Technical University and Olov Ågren at Uppsala University for establishing contact between these universities, and the Swedish Institute for financial support for this this collaboration, which gave me the opportunity of coming to Sweden as a research student.

Many thanks to my wonderful friends Anna Widenius, Youri Tsybin, Lena and Andrej Savin, Magnus Palmblad, Tatiana Orlova, Leonid Mokrushin (again), Denis Loginov, Elena Kolodizki, and Vassili Alfimov for all fantastic time that we spent together.

Finally, I would like to thank my mother and father, Irina and Gennady Fersman, and my brother Vadim Fersman, who, even from afar, were always present and provided me with so much love, support and inspiration.

This work has been partially supported by Swedish Board for Technical Development (NUTEK), the Swedish Technical Research Council (TFR), and EC via the AIT-WOODDES project.
This thesis includes, summarises and discusses mainly the results presented in four research papers written between 1999 and 2003. These papers are listed as follows:


**Comments on My Participation**

**Paper A:** I participated in discussions and implemented the algorithm. I wrote a part of the paper.

**Paper B:** I participated in discussions, designed and implemented the algorithms. I wrote a large part of the paper.

**Paper C:** I designed and implemented the algorithms and wrote the paper.

**Paper D:** I participated in discussions, modeling and the analysis of the case study. I wrote a part of the paper.
Apart from the papers listed above, I have also participated in the following work:


# Contents

## Introduction 1

1. Background ........................................ 1
2. Real-Time Scheduling ............................ 4
   2.1 Task Model ........................................ 4
   2.2 Task Constraints ................................. 6
   2.3 Classification of Scheduling Algorithms ...... 8
   2.4 Optimal Scheduling .............................. 11
   2.5 Handling Shared Resources ..................... 13
3. The Theme and Contributions of This Thesis .... 16
   3.1 A Unified Model for Timed Systems .......... 17
   3.2 Schedulability Analysis ......................... 18
4. Related Work ....................................... 21
5. Conclusions ........................................ 22

## Paper A: Timed Automata with Asynchronous Processes: Schedulability and Decidability 27

1. Introduction ........................................ 29
2. Timed Automata with Tasks ....................... 32
3. Schedulability Analysis ............................ 36
4. Decidability and Proofs ............................ 39
### Paper B: Schedulability Analysis Using Two Clocks

1. Introduction ........................... 55
2. Preliminaries .......................... 57
   2.1 Timed Automata with Tasks .............. 57
   2.2 Schedulability and Decidability .......... 59
3. Main Result: Two Clocks Encoding ............ 60
4. Analysing Data-Dependent Control ............ 65
   4.1 Extended Timed Automata with Data Variables ... 65
   4.2 Schedulability Analysis ................ 67
5. Implementation .......................... 73
6. Conclusions and Related Work ................ 74

### Paper C: Handling Precedence and Resource Constraints in Schedulability Analysis Using Timed Automata

1. Introduction ........................... 81
2. A Generic Task Model ..................... 83
   2.1 Tasks Parameters and Constraints .......... 83
   2.2 Timed Automata as Task Arrival Patterns .... 85
3. Operational Semantics ..................... 87
4. Schedulability Analysis .................... 91
   4.1 Timing Constraints ...................... 92
   4.2 Precedence Constraints ................ 94
   4.3 Resource Constraints .................. 95
5. Implementation .......................... 98
## Paper D: Code Synthesis for Timed Automata

1. Introduction ................................................. 107
2. The Design Language ........................................ 109
   2.1 Syntax ................................................. 109
   2.2 Operational Semantics ............................... 111
   2.3 Analysis of Design Model ............................ 114
   2.4 Deterministic and Executable Semantics .......... 115
3. Code Synthesis ............................................. 116
   3.1 Handling Tasks and Variables ....................... 117
   3.2 Encoding and Executing the Controller Automata ... 118
   3.3 Correctness .......................................... 121
   3.4 Prototype for legOS ................................. 123
4. Modelling and Design of a Production Cell ........... 124
   4.1 Overall Control Structure ............................ 125
   4.2 Robot Controller Model ................................ 126
   4.3 Feed Belt Controller Model ........................... 130
5. Analysis and Code Synthesis for the Production Cell .... 131
   5.1 Generated legOS-code ................................ 133
6. Conclusions ................................................. 134
A. The Analysis Model ........................................ 137
   A.1 Query File ........................................... 141
B. Generated Code ........................................... 144
   B.1 Production Cell Code .................................. 144
   B.2 Run-time system ...................................... 147
   B.3 Header file .......................................... 148
Introduction

1 Background

Nowadays our society is becoming more and more dependent on computers. We have digital processors embedded everywhere in our environment e.g. in cars, trains, aircrafts, medical devices. According to statistics [Hal00, Tur02] more than 98% of processors produced today are applied in these "non-computing" systems, and they are no longer visible to the customer as computers in the ordinary sense. These systems are known as embedded systems.

Most embedded systems can be characterized as real-time systems. In a real-time system the correctness of the computations depends not only on their logical correctness, but also on the time at which the result is produced, which means that, an answer not in time is a wrong answer. An example of real-time system is a robot used to pick up an object from a conveyor belt. The object is moving, and the robot must pick up the object within a given time slot. If the robot operates too slowly, it will miss the object even though it moves to the right place. On the other hand if the robot operates too quickly, the object will not be there yet, and the robot may block it.

A typical architecture of real-time system is shown in Figure 1. It consists of three main parts: control software, hardware resources, and a scheduling unit. Hardware resources include one or more processors, memory, peripheral devices, etc. The environment communicates with the system through sensors and actuators. Normally the control software is organized as a set of concurrent tasks. A task is a computation entity that is triggered by the target environment, and released for execution on the processor. The scheduling unit decides the order of task execution.

A real time system is usually required to satisfy certain timing constraints. The most used timing constraint is *deadline* i.e. the time point before which the ex-
Introduction

A key issue in the development of real-time systems is ensuring that the expected result is computed and delivered on time. According to the effects of missing a deadline, real-time systems are often classified as either hard or soft as shown in Figure 2, where $v(t)$ is a cost measure associated with a task as a function of its completion time. In a hard real-time system, missing a deadline may cause catastrophic consequences like damage of equipment or even death of people. For example, flight control and anti-lock braking systems in a car are typical hard real-time systems. In a soft real-time system, missing a deadline can only decrease system performance. Examples of soft real-time systems are user interfaces, multimedia applications, computer simulation etc.

Figure 1: Real-Time System Components.

Figure 2: Illustration of different types of real-time systems.

In this thesis, we will study hard real-time systems.
ment of such systems is to ensure that the computations in the system complete within their deadlines. This is a non-trivial problem due to dependencies between tasks and mixed task constraints. In a single-processor system, the tasks share the processor time according to a predefined scheme, which is called a scheduling policy. The set of rules that, at any time, determines the order of task execution is called a scheduling algorithm. As shown in Figure 3, when tasks are invoked by application software they are stored in the ready queue, which is scheduled according to a chosen scheduling algorithm. For example, when a task with high priority arrives to the ready queue, the currently executing task can be preempted if its priority is lower.

![Figure 3: A queue of tasks ready for execution.](image)

A schedule is an assignment of tasks to the processor time so that each task is executed until completion. In other words, it is a reservation of spatial (processor, memory) and temporal (time) resources for a given task set.

- A schedule is said to be feasible if it meets all application constraints for a given task set.

- A set of tasks is said to be schedulable if there exists at least one scheduling algorithm that can generate a feasible schedule.

- A scheduling algorithm is said to be optimal with respect to schedulability if it can always find a feasible schedule whenever any other scheduling algorithm can do so.

In the following section we give a brief introduction to classical real-time scheduling theory.
2 Real-Time Scheduling

The primary goal of research on scheduling is to develop techniques for checking the schedulability of application tasks and finding feasible as well as optimal schedules. Roughly speaking the procedure of finding schedules is known as scheduling algorithm based on task models for a given set of task constraints.

2.1 Task Model

A task (or task type) is an executable program. 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 worst case execution time, deadline etc. Let \( P \) 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. In the following we describe task parameters and task arrival patterns, that are the main elements for the standard notion of task model.

Task parameters. The most important task parameters are worst-case execution time (WCET) and deadline.

WCET, denoted by \( C \), is the maximal time it can take for a task to execute on a given platform. Estimation of worst-case execution times for tasks executing on various architectures is a wide research area. A number of tools for WCET calculation have been developed \[\text{HLS00, EES03}\].

Deadline, denoted by \( D \), is a typical timing constraint of a task. Deadline represents the time before which a task should complete its execution without causing any damage to the system. We will consider relative deadlines, i.e. the time counted from the task arrival.

Task parameters can be classified as follows:

- The static parameters of a task describe characteristics of the task that are independent from the other tasks in the system. Examples of such parameters are WCET, deadline, period (for periodic tasks), static priority. These parameters are derived from the system specification or implementation.
• The *dynamic* parameters of a task describe effects that occur during the task execution. These are for example start time, blocking time, completion time, response time and dynamic priority. Such parameters are derived from characteristics of the other tasks and the run-time scheduling policy.

![Figure 4: Static and dynamic task parameters.](image)

Figure 4 shows two task instances. Static task parameters are shown on the first instance, and dynamic parameters are shown on the second one. We shall use $P_{ij}$ to denote the $j$-th instance of the task $P_i$. Static parameters of the task $P_i$ are WCET ($C_i$), deadline ($D_i$), period ($T_i$), offset ($O_i$). Parameters of the task instance $P_{ij}$ are arrival time ($a_{ij}$), start time ($s_{ij}$), finish (or completion) time ($f_{ij}$), response time ($R_{ij}$). Other parameters that can characterize a task are the following:

- **Criticalness**: a parameter related to consequences of missing deadline (typically, hard or soft);
- **Lateness** $L_{ij}$: the delay of a task completion with respect to its deadline, $L_{ij} = f_{ij} - D_i$; if the task instance completes before its deadline, $L_{ij}$ is negative;
- **Laxity** $X_{ij}$: a maximum time a task can be delayed after its activation to complete within its deadline, $L_{ij} = D_i - a_{ij} - C_i$.

**Task arrival patterns.** The classical literature distinguishes three types of task arrival patterns:

- **Periodic tasks** arrive periodically according to a constant interval $T$, i.e. the *period* of the task. For example, temperature and gas level monitoring are typical periodic tasks. An advantage of using models with such behaviour is that scheduling algorithms and analysis techniques for such tasks are well-studied.
• *Sporadic tasks* are supposed to arrive within varying intervals of time, but with a given minimal inter-arrival time. These tasks typically are reactions on signals from the environment, when it takes some time for the environment to request the next task invocation. Typical example of sporadic tasks is sampling of environmental values.

• *Aperiodic tasks* execute at irregular intervals and have only soft deadlines, but adequate response times are desirable. For example, an aperiodic task may process user input from a terminal.

### 2.2 Task Constraints

There are typically three types of constraints specified on tasks: *timing* constraints such as 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.

**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, execution times can be also viewed as (timing) constraints meaning that the tasks can not consume more than the given execution times.

**Precedence constraints.** The execution of a task set may have to respect some precedence constraints between tasks. These constraints are usually caused by data flow, and impose a partial order on a task set. An example of a precedence graph is shown in Figure 5. The task that computes the final result in
the end can not start before the two tasks responsible for computation of intermediate result complete their execution, and each of those can start only after completion of a task reading a corresponding sensor. Note that there is no restriction on the order of the tasks ReadSensor1 and ReadSensor2, as well as the order of ComputeValue1 and ComputeValue2.

**Resource constraints.** During execution, tasks may need to access certain resources. A resource can be a variable, a data structure, a file, etc. For data consistency, many resources forbid simultaneous access. Such shared resources are called exclusive.

To ensure exclusive access to system resources, most operating systems provide semaphore mechanism. A semaphore is used to represent an exclusive resource. When the semaphore is locked by a task, the resource cannot be accessed by the other tasks. Each task may have locked a set of semaphores, i.e. shared resources that it has got the access during its execution. An access pattern to a semaphore $S$ protecting a resource shared by two tasks is shown in Figure 6. The higher priority task $P_H$ enters the critical section after 1 time unit of execution and locks the semaphore $S$ for 2 time units. Lower priority task $P_L$ locks $S$ for 4 time units after 2 time units of normal execution. The execution of these two tasks must preserve mutual access to the semaphore.

In Figure 7 an example of the task execution is shown. At time 3 the lower priority task $P_L$ locks the semaphore $S$ and enters the critical section. At time 5, still being in the critical section, $P_L$ is preempted by the higher priority task...
At time 6 \( P_H \) fails to enter the critical section because \( S \) is still locked by \( P_L \), therefore \( P_L \) resumes its execution until it releases the semaphore at time 8 when \( P_H \) can lock the semaphore and continue its execution.

\[ \text{Figure 7: Blocking on an exclusive resource.} \]

### 2.3 Classification of Scheduling Algorithms

A scheduling algorithm is the set of rules that, at any time, determines the order of task execution in a given task set. Usually scheduling algorithms are classified as preemptive or non-preemptive:

- **Non-preemptive.** A task, when started, completes its execution without interruptions. The advantage of this method is that mutual exclusion of shared resources is automatically guaranteed. However, non-preemptive scheduling has negative effect on schedulability because a scheduling decision takes effect only after a task has been completed.

- **Preemptive.** Execution of a running task can be interrupted at any time if a higher priority task has arrived. This method provides better processor utilization preventing any process from monopolising the processor. However, task switching causes more overhead and need of forcing mutual exclusion, by use of e.g. semaphores and resource access protocols.

Scheduling algorithms can be further classified as shown in Figure 8.

**Off-line scheduling.** The idea of off-line scheduling is that the schedule is computed before run-time. It is stored in a table and later executed by a dispatcher. The most popular off-line scheduling approach is *round-robin*. With this method tasks are checked for readiness in a predefined order, and ready tasks are executed immediately. The processor executes a task for only a single quantum of time before moving on to the next task. The quantum is a parameter that can be changed. Long quantums result in first-come-first-served (FCFS)
The advantage of this approach is that it is easy to implement. However, there are situations when an urgent task has to wait for all other tasks to execute before it gets a chance to start. This gives a negative effect on schedulability.

A slight variation of round-robin is cyclic executive approach. As in round-robin method, tasks are checked for readiness in a predefined order, but each task can be checked in the cycle several times. The complete schedule is computed so that its repeated execution will cause all tasks to run at their correct rates. The complete schedule is called major cycle, which typically consists of a fixed number of minor cycles of fixed duration. For example, a task set shown in Figure 9 can be executed according to the schedule shown in Table 1. Minor cycle is 10, major cycle is 40.

Table 1 can be executed according to the schedule shown in Figure 9 and the code for such system will look as follows:
begin loop:
Wait_for_interrupt;
P1; P2;
Wait_for_interrupt;
P1; P3; P4;
Wait_for_interrupt;
P1; P2;
Wait_for_interrupt;
P1; P3; P5;
end loop;

Each minor cycle is a sequence of procedure calls. The procedures share common address space, they can easily pass data between themselves, and there is no need for semaphores because concurrent access is not possible. However, there is a number of disadvantages in this method. First of all, whenever a cyclic schedule is constructed, adding another task requires recalculation of the whole schedule, and construction of a cyclic schedule is a NP-hard problem. Secondly, sporadic tasks, as well as tasks with long periods, are difficult to incorporate because the major cycle is the length of the maximal period.

**On-line scheduling.** The idea of on-line scheduling is that scheduling decisions are taken at runtime every time a new task enters the system or a running task terminates. On-line scheduling can be classified as scheduling with static or dynamic priorities, when scheduling decisions are based on static or dynamic task parameters respectively.

On-line scheduling has become of significant interest after pioneering work of Liu and Layland [LL73] on rate-monotonic scheduling (see Section 2.4), where scheduling follows a preemptive static priority scheme, and at any point in time a task with the fixed highest priority among the enabled tasks is executing.

One of the most popular scheduling algorithms with dynamic priorities is earliest deadline first (EDF) (see Section 2.4), where the priority of an enabled task depends on the time left until its deadline expires. Dynamic priorities can also depend on other dynamic characteristics of task execution. For example, in Linux, tasks that have not received the processor for a long time, get their priorities increased, and tasks that have been executing on the processor often, have their priorities decreased.
2.4 Optimal Scheduling

The criterion which is mostly used to measure the performance of a scheduling algorithm is its ability to find feasible schedules for a given task set whenever such schedules exist. A scheduling algorithm is said to be optimal with respect to schedulability if it can always find a feasible schedule in case the given set of tasks has feasible schedules. Conversely, if an optimal algorithm fails to find a feasible schedule, then the given task set cannot be feasibly scheduled with any algorithm. This subsection gives examples of well-studied optimal scheduling algorithms.

Earliest Deadline First

An algorithm with dynamic priority assignment for scheduling a set of aperiodic tasks on a single processor called earliest deadline first (EDF) was first described in [Jac55]. This algorithm is optimal for scheduling preemptable tasks on single-processor systems. The idea of EDF is that at any point in time the priority of an enabled task depends on the time left until its deadline expires. According to EDF, the executing task must always have least time remaining until its deadline among all enabled tasks.

For periodic tasks, sufficient and necessary condition for schedulability of a task set scheduled according to EDF is that processor utilization is less or equal $1$, i.e.

$$\sum_{i=1}^{N} \frac{C_i}{T_i} \leq 1$$

However, despite attractive theoretical properties of EDF, it is not widely used in embedded systems design due to the costly runtime overhead of EDF scheduling.

EDF is not optimal when tasks are non-preemptable or when there is more than one processors.

Rate-Monotonic Scheduling

One of the most well-studied and widely used scheduling algorithms is Rate-Monotonic (RM) scheduling algorithm [LL73]. It is optimal for scheduling preemptable tasks with static priorities on one processor. This approach assumes that the task set has the following properties:
tasks are released periodically, with constant and known interval between
invocations;
- tasks are independent, i.e. invocation or execution of a certain task does
  not depend on the executions of other tasks;
- each task must be completed before the next instance of it is released.

The idea is to determine fixed priorities by task frequencies. Tasks with higher
rates, i.e. shorter periods, are assigned higher priorities. This algorithm is opti-
mal in a sense that if a rate-monotonic assignment is not feasible, then the task
set is not schedulable, under assumption that tasks have deadlines equal to their
periods.

**Sufficient Test for Schedulability**

A set of $n$ independent periodic tasks, with deadlines equal to periods and
scheduled by the RM algorithm will always meet its deadlines, if

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$$

This is a sufficient schedulability test for RM scheduling algorithm. When $n \to \infty$ the utilization bound approaches $ln \ 2 \approx 0.693$.

**Precise Test for Schedulability**

Exact schedulability test for fixed priority scheduling algorithms, called re-
sponse time analysis, was suggested by Joseph and Pandya [JP86]. For any
task $P_i$ worst-case response time is given by formula:

$$R_i = C_i + I_i$$

where $I_i$ is the maximal interference that task $P_i$ can experience during the
interval $[t, t + R_i]$. For the highest priority task, the worst-case response time is
its worst-case execution time, i.e. $R = C$. Other tasks will suffer interference
from higher-priority tasks, which has the following value:

$$I_i = \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

The worst possible response can then be calculated using the response time
formula:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$
Table 2: A periodic task set to be scheduled using DM and EDF strategies.

<table>
<thead>
<tr>
<th>Task</th>
<th>C</th>
<th>T</th>
<th>D</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_H )</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>High</td>
</tr>
<tr>
<td>( P_L )</td>
<td>4</td>
<td>7</td>
<td>7</td>
<td>Low</td>
</tr>
</tbody>
</table>

The response time analysis calculates the time when a task will complete its execution if it is activated at the same time as all other tasks with higher priority. The summation gives us the total time the tasks with higher priority will execute before task \( P_H \) has completed. The response time can be calculated by solving the above equations.

Response time analysis is a sufficient and necessary schedulability test, i.e. if a task set passes the test, then none of the tasks will miss a deadline, and if the test is failed then at run-time a task will miss its deadline.

**Deadline-Monotonic Scheduling**

*Deadline-Monotonic* (DM) scheduling algorithm [LW82] is a generalization of Rate-Monotonic algorithm, when task deadlines can be less or equal than periods. According to their priority assignment, tasks with shorter deadlines get higher priorities. Deadline-Monotonic algorithm is optimal among scheduling algorithms that use static priorities under the assumption that task deadlines are less or equal than periods for all tasks.

Note that deadline-monotonic approach has lower processor utilization than EDF, and therefore, worse schedulability results. For example, Table 2 and Figure 10 shows a task set which is not schedulable by deadline monotonic assignment, but schedulable using EDF. Processor utilization for this task set is 0.97.

### 2.5 Handling Shared Resources

Now we relax the assumption about independency of tasks. Concurrent tasks can share common resources, and if a preemptive scheduling policy is used, mutual exclusion of accesses to shared resources has to be guaranteed.

One of the main problems to solve when common resources are protected by semaphores is priority inversion phenomenon shown in Figure 11. Here, in addition to two tasks described in Figure 6 that share a semaphore, we add a task
Introduction

Figure 10: Gantt charts for the task set from the Table 2 scheduled using deadline-monotonic and EDF strategies.

$P_M$ with intermediate priority level. At time 7, when the highest priority task $P_H$ is blocked on semaphore used by the lowest priority task $P_L$, the task $P_M$ is released and, preempting $P_L$ starts executing. As a result, the execution of $P_M$ delays the execution of $P_H$ even though the priority of $P_H$ is higher and these tasks do not share any resources.

Figure 11: Priority inversion phenomenon.

To avoid priority inversion problem, various resource access protocols have been developed in the literature [SRL90, RSL98].

Priority Inheritance Protocol

The basic idea of Priority Inheritance Protocol [SRL90] is that a task blocking one or more higher-priority tasks temporarily inherits the highest priority of the blocked tasks. This idea is demonstrated in Figure 12. At time 6, when the
highest priority task $P_H$ is blocked by $P_L, P_L$ inherits the priority of $P_H$, which prevents preemption by $P_M$ and the priority inversion problem.

![Figure 12: Priority inheritance.](image)

However, the main disadvantage of this protocol is that it can cause deadlocks. For example, if tasks $P_H$ and $P_L$ share two semaphores, $s_1$ and $s_2$, then the following situation can occur:

- the lower priority task $P_L$ locks $s_1$,
- right after that task $P_H$ runs and locks $s_2$ and then tries to lock $s_1$, which it cannot lock because $P_L$ is blocking it,
- then task $P_L$ inherits the priority of $P_H$, starts running, and tries to lock $s_2$, which is blocked by $P_L$; neither task can execute, hence the system deadlocked.

Another disadvantage of priority inheritance protocol is that it can cause chained blocking, i.e. a task can be blocked several times during its execution.

**Priority Ceiling Protocol**

The basic idea of Priority Ceiling Protocol [SRL90] is that each resource is assigned a priority ceiling equal to the priority of the highest-priority task that can lock it. Then, a task $P_i$ is allowed to enter a critical section, i.e. lock a semaphore, only if its priority is higher than all priority ceilings of the resources currently locked by tasks other than $P_i$. And as before, when a task blocks on a semaphore $s$, the task currently holding $s$ inherits the priority of that task.

This protocol is deadlock-free, and chained blocking is not possible, i.e. a task can be blocked at most a duration of one critical section. Now the response time
for task $P_i$ can be calculated as follows:

$$R_i = C_i + B_i + \sum_{j \in h_p(i)} \left[ \frac{R_j}{T_j} \right] C_j$$

The formula for response time calculation includes term $B_i$, which tells how long the task is blocked by semaphores. A given task $P_i$ can be blocked at most one critical section of any lower priority task locking a semaphore with priority ceiling greater than or equal to the priority of the task $P_i$. Therefore, the blocking time is calculated as follows:

$$B_i = \max\{cs_{P_{h,i}}|P_k \in lp(i) \land s \in uses(P_k) \land ceil(s) \geq pri(P_i)\}$$

where $cs_{P_{h,i}}$ is the length of the critical section where the task $P_k$ locks the semaphore $s$, $lp(i)$ is the set of all lower priority tasks than $P_i$, $uses(P_k)$ is the set of semaphores used by the task $P_k$, $ceil(s)$ is the priority ceiling of the semaphore $s$ and $pri(P_i)$ is the priority of $P_i$.

**Highest Locker Protocol**

Highest Locker Protocol [RSL98], also known as immediate inheritance protocol, priority ceiling emulation, and stack based priority-ceiling protocol, is a slight modification of the Priority Ceiling Protocol. According to the Highest Locker Protocol, whenever a task succeeds in locking a semaphore $s$, its priority is changed dynamically to the maximum of its current priority and $ceil(s)$. When the task unlocks $s$, it sets its priority back to what it was before.

With this protocol, a task is blocked while attempting to preempt, and not when entering a critical section. This protocol is easier to implement than Priority Ceiling Protocol. Blocking time calculation is the same as for Priority Ceiling Protocol.

### 3 The Theme and Contributions of This Thesis

The central topic of this thesis is schedulability analysis. We notice that the results on schedulability analysis published in the literature and summarized in the previous section are based on the assumption that tasks are released periodically. Indeed, many activities in real-time embedded systems are periodic: audio sampling and sample processing, temperature and speed monitoring, etc. However, often applications require certain tasks to be executed when a request
from the environment occurs. Such requests for task execution can occur as long as the system is in function thus generating infinite sequences of aperiodic tasks.

To deal with non-periodic tasks in event–driven systems, the standard method is to consider non-periodic tasks as periodic using the estimated minimal inter-arrival times as task periods. Clearly, the analysis based on such task model is pessimistic in many cases, i.e. a task set which is schedulable may be considered as non-schedulable as the inter-arrival times of the tasks may vary over time, that are not necessary minimal. To achieve more precise analysis, we need task models that allow more precise and relaxed timing constraints.

One of the successful theories for modelling and analysis of timed systems is timed automata [AD94]. The advantage of using timed automata in system modelling is that one can specify relaxed timing constraints on events (i.e. discrete transitions). Moreover, timed automata also allow to model other behavioral aspects of systems such as synchronization and concurrency. However, it is not clear how timed automata can be used for schedulability analysis because there is no support for specifying resource requirements and hard time constraints on computations.

### 3.1 A Unified Model for Timed Systems

We propose to use timed automata extended with asynchronous processes i.e. tasks triggered by events. This model was first presented in [EWY98] and further studied in [FPY02]. The main idea behind the model is to associate each location of an automaton with a task (or a set of tasks in the general case). A task is an executable program characterized by its worst case execution time and deadline, and possibly other parameters such as priorities etc for scheduling. Intuitively a transition leading to a location in the automaton denotes an event triggering the task and the guard (clock constraints) on the transition specifies the possible arrival times of the event. Semantically, an automaton may perform two types of transitions. Delay transitions correspond to the execution of running tasks (with highest priority) and idling for the waiting tasks. Discrete transitions correspond to the arrival of new task instances. Whenever a task is triggered, it is put in the scheduling queue for execution (i.e. the ready queue in operating systems) according to a given scheduling strategy e.g. FPS (fixed priority scheduling) or EDF (earliest deadline first).

This model unifies timed automata with the classic task models from schedul-
ing theory. It can be used to specify resource requirements and hard timing constraints on computations in addition to features offered by timed automata. It is general and expressive enough to describe concurrency and synchronization, and tasks which may be periodic, sporadic, preemptive and (or) non-preemptive, and have combinations of timing, precedence and resource constraints. We will use such extended timed automata to describe design models for timed systems and study scheduling problems for such systems. Roughly speaking a design model we will study in this thesis may contain the following elements:

- a set of tasks, where each task is an executable program with timing constraints: execution time and deadline,
- a set of precedence constraints (a precedence graph) for the task set,
- a set of resource constraints (semaphore access patterns) for the task set,
- a set of automata (or a single automaton) that describes how the tasks are triggered,
- a set of data variables shared between tasks, which may be read and tested by the automata and the tasks, and updated by the tasks,
- a set of data variables shared between automata, which may be read and updated by the automata,
- a set of communication channels for handshaking synchronization of the automata.

3.2 Schedulability Analysis

Following the work of [EWY98], we shall formalize the notion of schedulability in terms of reachable states. A state of an extended automaton is a triple \((l, u, q)\) consisting of a location \(l\), a clock assignment \(u\) and a task queue \(q\). The task queue contains pairs of remaining computing times and relative deadlines for all released tasks. Naturally, a state \((l, u, q)\) is schedulable if \(q\) is schedulable in the sense that there exists a scheduling strategy such that all tasks in \(q\) can be computed within their deadlines. An automaton is schedulable if all its reachable states are schedulable.

In [EWY98], it has been shown that for any non-preemptive scheduling strategy, the schedulability checking problem can be transformed to a reachability problem for ordinary timed automata and thus it is decidable. For preemptive
scheduling strategies, it has been suspected that the schedulability checking problem is undecidable since in preemptive scheduling we may need to use stop-watches to accumulate computing times for tasks. In this thesis we prove that the problem is decidable. The main idea of the proof is to model scheduling strategies with variants of timed automata (not stop-watch automata), and then encode the schedulability analysis problem as a reachability problem. Note that the preemptive earliest deadline first algorithm (EDF) is optimal in the sense that if EDF can not schedule a task set, no other algorithms can. Thus to check an extended timed automaton for schedulability it suffices to check if it can be scheduled with EDF. In the following we summarize briefly the technical contributions of this thesis.

**EDF.** To encode a dynamic scheduling policy like EDF we need at least two clocks for each task: execution time clock to accumulate the task execution time, and deadline clock to check for deadline violation. When more than one instance of a task is allowed in the system, we also need to use a deadline clock for each of the waiting task instances (we do not need more execution time clocks because instances of the same task do not preempt each other). Hence, for the general case we need $N + \sum_{i=0}^{N-1} \left\lfloor \frac{D[P_i]}{C[P_i]} \right\rfloor$ clocks to check schedulability, where $N$ is the number of task types in the system, $D(P)$ is the relative deadline of task $P$, and $C(P)$ is the WCET of $P$. For systems where only one instance per task type is allowed, we need $2N$ clocks to check if the task set is schedulable with EDF. The result is presented in [Paper A].

**Fixed Priority Scheduling.** For fixed priority scheduling strategies, we have shown that the schedulability checking problem can be solved by reachability analysis on standard timed automata using only two extra clocks in addition to the clocks used in the original model to describe task arrival times. The analysis can be done in a similar manner to response time analysis in classic Rate-Monotonic Scheduling. We check schedulability for each task separately, increasing the task response time whenever a higher-priority task arrives by its execution time constant. One clock is used to keep track of task execution time, and the other one is used to check for deadline violation. The result is presented in [Paper B].

**Precedence Constraints.** When precedence constraints are imposed on the task set, the related schedulability analysis problem can be solved efficiently using the same technique. For every task that must precede other tasks (predecessor) we are interested in the time point when it finishes its execution. Therefore, in the analysis we need one extra clock for each predecessor. The number of clocks can be reduced for precedence relations of the form $P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_n$, 

---

*Introduction 19*
where task $P_{i-1}$ is an immediate predecessor of $P_i$. For each such sequence we can use only one clock to calculate when any task in the sequence finishes its execution due to the fact that the tasks will execute consequently, and the clock can be reused for any predecessor in the sequence. Hence, for the analysis we need $k$ extra clocks, where $k$ is the number of such sequences. The result is presented in [Paper C].

**Shared Resources.** We use semaphores to represent logical resources, i.e. data shared between tasks, which requires mutually exclusive access. We define a semaphore access pattern of a task as a sequence of triples $\{(\text{OP}_i, \text{T}_i, \text{A}_i)\}$, where $\text{OP}_i$ is the semaphore operation to be performed on a semaphore (i.e. locking or unlocking), $\text{T}_i$ is the computation time for performing $\text{OP}_i$, and $\text{A}_i$ is a sequence of assignments updating the set of shared variables.

For schedulability analysis of such models, it is important to know the status of semaphores at any time point. Hence, for each task that uses one or more semaphores, we need to use one clock that would run along the task access pattern and trigger the events of locking and unlocking the semaphores. This results in $l$ extra clocks, where $l$ is the number of tasks that use shared resources protected by semaphores. The result is presented in [Paper C].

**Data-Dependent Control.** In systems with data-dependent control the release times of a task may depend on the values of data variables, shared between tasks and the automata, and hence on the time-points at which other tasks finish their execution. For such systems we need to use additional clocks for keeping track of execution of the tasks that have shared variables with the control automata. We shall use one additional clock for each task type that updates variables shared between control automata and/or other tasks. In this case the schedulability checking problem uses $m$ extra clocks, where $m$ is the number of tasks types that update the shared variables. However, this result is applicable only when exact execution times of tasks are known. When the task execution times are given as intervals, an over-approximation technique can be used. The result is presented in [Paper B].

**Combination of Constraints.** The problems and solutions described above are in fact orthogonal to each other. In the worst case, for fixed priority scheduling the upper bound on the number of clocks for checking schedulability is $2 + k + l + m$. However, this number can be decreased by reuse of clocks, because each task involved in a mixture of constraints we need only one extra clock.
4 Related Work

For systems restricted to periodic tasks, a numerous number of scheduling techniques have been developed, see e.g. [But97, KS97, Liu00]. In the past years, these classic works have been extended to deal with more complex constraints e.g. unfolding [BLMSv98] for precedence constraints and priority ceiling protocols [SRL90, RSL98] for shared resources. These methods can be extended to handle non-periodic tasks by considering them as periodic with the minimal inter-arrival time as the task periods. For fixed priority periodic tasks with offsets and release jitters of techniques for schedulability analysis have been developed [Tin94, PH98, RT02]. Our work is more related to work on using automata to model and solve scheduling problems.

In [Cor94, CL00], stopwatch automata [ACH+95] are applied to model scheduling algorithms with sporadic tasks and semi-decision algorithms are presented. Timed automata [AD94] have been used to solve non-preemptive scheduling problems mainly for job-shop scheduling [AM01, Feh99, HLP01]. Similarly, stopwatch automata have been used to solve preemptive job-shop scheduling problems in [AM02]. 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 [AGP+99, AGS00, AGS02]. The idea is to achieve schedulability by construction. The authors present a controller synthesis technique that can be used to construct a scheduler to control the system so that all given scheduling constraints in the model are satisfied. An alternative approach is presented in [ZM01] in which the schedulability of a system is established by proving that the specification (formalised in the temporal logic TLA) of the system and the scheduler satisfies the given scheduling constraint.

In [MV94], McManis and Varaiya present a restricted class of stopwatch automata, called suspension automata. We use the idea of replacing suspensions of timers by subtraction of clock values, as suggested in [MV94]. It has been shown in [BDFP00] that updating clock variables by subtraction of integer values in timed automata is undecidable in general. We identify a decidable class of such updatable automata [FPY02], which is precisely what we need to solve scheduling problems.
5 Conclusions

We have studied a model of timed systems, which unifies timed automata with the classic task models from scheduling theory. The model can be used to specify resource requirements and hard time constraints on computations, in addition to features offered by timed automata. It is general and expressive enough to describe concurrency and synchronization, and tasks which may be periodic, sporadic, preemptive and (or) non-preemptive. The classic notion of schedulability is naturally extended to this generic model for timed systems.

The main technical contribution of this thesis is the proof that the schedulability checking problem is decidable. The problem has been suspected to be undecidable due to the nature of preemptive scheduling. To our knowledge, this is the first decidability result for preemptive scheduling in dense-time models. We have shown that for fixed priority scheduling strategy, the schedulability checking problem of timed automata extended with tasks can be solved by reachability analysis on standard timed automata using only two additional clocks. We have extended the result to deal with tasks that have not only timing constraints but also precedence and resource constraints and 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 techniques. We have also shown how to extend the result to systems with data-dependent control, i.e. systems in which the release time-points of a task may depend on the values of shared variables, and hence on the time-point at which other tasks finish their execution.

The presented results have been implemented in the TIMES tool [AFM⁺02, AFM⁺03] for automated schedulability analysis. In addition, we have shown how to synthesise executable code with predictable behaviour from extended timed automata. A case study has been reported using the tool to develop control software for a production cell. We believe that timed automata and our contributions provide a bridge between scheduling theory and automata-theoretic approaches to system modelling and verification for real time systems.

An interesting direction for future work is performance analysis of soft real-time systems. The cost function in such systems can depend on portion of tasks with missed deadlines or the time that it takes to complete released tasks after missed deadlines. Another direction for future work the schedule synthesis. More precisely given an automaton, it is desirable to characterize the set of schedulable traces accepted by the automaton.
References


Introduction


J.R. Jackson. Scheduling a production line to minimize maximal tardiness, management science research project 43. Technical report, University of California, Los Angeles, 1955.


Paper A:

Timed Automata with Asynchronous Processes: Schedulability and Decidability

Elena Fersman, Paul Pettersson, and Wang Yi.
Timed Automata with Asynchronous Processes: Schedulability and Decidability

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

Abstract. In this paper, we extend timed automata with asynchronous processes i.e. tasks triggered by events as a model for real-time systems. The model is expressive enough to describe concurrency and synchronization, and real time tasks which may be periodic, sporadic, preemptive or non-preemptive. We generalize the classic notion of schedulability to timed automata. An automaton is schedulable if there exists a scheduling strategy such that all possible sequences of events accepted by the automaton are schedulable in the sense that all associated tasks can be computed within their deadlines. We believe that the model may serve as a bridge between scheduling theory and automata-theoretic approaches to system modelling and analysis. Our main result is that the schedulability checking problem is decidable. To our knowledge, this is the first general decidability result on dense-time models for real time scheduling without assuming that preemptions occur only at integer time points. The proof is based on a decidable class of updatable automata: timed automata with subtraction in which clocks may be updated by subtractions within a bounded zone. The crucial observation is that the schedulability checking problem can be encoded as a reachability problem for such automata. Based on the proof, we have developed a symbolic technique and a prototype tool for schedulability analysis.

1 Introduction

One of the most important issues in developing real time systems is schedulability analysis prior to implementation. In the area of real time scheduling, there are well-studied methods [But97] e.g. rate monotonic scheduling, that are widely applied in the analysis of periodic tasks with deterministic behaviours. For non-periodic tasks with non-deterministic behaviours, there are no satisfactory solutions. There are approximative methods with pessimistic analysis e.g.
using periodic tasks to model sporadic tasks when control structures of tasks are not considered. The advantage with automata-theoretic approaches e.g. using timed automata in modelling systems is that one may specify general timing constraints on events and model other behavioural aspects such as concurrency and synchronization. However, it is not clear how timed automata can be used for schedulability analysis because there is no support for specifying resource requirements and hard time constraints on computations e.g. deadlines.

Following the work of [EWY98], we study an extended version of timed automata with asynchronous processes i.e. tasks triggered by events. A task is an executable program characterized by its worst case execution time and deadline, and possibly other parameters such as priorities etc for scheduling. The main idea is to associate each location of an automaton with a task (or a set of tasks in the general case). Intuitively a transition leading to a location in the automaton denotes an event triggering the task and the guard (clock constraints) on the transition specifies the possible arrival times of the event. Semantically, an automaton may perform two types of transitions. Delay transitions correspond to the execution of running tasks (with highest priority) and idling for the other waiting tasks. Discrete transitions correspond to the arrival of new task instances. Whenever a task is triggered, it will be put in the scheduling queue for execution (i.e. the ready queue in operating systems). We assume that the tasks will be executed according to a given scheduling strategy e.g. FPS (fixed priority scheduling) or EDF (earliest deadline first). Thus during the execution of an automaton, there may be a number of processes (released tasks) running in parallel (logically).

For example, consider the automaton shown in Figure 1. It has three locations $l_0, l_1, l_2$, and two tasks $P$ and $Q$ (triggered by $a$ and $b$) with computing time and relative deadline in brackets $(2, 10)$, and $(4, 8)$ respectively. The automaton
models a system starting in $l_0$ may move to $l_1$ by event $a$ at any time, which triggers the task $P$. In $l_1$, as long as the constraints $x \geq 10$ and $y \leq 40$ hold and event $a$ occurs, a copy of task $P$ will be created and put in the scheduling queue. However, in $l_1$, it cannot create more than 5 instances of $P$ because the constraint $y \leq 40$ will be violated after 40 time units. In fact, every copy will be computed before the next instance arrives and the scheduling queue may contain at most one task instance and no task instance will miss its deadline in $l_1$. In $l_1$, the system is also able to accept $b$, trigger $Q$ and then switch to $l_2$. In $l_2$, because there is no constraints labelled on the $b$-transition, it may accept any number of $b$’s, and create any number of $Q$’s in 0 time. This is the so-called zeno-behaviour. However, after more than two copies of $Q$, the queue will be non-schedulable. This means that the system is non-schedulable. Thus, zeno-behaviour will correspond to non-schedulability, which is a natural property of the model.

We shall formalize the notion of schedulability in terms of reachable states. A state of an extended automaton will be a triple $(l, u, q)$ consisting of a location $l$, a clock assignment $u$ and a task queue $q$. The task queue contains pairs of remaining computing times and relative deadlines for all released tasks. Naturally, a state $(l, u, q)$ is schedulable if $q$ is schedulable in the sense there exists a scheduling strategy with which all tasks in $q$ can be computed within their deadlines. An automaton is schedulable if all reachable states of the automaton are schedulable. Note that the notion of schedulability here is relative to the scheduling strategy. A task queue which is not schedulable with one scheduling strategy, may be schedulable with another strategy. In [EWY98], we have shown that under the assumption that the tasks are non-preemptive, the schedulability checking problem can be transformed to a reachability problem for ordinary timed automata and thus it is decidable. The result essentially means that given an automaton it is possible to check whether the automaton is schedulable with any non-preemptive scheduling strategy. For preemptive scheduling strategies, it has been suspected that the schedulability checking problem is undecidable because in preemptive scheduling we must use stop-watches to accumulate computing times for tasks. It appears that the computation model behind preemptive scheduling is stop-watch automata for which it is known that the reachability problem is undecidable. Surprisingly the above intuition is wrong. In this paper, we establish that the schedulability checking problem for extended timed automata is decidable for preemptive scheduling. In fact, our result applies to not only preemptive scheduling, but any scheduling strategy. That is, for a given extended timed automata, it is checkable if there exists a scheduling strategy (preemptive or non-preemptive) with which the automaton
is schedulable. The crucial observation in the proof is that the schedulability checking problem can be translated to a reachability problem for a decidable class of updatable automata, that is, timed automata with subtraction where clocks may be updated with subtraction only in a bounded zone.

The rest of this paper is organized as follows: Section 2 presents the syntax and semantics of timed automata extended with tasks. Section 3 describes scheduling problems related to the model. Section 4 is devoted to the main proof that the schedulability checking problem for preemptive scheduling is decidable. Section 5 concludes the paper with summarized results and future work, as well as a brief summary and comparison with related work.

2 Timed Automata with Tasks

Let $\mathcal{P}$, ranged over by $P, Q, R$, denote a finite set of task types. A task type may have different instances that are copies of the same program with different inputs. We further assume that the worst case execution times and hard deadlines of tasks in $\mathcal{P}$ are known $^1$. Thus, each task $P$ is characterized as a pair of natural numbers denoted $P(C, D)$ with $C \leq D$, where $C$ is the worst case execution time of $P$ and $D$ is the relative deadline for $P$. We shall use $C(P)$ and $D(P)$ to denote the worst case execution time and relative deadline of $P$ respectively.

As in timed automata, assume a finite alphabet $\mathcal{A}$ ranged over by $a, b$ etc and a finite set of real-valued clocks $\mathcal{C}$ ranged over by $x_1, x_2$ etc. We use $\mathcal{B}(\mathcal{C})$ ranged over by $g$ 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. The elements of $\mathcal{B}(\mathcal{C})$ are called clock constraints.

**Definition 1** A timed automaton extended with tasks, over actions $\mathcal{A}$, 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
  - $E \subseteq N \times \mathcal{B}(\mathcal{C}) \times \mathcal{A} \times 2^\mathcal{C} \times N$ is the set of edges.

$^1$Tasks may have other parameters such as fixed priority for scheduling and other resource requirements e.g. on memory consumption. For simplicity, in this paper, we only consider computing time and deadline.
– \( I : N \mapsto \mathcal{B}(\mathcal{C}) \) is a function assigning each location with a clock constraint (a location invariant).

– \( M : N \mapsto \mathcal{P} \) is a partial function assigning locations with tasks \(^2\).

Intuitively, a discrete transition in an automaton denotes an event triggering a task annotated in the target location, and the guard on the transition specifies all the possible arrival times of the event (or the annotated task). Whenever a task is triggered, it will be put in the scheduling (or task) queue for execution (corresponding to the ready queue in operating systems).

Clearly extended timed automata are at least as expressive as timed automata; in particular, if \( M \) is the empty mapping, we will have ordinary timed automata. It is a rather general and expressive model. For example, it may model time-triggered periodic tasks as a simple automaton as shown in Figure 2(a) where \( P \) is a periodic task with computing time 2, deadline 8 and period 20. More generally it may model systems containing both periodic and sporadic tasks as shown in Figure 2(b) which is a system consisting of 4 tasks as annotation on locations, where \( P_1 \) and \( P_2 \) are periodic with periods 10 and 20 respectively (specified by the constraints: \( x=10 \) and \( x=20 \)), and \( Q_1 \) and \( Q_2 \) are sporadic or even driven by event \( a \) and \( b \) respectively.

In general, there may be a number of released tasks running logically in parallel. For example, an instance of \( Q_2 \) may be released before the preceding instance of \( P_1 \) is finished because there is no constraint on the arrival time of \( b_2 \). This means that the queue may contain at least \( P_1 \) and \( Q_2 \). In fact, instances of all four task types may appear in the queue at the same time.

---

\(^{2}\)Note that \( M \) is a partial function meaning that some of the locations may have no task. Note also that we may also associate a location with a set of tasks instead of a single one. It will not introduce technical difficulties.
**Parallel Composition.** To handle concurrency and synchronization, a parallel composition of extended timed automata may be defined as a product automaton in the same way as for ordinary timed automata (e.g. see [LPY95]). Note that the parallel composition here is only an operator to construct models of systems based on their components. It has nothing to do with multi-processor scheduling. A product automaton may be scheduled to run on a one- or multi-processor system.

Semantically, an automaton may perform two types of transitions. Delay transitions correspond to the execution of running tasks with highest priority (or earliest deadline) and idling for the other tasks waiting to run. Discrete transitions corresponds to the arrivals of new task instances.

We represent the values of clocks as functions (i.e. clock assignments) from $C$ to the non-negative reals $\mathbb{R}_{\geq 0}$. We denote by $\mathcal{V}$ the set of clock assignments for $C$. Naturally, a semantic state of an automaton is a triple $(l, u, q)$ where $l$ is the current location, $u \in \mathcal{V}$ denotes the current values of clocks, and $q$ is the current task queue. We assume that the task queue takes the form: $[P_1(c_0, d_0), P_2(c_1, d_1)...P_n(c_n, d_n)]$ where $P_i(c_i, d_i)$ denotes a released instance of task type $P_i$ with remaining computing time $c_i$ and relative deadline $d_i$.

Assume that there are a fixed number of processors running the released task instances according to a certain scheduling strategy $\text{Sch}$ e.g. FPS (fixed priority scheduling) or EDF (earliest deadline first) which sorts the task queue whenever new tasks arrives according to task parameters e.g. deadlines. 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 with $t$ time units. Thus the delay transition will decrease the computing time of the first task with $t$. If its computation time becomes 0, the task should be removed from the queue. Moreover, all deadlines in the queue will be decreased by $t$ (time has progressed by $t$). To summarize the above intuition, we introduce the following functions on task queues:

- $\text{Sch}$ is a sorting function for task queues (or lists), that may change the ordering of the queue elements only. For example, $\text{EDF}([P(3.1, 10), Q(4, 5.3)]) = [Q(4, 5.3), P(3.1, 10)]$. We call such sorting functions a scheduling strategy that may be preemptive or non-preemptive $^3$.

- $\text{Run}$ is a function which given a real number $t$ and a task queue $q$ returns

---

$^3$A non-preemptive strategy will never change the position of the first element of a queue and a preemptive strategy may change the ordering of task types only, but never change the ordering of task instances of the same type.
the resulted queue after $t$ time units of execution according to available resources. For simplicity, we assume that only one processor is available. Then the meaning of $\text{Run}(q,t)$ should be obvious and it can be defined inductively as follows: $\text{Run}(q,0) = q$, $\text{Run}([P_1(c_0,d_0), P_1(c_1,d_1) \ldots P_n(c_n,d_n)],t) = \text{Run}([P_1(c_1,d_1 - c_0) \ldots P_n(c_n,d_n - c_0)],t - c_0)$ if $c_0 \leq t$ and $\text{Run}([P_1(c_0,d_0) \ldots P_n(c_n,d_n)],t) = [P_1(c_0 - t,d_0 - t) \ldots P_n(c_n,d_n - t)]$ if $c_0 > t$. For example, let $q = [Q(4,5), P(3,10)]$. Then $\text{Run}(q,6) = [P(1,4)]$ in which the first task is finished and the second has been executed for 2 time units.

We use $u \models g$ to denote that the clock assignment $u$ satisfies the constraint $g$. For $t \in \mathbb{R}_{\geq 0}$, we use $u + t$ to denote the clock assignment which maps each clock $x$ to the value $u(x) + t$, and $u[r \mapsto 0]$ for $r \subseteq C$, to denote the clock assignment which maps each clock in $r$ to 0 and agrees with $u$ for the other clocks (i.e. $C \setminus r$). Now we are ready to present the operational semantics for extended timed automata by transition rules:

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

\begin{itemize}
  \item $(l, u, q) \xrightarrow{a_{\text{Sch}}}(m, u[r \mapsto 0], \text{Sch}(M(m) :: q))$ if $t \xrightarrow{\text{Run}} m$ and $u \models g$
  \item $(l, u, q) \xrightarrow{t_{\text{Sch}}}(l, u + t, \text{Run}(q,t))$ if $(u + t) \models I(l)$
\end{itemize}

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

Note that the transition rules are parameterized by $\text{Sch}$ (scheduling strategy), and $\text{Run}$ (function representing the available computing resources). According to the transition rules, the task queue is growing with action transitions and shrinking with delay transitions. Multiple copies (instances) of the same task type may appear in the queue.

Whenever it is understood from the context, we shall omit $\text{Sch}$ from the transition relation. Consider the automaton in Figure 2(b). Assume that preemptive earliest deadline first (EDF) is used to schedule the task queue. Then the automaton with initial state $(m_1, [x = 0], [Q_1(1, 2)])$ may demonstrate the following sequence of typical transitions:

---

4 The semantics may be extended to multi-processor setting by modifying the function $\text{Run}$ according the number of processors available.
Definition 3 We shall write $(m_1, [x = 0], [Q_1(1, 2)])$
\[ \frac{1}{\cdot} (m_1, [x = 1], [Q_1(0, 1)]) = (m_1, [x = 1], [])\]
\[ \frac{0.5}{\cdot} (m_1, [x = 10.5], [])\]
\[ \frac{a_1}{\cdot} (m_1, [x = 0], [P_1(4, 20)])\]
\[ \frac{0.5}{\cdot} (m_1, [x = 0.5], [P_1(3.5, 19.5)])\]
\[ \frac{b_2}{\cdot} (m_2, [x = 0.5], [Q_2(1, 4), P_1(3.5, 19.5)])\]
\[ \frac{0.3}{\cdot} (m_2, [x = 0.8], [Q_2(0.7, 3.7), P_1(3.5, 19.2)])\]
\[ \frac{a_2}{\cdot} (m_2, [x = 0], [Q_2(0.7, 3.7), P_2(2, 10), P_1(3.5, 19.2)])\]
\[ \frac{b_1}{\cdot} (m_1, [x = 0], [Q_2(0.7, 3.7), Q_1(1, 2), P_2(2, 10), P_1(3.5, 19.2)])\]
\[ \frac{10}{\cdot} (m_1, [x = 10], [])\]
\[ \ldots \]

This is only a partial behaviour of the automaton. A question of interest is whether it can perform a sequence of transitions leading to a state where the task queue is non-schedulable.

3 Schedulability Analysis

In this section we study verification problems related to the model presented in the previous section. First, we have the same notion of reachability as for timed automata.

Definition 3 We shall write $(l, u, q) \rightarrow (l', u', q')$ if $(l, u, q) \xrightarrow{a} (l', u', q')$ for an action $a$ or $(l, u, q) \xrightarrow{t} (l', u', q')$ for a delay $t$. For an automaton with initial state $(l_0, u_0, q_0)$, $(l, u, q)$ is reachable iff $(l_0, u_0, q_0) \rightarrow^* (l, u, q)$.

In general, the task queue is unbounded though the constraints of a given automaton may restrict the possibility of reaching states with infinitely many different task queues. This makes the analysis of automata more difficult. However, for certain analysis, e.g. verification of safety properties that are not related to the task queue, we may only be interested in the reachability of locations. A nice property of our extension is that the location reachability problem can be checked by the same technique as for timed automata [HNSY94, YPD94]. So we may view the original timed automaton (without task assignment) as an abstraction of its extended version preserving location reachability. The existing model checking tools such as [Yov97, LPY97] can be applied directly to verify the abstract models.
Timed Automata with Asynchronous Processes

But if properties related to the task queue are of interests, we need to develop new verification techniques. One of the most interesting properties of extended automata related to the task queue is schedulability.

According to the transition rules, the task queue is growing with action transitions and shrinking with delay transitions. Multiple copies (instances) of the same task type may appear in the queue. To illustrate the changing size of the task queue, we consider again the automaton shown in Figure 1 in Section 1. In location $l_1$, it can never generate more than 5 instances of $P$ due to the constraint $y \leq 40$, namely the number of copies may be bounded by the clock constraints. This is illustrated by the following transitions:

$$(l_0, [x = 0, y = 0], []) \xrightarrow{a} (l_1, [x = 0, y = 0], [P(2, 10)])$$

$$\frac{2.5}{a} (l_1, [x = 2.5, y = 2.5], [P(0, 7.5)]) = (l_1, [x = 2.5, y = 2.5], [])$$

$$\frac{7.5}{a} (l_1, [x = 10, y = 10], [])$$

$$\frac{a}{a} (l_1, [x = 0, y = 10], [P(2, 10)])$$

$$\ldots$$

$$\frac{3.4}{a} (l_1, [x = 3.4, y = 43.4], [P(0, 6.6)]) = (l_1, [x = 3.4, y = 43.4], [])$$

In fact, the queue size in location $l_1$ is bounded by 1. But in location $l_2$, an infinite number of copies of $Q$ may be released in zero time because there is no constraint on the $b$-transition, which demonstrates the so-called zeno behaviour:

$$(l_1, [x = 3.4, y = 43.4], []) \xrightarrow{b} (l_2, [x = 3.4, y = 43.4], [Q(4, 8)])$$

$$\frac{b}{b} (l_2, [x = 3.4, y = 43.4], [Q(4, 8), Q(4, 8)])$$

$$\frac{b}{b} (l_2, [x = 3.4, y = 43.4], [Q(4, 8), Q(4, 8), Q(4, 8)])$$

$$\ldots$$

But note that after more than two copies of $Q$, the queue will be non-schedulable.

We notice that zeno-behaviour will correspond to non-schedulability in our setting which is a nice property of the model. We shall see that non-schedulability

---

In fact, in location $l_1$, the size of the queue will be bounded by 1 because every released instance will be computed before the next release that will not arrive within 10 time units due to the constraint $x \geq 10$.

According to the optimal scheduling strategy EDF, no scheduling strategy will be able to schedule the queue $[Q(4, 8), Q(4, 8), Q(4, 8), Q(4, 8)]$ to meet the deadline for the last instance of $Q$. 

---
can be checked by reachability analysis. However zeno-freeness does not necessarily imply schedulability.

Now we formalize the notion of schedulability.

**Definition 4** *(Schedulability)* A state \( (l, u, q) \) with \( q = [P_1(c_1, d_1), \ldots P_n(c_n, d_n)] \) is a failure denoted \( (l, u, \text{Error}) \) if there exists \( i \) such that \( c_i \geq 0 \) and \( d_i < 0 \), that is, a task failed in meeting its deadline. Naturally an automaton \( A \) with initial state \( (l_0, u_0, q_0) \) is non-schedulable with \( \text{Sch} \) iff \( (l_0, u_0, q_0) \rightarrow_{\text{Sch}}^*(l, u, \text{Error}) \) for some \( l \) and \( u \). Otherwise, we say that \( A \) is schedulable with \( \text{Sch} \). More generally, we say that \( A \) is schedulable iff there exists a scheduling strategy \( \text{Sch} \) with which \( A \) is schedulable.

The schedulability of a state may be checked by the standard test. We say that \( (l, u, q) \) is schedulable with \( \text{Sch} \) if \( \text{Sch}(q) = [P_1(c_1, d_1), \ldots P_n(c_n, d_n)] \) and \( \sum_{i \leq k} c_i \leq d_k \) for all \( k \leq n \). Alternatively, an automaton is schedulable with \( \text{Sch} \) if all its reachable states are schedulable with \( \text{Sch} \).

Checking schedulability of a state is a trivial task according to the definition. But checking the relative schedulability of an automaton with respects to a given scheduling strategy is not easy, and checking the general schedulability (equivalent to finding a scheduling strategy to schedule the automaton) is even more difficult.

Fortunately the queues of all schedulable states of an automaton are bounded. First note that a task instance that has been started can not be preempted by another instance of the same task type. This means that there is only one instance of each task type in the queue whose computing time can be a real number and it can be arbitrarily small. Thus the number of instances of each task type \( P \in \mathcal{P} \), in a schedulable queue is bounded by \( \lceil D(P) / C(P) \rceil \) and the size of schedulable queues is bounded by \( \sum_{P \in \mathcal{P}} \lceil D(P) / C(P) \rceil \).

We will code schedulability checking problems as reachability problems. First, we consider the case of non-preemptive scheduling to introduce the problems. We have the following positive result.

**Theorem 1** The problem of checking schedulability relative to non-preemptive scheduling strategy for extended timed automata is decidable.

**Proof** A detailed proof is given in [EWY98]. We sketch the proof idea here. It is to code the given scheduling strategy as a timed automaton (called the scheduler) denoted \( E(\text{Sch}) \) which uses clocks to remember computing times and relative deadlines for released tasks. The scheduler automaton is constructed as follows: Whenever a task instance \( P_i \) is released by an event \( \text{release}_i \), a clock
$d_i$ is reset to 0. Whenever a task is started to run, a clock $c$ is reset to 0. Whenever the constraint $d_i = D(P_i)$ is satisfied, and $P_i$ is not finished, an error-state (non-schedulable) should be reached. We also need to transform the original automaton $A$ to $E(A)$ to synchronize with the scheduler that $P_i$ is released whenever a location, say $l$ to which $P_i$ is associated, is reached. This is done simply by replacing actions labelled on transitions leading to $l$ with release$_i$. Finally we construct the product automaton $E(Sch) || E(A)$ in which both $E(Sch)$ and $E(A)$ can only synchronize on identical action symbols namely release$_i$’s. It can be proved that if an error-state of the product automaton is reachable, the original extended timed automaton is non-schedulable.

For preemptive scheduling strategies, it has been conjectured that the schedulability checking problem is undecidable. The reason is that if we use the same ideas as for non-preemptive scheduling to encode a preemptive scheduling strategy, we must use stop-watches (or integrators) to add up computing times for suspended tasks. It appears that the computation model behind preemptive scheduling is stop-watch automata for which it is known that the reachability problem is undecidable. Surprisingly this conjecture is wrong.

**Theorem 2** The problem of checking schedulability relative to a preemptive scheduling strategy for extended timed automata is decidable.

The rest of this paper will be devoted to the proof of this theorem. It follows from Lemma 3, 4, and 5 established in the following section. Before we go further, we state a more general result that follows from the above theorem.

**Theorem 3** The problem of checking schedulability for extended timed automata is decidable.

From scheduling theory [But97], we know that the preemptive version of Earliest Deadline First scheduling (EDF) is optimal in the sense that if a task queue is non-schedulable with EDF, it can not be schedulable with any other scheduling strategy (preemptive or non-preemptive). Thus, the general schedulability checking problem is equivalent to the relative schedulability checking with respects to EDF.

## 4 Decidability and Proofs

We shall encode the schedulability checking problem as a reachability problem. For the case of non-preemptive scheduling, the expressive power of timed
automata is enough. For preemptive scheduling, we need a more expressive model.

4.1 Timed Automata with Subtraction

Definition 5 A timed automaton with subtraction is a timed automaton in which clocks may be updated by subtraction in the form $x := x - C$ in addition to reset of the form $x := 0$, where $C$ is a natural number.

This is the so-called updatable automata [BDFP00]. It is known that the reachability problem for this class of automata is undecidable. However, for the following class of suspension automata, location reachability is decidable.

Definition 6 (Bounded Timed Automata with Subtraction) A timed automaton is bounded iff for all its reachable states $(l, u, q)$, there is a maximal constant $C_x$ for each clock $x$ such that

1. $u(x) \geq 0$ for all clocks $x$, i.e. clock values should not be negative and
2. $u(x) \leq C_x$ if $l \xrightarrow{g(x)} l'$ for some $l'$ and $C$ such that $g(u)$ and $(x := x - C') \in v$.

In general, it may be difficult to compute the maximal constants from the syntax of an automaton. But we shall see that we can compute the constants for our encoding of scheduling problems.

Because subtractions on clocks are performed only within a bounded area, the region equivalence is preserved by the operation. We adopt the standard definition due to Alur and Dill [AD94].

Definition 7 (Region Equivalence denoted $\sim$) For a clock $x \in \mathcal{C}$, let $C_x$ be a constant (the ceiling of clock $x$). For a real number $t$, let $\{t\}$ denote the fractional part of $t$, and $\lfloor t \rfloor$ denote its integer part. For clock assignments $u, v \in \mathcal{V}$, $u, v$ are region-equivalent denote $u \sim v$ iff

1. for each clock $x$, either $\lfloor u(x) \rfloor = \lfloor v(x) \rfloor$ or $u(x) > C_x$ and $v(x) > C_x$ and
2. for all clocks $x, y$ if $u(x) \leq C_x$ and $u(y) \leq C_y$ then
   (a) $\{u(x)\} = 0$ iff $\{v(x)\} = 0$ and
   (b) $\{u(x)\} \leq \{u(y)\}$ iff $\{v(x)\} \leq \{v(y)\}$
It is known that region equivalence is preserved by the delay (addition) and reset. In the following, we establish that region equivalence is also preserved by subtraction for clocks that are bounded as defined in Definition 6. For a clock assignment \( u \), let \( u(x - C) \) denote the assignment: \( u(x - C)(x) = u(x) - C \) and \( u(x - C)(y) = u(y) \) for \( y \neq x \).

![Figure 3: Region equivalence preserved by subtraction when clocks are bounded.](image)

**Lemma 1** Let \( u, v \in \mathcal{V} \). Then \( u \sim v \) implies

1. \( u + t \sim v + t \) for a positive real number \( t \), and
2. \( u[x \mapsto 0] \sim v[x \mapsto 0] \) for a clock \( x \) and
3. \( u(x - C) \sim v(x - C) \) for all natural numbers \( C \) such that \( C \leq u(x) \leq C \).

**Proof** The proof for the first two items can be found in the literature e.g. [LW97].

We check the third according to the definition of region equivalence. It is illustrated in Figure 3 that the equivalence is preserved by subtraction in the bounded area. Assume that \( u \sim v \), and for each clock \( x \) we also have \( C \leq u(x) \leq C \). Because \( u \sim v \), and \( C \leq u(x) \), we have \( C \leq v(x) \). Then we have \( u(x - C)(y) \geq 0 \) and \( v(x - C)(y) \geq 0 \) for any clock \( y \). Now we have two cases to check:

1. If \( \lfloor u(x) \rfloor = \lfloor v(x) \rfloor \), obviously we have \( \lfloor u(x) - C \rfloor = \lfloor v(x) - C \rfloor \) for a natural number \( C \) and then \( \lfloor u(x - C) \rfloor = \lfloor v(x - C) \rfloor \) by definition.
2. Note that the subtraction operation on clocks does not change the fractional parts of clock values. Therefore for all clocks \( y \) and \( z \), we have:

   (a) \( \{u(x - C)(y)\} = 0 \) iff \( \{v(x - C)(y)\} = 0 \) and
(b) \( \{u(x - C)(y)\} \leq \{u(x - C)(z)\} \) iff \( \{v(x - C)(y)\} \leq \{v(x - C)(z)\} \)

In fact, region equivalence over clock assignments induces a bisimulation over reachable states of automata, which can be used to partition the whole state space as a finite number of equivalence classes.

**Lemma 2** Assume a bounded timed automaton with subtraction, a location \( l \) and clock assignments \( u \) and \( v \). Then \( u \sim v \) implies that

1. whenever \( (l, u) \rightarrow (l', u') \) then \( (l, v) \rightarrow (l', v') \) for some \( v' \) s.t. \( u' \sim v' \) and
2. whenever \( (l, v) \rightarrow (l', v') \) then \( (l, u) \rightarrow (l', u') \) for some \( u' \) s.t. \( u' \sim v' \).

**Proof** It follows from Lemma 1.

The above lemma essentially states that if \( u \sim v \) then \( (l, u) \) and \( (l, v) \) are bisimilar, which implies the following result.

**Lemma 3** The location reachability problem for bounded timed automata with subtraction, whose clocks are bounded with known maximal constants is decidable.

**Proof** Because each clock of the automaton is bounded by a maximal constant, it follows from lemma 2 that for each location \( l \), there is a finite number of equivalence classes of states which are equivalent in the sense that they will reach the same equivalence classes of states. Because the number of locations of an automaton is finite, the whole state space of an automaton can be partitioned into finite number of such equivalence classes.

### 4.2 Encoding of Schedulability as Reachability

Assume an automaton \( A \) extended with tasks, and a preemptive scheduling strategy \( \text{Sch} \). The aim is to check if \( A \) is schedulable with \( \text{Sch} \). As for the case of non-preemptive scheduling (Theorem 1), we construct \( E(A) \) and \( E(\text{Sch}) \), and check a pre-defined error-state in the product automaton of the two. The construction is illustrated in figure 4.

\( E(A) \) is constructed as a timed automaton which is exactly the same as for the non-preemptive case (Theorem 1) and \( E(\text{Sch}) \) will be constructed as a timed automaton with subtraction.
We introduce some notation. Let $C(i)$ and $D(i)$ stand for the worst case execution time and relative deadline respectively for each task type $P_i$. We use $P_{ij}$ to denote the $j$th instance of task type $P_i$.

For each task instance $P_{ij}$, we have the following state variables: $\text{status}(i,j)$ initialized to free. Let $\text{status}(i,j) = \text{running}$ stand for that $P_{ij}$ is executing on the processor, $\text{preempted}$ for that $P_{ij}$ is started but not running, and $\text{released}$ for that $P_{ij}$ is released, but not started yet. We use $\text{status}(i,j) = \text{free}$ to denote that $P_{ij}$ is not released yet or position $(i,j)$ of the task queue is free.

According to the definition of scheduling strategy, for all $i$, there should be only one $j$ such that $\text{status}(i,j) = \text{preempted}$ (only one instance of the same task type is started), and for all $i, j$, there should be only one pair $(k,l)$ such that $\text{status}(k,l) = \text{running}$ (only one is running for a one-processor system).

We need two clocks for each task instance:

1. $c(i,j)$ (a computing clock) is used to remember the accumulated computing time since $P_{ij}$ was started (when Run$(i,j)$ became true) \footnote{In fact, for each task type, we need only one clock for computing time because only one instance of the same task type may be started.}, and subtracted with $C(k)$ when the running task, say $P_{kl}$, is finished if it was preempted after it was started.

2. $d(i,j)$ (a deadline clock) is used to remember the deadline and reset to 0 when $P_{ij}$ is released.

We use a triple $(c(i,j), d(i,j), \text{status}(i,j))$ to represent each task instance, and the task queue will contain such triples. We use $q$ to denote the task queue. Note
that the maximal number of instances of $P_i$ appearing in a schedulable queue is $[D(i)/C(i)]$. We have a bound on the size of queue as claimed earlier, which is $\sum_{P_i \in \mathcal{P}} [D(i)/C(i)]$. We shall say that queue is empty denoted $\text{empty}(q)$ if $\text{status}(i,j) = \text{free}$ for all $i,j$.

For a given scheduling strategy $\text{Sch}$, we use the predicate $\text{Run}(m,n)$ to denote that task instance $P_{mn}$ is scheduled to run according to $\text{Sch}$. For a given $\text{Sch}$, it can be coded as a constraint over the state variables. For example, for EDF, $\text{Run}(m,n)$ is the conjunction of the following constraints:

1. $d(k,l) \leq D(k)$ for all $k,l$ such that $\text{status}(k,l) \neq \text{free}$: no deadline is violated yet
2. $\text{status}(m,n) \neq \text{free}$: $P_{mn}$ is released or preempted
3. $D(m) - d(m,n) \leq D(i) - d(i,j)$ for all $(i,j)$: $P_{mn}$ has the shortest deadline

$E(\text{Sch})$ contains three type of locations: $\text{Idle}$, $\text{Running}$ and $\text{Error}$ with $\text{Running}$ being parameterized with $(i,j)$ representing the running task instance.

1. $\text{Idle}$ denotes that the task queue is empty.
2. $\text{Running}(i,j)$ denotes that task instance $P_{ij}$ is running, that is, $\text{status}(i,j) = \text{running}$. We have an invariant for each $\text{Running}(i,j)$: $c(i,j) \leq C(i)$ and $d(i,j) \leq D(i)$.
3. $\text{Error}$ denotes that the task queues are non-schedulable with $\text{Sch}$.

There are five types of edges labelled as follows:

1. $\text{Idle}$ to $\text{Running}(i,j)$: there is an edge labelled by
   - guard: none
   - action: $\text{release}_i$
   - reset: $c(i,j) := 0, d(i,j) := 0$, and $\text{status}(i,j) := \text{running}$

2. $\text{Running}(i,j)$ to $\text{Idle}$: there is only one edge labelled with
   - guard: $\text{empty}(q)$ that is, $\text{status}(i,j) = \text{free}$ for all $i,j$ (all positions are free).
   - action: none
   - reset: none

3. $\text{Running}(i,j)$ to $\text{Running}(m,n)$: there are two types of edges.
(a) The running task $P_{ij}$ is finished, and $P_{mn}$ is scheduled to run by Run$(m, n)$. There are two cases:

i. $P_{mn}$ was preempted earlier:
   - guard: $c(i, j) = C(i)$, status$(m, n) = \text{preempted}$ and Run$(m, n)$
   - action: none
   - reset: status$(i, j) := \text{free}$, $\{c(k, l) := c(k, l) - C(i)\}$, status$(k, l) = \text{preempted}$, and status$(m, n) := \text{running}$

ii. $P_{mn}$ was released, but never preempted (not started yet):
   - guard: $c(i, j) = C(i)$, status$(m, n) = \text{released}$ and Run$(m, n)$
   - action: none
   - reset: status$(i, j) := \text{free}$, $\{c(k, l) := c(k, l) - C(i)\}$, status$(k, l) = \text{preempted}$, $c(m, n) := 0$, $d(m, n) := 0$, and status$(m, n) := \text{running}$

(b) A new task $P_{mn}$ is released, which preempts the running task $P_{ij}$:
   - guard: status$(m, n) = \text{free}$, and Run$(m, n)$
   - action: released$_m$
   - reset: status$(m, n) := \text{running}$, $c(m, n) := 0$, $d(m, n) := 0$, and status$(i, j) := \text{preempted}$

4. Running$(i, j)$ to Running$(i, j)$. There is only one edge representing the case when a new task is released, and the running task $P_{ij}$ will continue to run:
   - guard: status$(k, l) = \text{free}$, and Run$(i, j)$
   - action: released$_k$
   - reset: status$(k, l) := \text{released}$ and $d(k, l) := 0$

5. Running$(i, j)$ to Error: for each pair $(k, l)$, there is an edge labelled by $d(k, l) > D(k)$ and status$(k, l) \neq \text{free}$ meaning that the task $P_{kl}$ which is released (or preempted) fails in meeting its deadline.
The third step of the encoding is to construct the product automaton $E(Sch) \parallel E(A)$ in which both $E(Sch)$ and $E(A)$ can only synchronize on identical action symbols. Now we show that the product automaton is bounded.

**Lemma 4** All clocks of $E(Sch)$ in $E(Sch) \parallel E(A)$ are bounded and non-negative.

**Proof** All computing clocks $c(k,l)$ are bounded by $D(k)$. This is due to the fact that all edges labelled with a subtraction is guarded by the constraint: $\text{Run}(m,n)$ which requires $d(k,l) \leq D(k)$ (no deadline is violated in order to stay in Running). $d(k,l) \leq D(k)$ implies that $c(k,l) \leq D(k)$ because $d(k,l)$ is always reset to zero before (or at the same time) $c(k,l)$ when a new instance of $P_k$ is released. Thus $c(k,l)$ is bounded.

Secondly, the only possibility for a computing clock, say $c(k,l)$ for task $P_{kl}$, to become negative is by subtractions. But a subtraction is done on $c(k,l)$ only when a task, say $P_{ij}$, is finished i.e. $c(i,j) = C(i)$ holds. As $c(i,j)$ is reset to zero before $c(k,l)$ is reset to zero ($\text{status}(k,l) = \text{preempted}$ implying that $P_{kl}$ was released and preempted earlier), we have $c(i,j) \geq c(k,l)$ implying that $c(i,j) - C(k) \geq 0$. Thus all clocks are non-negative. \hfill \square

Now we have the correctness lemma for our encoding. Assume, without losing generality, that the initial task queue of an automaton is empty.

**Lemma 5** Let $A$ be an extended timed automaton and $Sch$ a scheduling strategy. Assume that $(l_0, u_0, q_0)$ and $(\langle l_0, \text{idle} \rangle, u_0 \cup v_0)$ are the initial states of $A$ and the product automaton $E(A) \parallel E(Sch)$ respectively where $l_0$ is the initial location of $A$, $u_0$ and $v_0$ are clock assignments assigning all clocks with 0 and $q_0$ is the empty task queue. Then

1. For all $l$ and $u$: $(l_0, u_0, q_0) \rightarrow^* (l, u, \text{Error})$ implies $(\langle l_0, \text{idle} \rangle, u_0 \cup v_0) \rightarrow^* (\langle l, \text{Error} \rangle, u \cup v)$ for some $v$.
2. For all $l$, $u$ and $v$: $(\langle l_0, \text{idle} \rangle, u_0 \cup v_0) \rightarrow^* (\langle l, \text{Error} \rangle, u \cup v)$ implies $(l_0, u_0, q_0) \rightarrow^* (l, u, \text{Error})$

**Proof** We use $c_{ij}$ and $d_{ij}$ to denote the remaining computing time and relative deadline respectively for a task instance $P_{ij}$ in the ready queue. Whenever it is understood, we shall write $c_{ij}, d_{ij}, \text{status}(P_{ij})$ to denote $v(c_{ij}), v(d_{ij}), v(\text{status}(P_{ij}))$. We show the existence of a bisimulation mapping between the states of $A$ and $E(A) \parallel E(Sch)$.

Let $S_1 = \{(l, u, q), (\langle l, \text{idle} \rangle, (u \cup v)) \mid \text{empty}(q)\},$
\[ S_2 = \{(l, u, q), (\langle l, \text{Running}(m, n) \rangle, (u \cup v)) \mid \text{Cnd}_1 \land \text{Cnd}_2 \land \text{Cnd}_3 \land \text{Cnd}_4 \} \]

where

- \( \text{Cnd}_1 \equiv [d_{ij} \geq 0 \text{ and } d_{ij} = D(P_{ij}) - d(i, j) \text{ for all } i, j \text{ such that status}(P_{ij}) \neq \text{free}] \)
- \( \text{Cnd}_2 \equiv [c_{ij} \geq 0 \text{ and } c_{ij} = C(P_{ij}) - c(i, j) \text{ for all } i, j \text{ such that status}(P_{ij}) = \text{running}] \)
- \( \text{Cnd}_3 \equiv [c_{ij} = C(P_{ij}) - (c(i, j) - c(m, n)) \text{ for all } i, j \text{ such that status}(P_{ij}) = \text{preempted}] \)
- \( \text{Cnd}_4 \equiv [c_{ij} = C(P_{ij}) \text{ for all } i, j \text{ such that status}(P_{ij}) = \text{preempted}] \)

and \( S_3 = \{(l, u, q), (\langle l, \text{Error} \rangle, (u \cup v)) \mid d_{ij} \leq 0, d_{ij} \geq D(P_{ij}) \text{ for some } i, j \} \)

We establish that \( S = S_1 \cup S_2 \cup S_3 \) is a bisimulation. We prove only one direction that every transition of the first element of a pair in \( S \) is followed by a transition from the second element and the resulted pair remains in \( S \). The proof for the other direction (that is, \( S \) is symmetrical) is similar.

(S1) Assume \( (l, u, q), (\langle l, \text{Idle} \rangle, (u \cup v)) \in S_1 \), and \( (l, u, q) \rightarrow^{a} (l', u', \text{Sch}(M(l') :: q)) \). Assume further that this transition is induced by \( l \xrightarrow{g.a} l' \). Let \( M(l') = P_i \). Then \( u \models g \). Thus the product automaton can make the following action transition: \( (\langle l, \text{Idle} \rangle, (u \cup v)) \rightarrow^{a} (\langle l', \text{Running}(i, j) \rangle, (u' \cup v')) \) for some \( j \). \( P_{ij} \) is inserted into the queue, \( c_{ij} = C(P_{ij}), d_{ij} = D(P_{ij}) \), and the variables in \( v \) are updated as follows: \( d(i, j) := 0, c(i, j) := 0 \) and \( \text{status}(i, j) := \text{running} \). Now, \( P_{ij} \) is the only task instance in the queue, and its status is running. It is obvious that the four conditions defining \( S_2 \) are satisfied. Therefore \( (\langle l', u', \text{Sch}(M(l') :: q) \rangle, (\langle l', \text{Running}(i, j) \rangle, (u' \cup v')) \) \( \in S_2 \).

For the delay transitions of \( (l, u, q) \), assume \( (l, u, q) \rightarrow^{t} (l, u + t, \text{Run}(q, t)) \), where \( (u + t) \models I(l) \). Then the product automaton can make the following delay transition: \( (\langle l, \text{Idle} \rangle, (u \cup v)) \rightarrow^{t} (\langle l, \text{Idle} \rangle, (u + t \cup v + t)) \). Running the empty queue for \( t \) time units results in an empty queue, therefore \( (l, u + t, \text{Run}(q, t)), (\langle l, \text{Idle} \rangle, (u + t \cup v + t)) \) \( \in S_1 \).

(S2) Assume \( (l, u, q), (\langle l, \text{Running}(m, n) \rangle, (u \cup v)) \in S_2 \) and \( (l, u, q) \rightarrow^{a} (l', u', \text{Sch}(M(l') :: q)) \). Further assume that this transition is induced by \( l \xrightarrow{g.a} l' \) and \( u \models g \).
Let \( M'(l) = P_{ij} \) and \( \text{Hd}(q) = P_{nn} \) (i.e. \( \text{status}(P_{nn}) = \text{running} \)). Then the product automaton can make the following action transition: \((l, \text{Running}(m, n)), (u \cup v)) \xrightarrow{a} (l', \text{Running}(k, l)) \), \((u' \cup v')\) where \((k, l)\) can be either \((m, n)\) or \((i, j)\) depending on the deadlines of \(P_{nn}\) and \(P_{ij}\). If \(d_{ij} < d_{nn}\) then \(\text{status}(P_{nn}) = \text{preempted}, \text{status}(P_{ij}) = \text{running}, d(i, j) = 0, c(i, j) = 0\). Otherwise, \(\text{status}(P_{nn}) = \text{running}, \text{status}(P_{ij}) = \text{released}\) and \(d(i, j) = 0\). It is easy to check the four conditions defining \(S_2\) are satisfied. Thus \(((l', u', \text{Sch}(M'(l) :: q)), ((l, \text{Running}(k, l)), (u' \cup v')))) \in S_2\).

Now for the delay transitions, assume \((l, u, q) \xrightarrow{l}(l, u + t, \text{Run}(q, t))\). Then there are three situations for the resulted queue \(\text{Run}(q, t)\):

1. Run\((q, t)\) is non-empty (the first task is running): In case, \(t\) is less than the remaining computing time of the first task in the queue, and it is also less than all the relative deadlines in the queue, the scheduler automaton will not change the status of the running task \(P_{nn}\). We have surely \(((l, u + t, \text{Run}(q, t)), ((l, \text{Running}(m, n)), (u + t \cup v + t))) \in S_2\). When \(t\) is equal to the remaining computing time \(c_{nn}\) of the first task in the queue, and also less than all the relative deadlines in the queue, we have the resulted queue with \(P_{nn}\) removed and the scheduler automaton will move to \(\text{running}(k, l)\) in case the task \(P_{kl}\) is chosen to run. If \(\text{status}(P_{kl}) = \text{released}\) then \(c(k, l) = 0\), and for all tasks \(P_{ij}\) if \(\text{status}(P_{ij}) = \text{preempted}\) then \(c(i, j) = c(i, j) - C(n, m)\). The four conditions defining \(S_2\) remain to hold. Therefore \(((l, u + t, \text{Run}(q, t)), ((l, \text{Running}(k, l)), (u + t \cup v + t))) \in S_2\).

2. Run\((q, t)\) is empty (all tasks are completed): This is similar to the above case. It means that \(P_{nn}\) completes its execution, and no more task to run. The product automaton can make the following delay transition and then a discrete transition: \(((l, \text{Running}(m, n)), (u \cup v)) \xrightarrow{l} ((l, \text{Running}(m, n)), (u + t \cup v + t)) \xrightarrow{} ((l, \text{Idle}), (u + t \cup v + t))\). For all \(P_{ij}\) \(\text{status}(P_{ij}) = \text{free}\), therefore \(((l, \text{Idle}), (u + t \cup v + t)) \in S_1\). It is easy to see that \(((l, u + t, \text{Run}(q, t)), ((l, \text{Idle}), (u + t \cup v + t))) \in S_1\).

3. Run\((q, t)\) is non-schedulable: Assume \((l, u, q) \xrightarrow{l}(l, u + t, \text{Run}(q, t))\), where \((u + t) \models I(l), d_{kl} < t < c_{kl}\) for some \(P_{kl} \in q\) with \(d_{kl}\) being the shortest relative deadline in the queue. This transition results in a non-schedulable queue where \(P_{kl}\) just missed its deadline. Then the product automaton can make the following delay transition: \(((l, \text{Running}(m, n)), (u \cup v)) \xrightarrow{l} ((l, \text{Running}(m, n)), (u + t \cup v + t))\). We have \(d_{kl} - t < 0\) and \(c_{kl} - t > 0\). Then by the first condition in defining \(S_2\), we have
$$d_{kl} = D(P_{kl}) - d(k, l).$$ Then we have \(d_{kl} - t = D(P_{kl}) - (d(k, l) + t)\) and because \(d_{kl} - t < 0\), we have \(D(P_{kl}) < d(k, l) + t\), which enables the transition for the scheduler automaton to move to the error-state. That is, \((l, \text{Running}(m, n)), (u + t \cup v + t) \rightarrow (l, \text{Error}), (u + t \cup v + t))\).

As \(d_{kl} - t < 0\), we have \((l, u + t, \text{Run}(q, t)), (l, \text{Error}, (u + t \cup v + t))) \in S_3\).

(S3) This case is trivial. \(\square\)

The above lemma states that the schedulability analysis problem can be solved by reachability analysis for timed automata extended with subtraction. From Lemma 4, we know that \(E(\text{Sch})\) is bounded. Because the reachability problem is decidable due to Lemma 3, we complete the proof for our main result stated in Theorem 2.

5 Conclusions and Related Work

We have studied a model of timed systems, which unifies timed automata with the classic task models from scheduling theory. The model can be used to specify resource requirements and hard time constraints on computations, in addition to features offered by timed automata. It is general and expressive enough to describe concurrency and synchronization, and tasks which may be periodic, sporadic, preemptive and (or) non-preemptive. The classic notion of schedulability is naturally extended to automata model.

Our main technical contribution is the proof that the schedulability checking problem is decidable. The problem has been suspected to be undecidable due to the nature of preemptive scheduling. To our knowledge, this is the first decidability result for preemptive scheduling in dense-time models. Based on the proof, we have developed a symbolic schedulability checking algorithm using the DBM techniques extended with a subtraction operation. It has been implemented in a prototype tool [AFM+02]. We believe that our work is one step forward to bridge scheduling theory and automata-theoretic approaches to system modelling and analysis. A challenge is to make the results an applicable technique combined with classic methods such as rate monotonic scheduling. We need new algorithms and data structures to represent and manipulate the dynamic task queue consisting of time and resource constraints. As another direction of future work, we shall study the schedule synthesis problem. More
specifically given an automaton, it is desirable to characterize the set of schedu-
able traces accepted by the automaton.

Related work. Scheduling is a well-established area. Various analysis meth-
ods have been published in the literature. For systems restricted to periodic
tasks, algorithms such as rate monotonic scheduling are widely used and effi-
cient methods for schedulability checking exist, see e.g. [But97]. These tech-
niques can be used to handle non-periodic tasks. The standard way is to consider
non-periodic tasks as periodic using the estimated minimal inter-arrival times
as task periods. Clearly, the analysis based on such a task model would be pes-
simistic in many cases, e.g. a task set which is schedulable may be considered
as non-schedulable as the inter-arrival times of the tasks may vary over time,
that are not necessary minimal. Our work is more related to work on timed
systems and scheduling.

A nice work on relating classic scheduling theory to timed systems is the con-
troller synthesis approach [AGP+99, AGS00]. The idea is to achieve schedu-
lability by construction. A general framework to characterize scheduling con-
straints as invariants and synthesize scheduled systems by decomposition of
constraints is presented in [AGS00]. However, algorithmic aspects are not dis-
cussed in this work. Timed automata has been used to solve non-preemptive
scheduling problems mainly for job-shop scheduling[AM01, Feh99, HLP01].
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. There have been sev-
eral work e.g. [MV94, Cor94, CL00] on using stop-watch automata to model
preemptive scheduling problems. As the reachability analysis problem for stop-
watch automata is undecidable in general [ACH+95], there is no guarantee for
termination for the analysis without the assumption that task preemptions occur
only at integer points. The idea of subtractions on timers with integers, was first
proposed by McManis and Varaiya in [MV94]. In general, the class of timed
automata with subtractions is undecidable, which is shown in [BDFP00]. In
this paper, we have identified a decidable class of updatable automata, which
is precisely what we need to solve scheduling problems without assuming that
preemptions occur only at integer points.

Acknowledgement: Thanks to the anonymous referees for their insights and
constructive comments.
References


Paper B:

Schedulability Analysis Using Two Clocks

Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi.
Schedulability Analysis Using Two Clocks

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

Abstract. In this paper, we present an efficient algorithm for schedulability analysis for timed automata extended with real time tasks. More precisely, we show that for a fixed priority scheduling, the schedulability checking problem can be solved by reachability analysis on standard timed automata using only two extra clocks in addition to the clocks used to describe task arrival times. We extend the result to systems with data-dependent control, in which the release time of a task may depend on the time-point at which other tasks finish their execution. For the case when the execution times of tasks are constants, we show that the schedulability problem can be encoded as a reachability problem using $n + 1$ extra clocks, where $n$ is the number of tasks.

1 Introduction

In the area of real time scheduling methods such as rate monotonic scheduling are widely applied in the analysis of periodic tasks with deterministic behaviours. For non-periodic tasks with non-deterministic behaviours, there are no satisfactory procedures. In reality control tasks are often triggered by sporadic events coming from the environment. The common approach to analyse schedulability of such systems with non-periodic tasks is to consider the minimal inter-arrival time of a task as its period and then follow the ordinary technique used for periodic tasks. Obviously such an approximate method is quite pessimistic since the task control structures are not considered. A major advantage can be gained using timed automata to specify relaxed timing constraints on events and model other behavioural aspects such as concurrency and synchronization. In order to perform schedulability analysis with timed automata the model of Extended Timed Automata (ETA) has been suggested in [FPY02]. It unifies timed automata [AD94] with the classic task models from scheduling theory allowing to execute tasks asynchronously and specify hard time con-
constraints on computations. Furthermore, the problem of schedulability analysis for this model has been proven to be decidable for any scheduling policy and the algorithm for schedulability analysis was presented. It is based on translation of the schedulability problem into reachability for the decrementation automata [MV94]. Unfortunately, the number of clocks needed in the analysis is proportional to the maximal number of schedulable task instances associated with a model, which in many cases is huge. A remaining challenge is to make the result applicable for schedulability analysis of systems with non-uniformly recurring tasks that scale up to industrial systems.

In this paper we present an efficient algorithm for schedulability analysis of systems with relaxed timing constraints, which uses only two additional clocks. The algorithm also allows to compute the worst-case response time for non-periodic tasks. More precisely, we show that for a fixed priority scheduling strategy, the schedulability checking problem can be solved by reachability analysis on standard timed automata using only two extra clocks in addition to the clocks used in the original model to describe task arrival times. We shall extend the result to systems with data-dependent control, in which the timed automata and the tasks may read and update shared data variables i.e. Then the release time-point of a task may depend on the values of the shared variables, and hence on the time-point at which other tasks finish their execution. For the case when the execution times of tasks are constants, we show that the schedulability problem can be encoded as a reachability problem for timed automata using \( n + 1 \) extra clocks, where \( n \) is the number of tasks. For the case when the execution times of tasks are intervals, unfortunately the problem is undecidable [KY03]; we present a solution using over-approximation.

The rest of this paper is organized as follows: Section 2 describes the syntax and semantics of ETA and defines scheduling problems related to the model. In Section 3, we present the main result of this paper – an algorithm to perform schedulability analysis of systems with relaxed timing constraints. Section 4 is devoted to schedulability analysis of systems with fixed priorities and data-dependent control. In Section 5, we describe implementation issues and how to perform worst-case response time analysis. Section 6 concludes the paper with summary and related work.
2 Preliminaries

2.1 Timed Automata with Tasks

A timed automaton \[ \text{[AD94]} \] is a standard finite-state automaton extended with a finite collection of real-valued clocks. One can interpret timed automata as an abstract model of a running system that describes the possible events occurring during its execution. Those events must satisfy given timing constraints. To clarify how events, accepted by a timed automaton, should be handled or computed we extend timed automata with asynchronous processes \[ \text{[FPY02]} \], i.e. tasks triggered by events asynchronously. The idea is to associate each location of a timed automaton with an executable program called a task. We assume that the execution times and hard deadlines of the tasks are known\(^1\).

**Syntax.** Let \( \mathcal{P} \) ranged over by \( P, Q, R \), denote a finite set of task types. A task type may have different instances that are copies of the same program with different inputs. Each task \( P \) is characterized as a pair of natural numbers denoted \( P(C, D) \) with \( C \leq D \), where \( C \) is the execution time (or computation time) of \( P \) and \( D \) is the deadline for \( P \). The deadline \( D \) is relative, meaning that when task \( P \) is released, it should finish within \( D \) time units. We shall use \( C(P) \) and \( D(P) \) to denote the worst case execution time and relative deadline of \( P \) respectively.

As in timed automata, assume a finite set of alphabets \( \text{Act} \) for actions and a finite set of real-valued variables \( \mathcal{C} \) for clocks. We use \( a, b \) etc. to range over \( \text{Act} \) and \( x_1, x_2 \) etc. to range over \( \mathcal{C} \). We use \( \mathcal{B}(\mathcal{C}) \) ranged over by \( g \) 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. The elements of \( \mathcal{B}(\mathcal{C}) \) are called clock constraints.

**Definition 8** A timed automaton extended with tasks, over actions \( \text{Act} \), clocks \( \mathcal{C} \) and tasks \( \mathcal{P} \) is a tuple \( \langle N, I_0, E, I, M \rangle \) where

- \( \langle N, I_0, E, I \rangle \) is a timed automaton where
  - \( N \) is a finite set of locations ranged over by \( l, m, n \),
  - \( I_0 \in N \) is the initial location, and

---

\(^1\)Task may have other parameters such as fixed priority for scheduling and other resource requirements, e.g. memory requirement.
- $E \subseteq N \times \mathcal{B}(C) \times Act \times 2^C \times N$ is the set of edges.

- $I : N \mapsto \mathcal{B}(C)$ is a function assigning each location with a clock constraint (a location invariant).

- $M : N \mapsto \mathcal{P}$ is a partial function assigning locations with tasks $^2$.

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 is triggered, it will be put in a scheduling (or task) queue for execution (corresponding to the ready queue in operating systems).

**Operational Semantics.** Extended timed automata may perform two types of transitions just as standard timed automata. The difference is that delay transitions correspond to the execution of running tasks with highest priority and idling for the other tasks waiting to run. Discrete transitions corresponds to the arrival of new task instances.

We represent the values of clocks as functions (called clock assignments) from $C$ to the non-negative reals. A state of an automaton is a triple $(l, u, q)$ where $l$ is the current control location, $u$ the clock assignment, and $q$ is the current task queue. We assume that the task queue takes the form: $[P_1(c_1, d_1), \ldots, P_n(c_n, d_n)]$ where $P_i(c_i, d_i)$ denotes a released instance of task type $P_i$ with remaining computing time $c_i$ and relative deadline $d_i$.

A scheduling strategy $\text{Sch}$ e.g. FPS (fixed priority scheduling) or EDF (earliest deadline first) is a sorting function which changes the ordering of the task queue elements according to the task parameters. For example, $\text{EDF}([P(3.1, 10), Q(4, 5.3)]) = [Q(4, 5.3), P(3.1, 10)])$. We call such sorting functions scheduling strategies that may be preemptive or non-preemptive$^3$. Thus an action transition will result in a sorted queue including the tasks released by this transition. A delay transition with $c$ time units is to execute the task in the first position of the queue with $c$ time units. Thus the delay transition will decrease the computing time of the first task with $c$. If its computation time becomes 0, the

$^2$ 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.

$^3$ As in scheduling theory, we adopt the standard assumptions on scheduling strategies: A non-preemptive strategy will never change the position of the first element of a queue. A preemptive strategy may change the ordering of task types only, but never change the ordering of task instances of the same type.
We shall write Definition 10 notion of reachability as for ordinary timed automata. In this section we briefly review the verification problems of ETA. For more details, we refer the reader to [FPY02]. We first mention that we have the same

\begin{definition}
Given a scheduling strategy Sch^4, the semantics of an extended timed automaton \( \langle N, I_0, E, I, M \rangle \) with initial state \( (l_0, u_0, q_0) \) is a transition system defined by the following rules:

\begin{itemize}
  \item \((l, u, q) \xrightarrow{a} \text{Sch}(m, u[r \mapsto 0], \text{Sch}(M(m) :: q)) \) if \( l \xrightarrow{a,r} m \) and \( u \models g \)
  \item \((l, u, q) \xrightarrow{t} \text{Sch}(l, u + t, \text{Run}(q,t)) \) if \( (u + t) \models I(l) \)
\end{itemize}

where \( M(m) :: q \) denotes the queue with \( M(m) \) inserted in \( q \).
\end{definition}

### 2.2 Schedulability and Decidability

In this section we briefly review the verification problems of ETA. For more details, we refer the reader to [FPY02]. We first mention that we have the same notion of reachability as for ordinary timed automata.

\begin{definition}
We shall write \((l, u, q) \xrightarrow{a} (l', u', q') \) if \( (l, u, q) \xrightarrow{a} (l', u', q') \) for an action \( a \) or \((l, u, q) \xrightarrow{t} (l', u', q') \) for a delay \( t \). For an automaton with initial state \((l_0, u_0, q_0)\), \((l, u, q)\) is reachable iff \((l_0, u_0, q_0) \xrightarrow{a} (l, u, q)\).
\end{definition}

Note that the reachable state-space of an ETA is infinite not only because of the real-valued clocks, but also unbounded size of the task queue.

---

4Note that we fix \( \text{Run} \) to be the function that represents a one-processor system.
Definition 11 (Schedulability) A state \((l, u, q)\) where \(q = [P_1(c_1, d_1), \ldots, P_n(c_n, d_n)]\) is a failure denoted \((l, u, \text{Error})\) if there exists \(i\) such that \(c_i \geq 0\) and \(d_i < 0\), that is, a task failed in meeting its deadline. Naturally an automaton \(A\) with initial state \((l_0, u_0, q_0)\) is non-schedulable with \(\text{Sch}\) iff \((l_0, u_0, q_0)(\rightarrow_{\text{Sch}})^*(l, u, \text{Error})\) for some \(l\) and \(u\). Otherwise, we say that \(A\) is schedulable with \(\text{Sch}\). More generally, we say that \(A\) is schedulable iff there exists a scheduling strategy \(\text{Sch}\) with which \(A\) is schedulable.

The schedulability of a state may be checked by the standard schedulability test. We say that \((l, u, q)\) is schedulable with \(\text{Sch}\) if \(\text{Sch}(q) = [P_1(c_1, d_1)\ldots P_n(c_n, d_n)]\) and \((\sum_{i \leq k} c_i) \leq d_k\) for all \(k \leq n\). Alternatively, an automaton is schedulable with \(\text{Sch}\) if all its reachable states are schedulable with \(\text{Sch}\).

Theorem 4 The problem of checking schedulability for extended timed automata is decidable.

Proof The proof is given in [FPY02].

3 Main Result: Two Clocks Encoding

In this section we present the main result of this paper. It shows that for timed automata extended with tasks executed according to fixed priorities, the scheduling problem can be encoded into a reachability problem of ordinary timed automata using only two additional clocks.

Assume an ETA \(A\) and a fixed priority scheduling strategy \(\text{Sch}\). To solve the scheduling problem, for each \(P_i \in \mathcal{P}\) we construct \(E_i(\text{Sch})\) 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, task \(P_i\) of automaton \(A\) is not schedulable with \(\text{Sch}\). The check is performed in priority order for each task in \(\mathcal{P}\), starting with the task of highest priority.

Our analysis technique is inspired by Joseph and Pandya’s rate-monotonic analysis of periodic tasks [JP86], where the worst-case response time of each task is calculated as the sum of the task’s execution time, and the blockings imposed by other tasks. Similar to Joseph and Pandya, for each task type we check independently that it meets its deadline. However, the model of ETA gives rise to a more general scheduling problem than systems with periodic tasks only. As a result, we can not base our analysis on the existence of an a priori known worst-case scenario for a given task. Instead, it will be part of the analysis to find all situations in which a task may execute.
Figure 1: Task execution schemes for tasks $P_i$ and $P_j$ with $\text{Prio}(j) > \text{Prio}(i)$. The symbols $\uparrow$ and $\downarrow$ indicate release and completion of tasks, respectively.

To construct the $E(A)$, the automaton $A$ is annotated with distinct synchronization actions $\text{release}_i$ on all edges leading to locations labelled with the task name $P_i$. The actions will allow the scheduler to observe when tasks are released for execution in $A$. The rest of this section is devoted to show that $E_i(\text{Sch})$ can be constructed as a timed automaton using only two clocks.

**Theorem 5** Given a fixed priority scheduling strategy $\text{Sch}$, $E_i(\text{Sch})$ can be encoded as a timed automaton containing two clocks.

**Proof** Follows from Lemma 6 and 7 shown later in this section. \qed

In the encoding of $E_i(\text{Sch})$, we shall use $C(i)$, $D(i)$ and $\text{Prio}(i)$ to denote the worst-case execution time, the deadline, and the priority of task type $P_i$, respectively. $E_i(\text{Sch})$ uses the following variables:

- $d$ - a clock measuring the time since the analysed task instance of $P_i$ was released for execution,
- $c$ - a clock accumulating the time since the task queue last empty (or containing only tasks $P_k$ with $\text{Prio}(k) < \text{Prio}(i)$).
- $r$ - a data variable used to sum up the time needed to complete all tasks released since the processor was last idle (i.e. not executing instances of $P_i$ and all higher priority tasks).

The clock $d$ is reset when the analysis of a task instance begins, and will be used to check it completes before its deadline. The clock $c$ is used to compute the time point when the analysed task instance of $P_i$ completes. The variable $r$ will be assigned so that $P_i$ completes when $c = r$. Fig.1 shows in two Gantt charts
Figure 2: Encoding of schedulability problem.

how the variables are used in $E_i(Sch)$. In Fig.1(a) task $P_i$ executes immediately
but is preempted by $P_j$. In Fig.1(b) task $P_i$ is released when task $P_j$ is already
executing. Note how the clocks c and d are reset, and variable r is updated in
the two scenarios so that task $P_i$ is completed when the condition $c = r$ is satisfied.
Note also that the deadline of $P_i$ is reached when $d = D(i)$ (as d is reset when
$P_i$ is released for execution).

The encoding of $E_i(Sch)$ is shown in Fig.2. Intuitively, the locations have the
following interpretations:

- **Idle$_i$** - denotes a situation where no task $P_j$ with $\text{Prio}(j) \geq \text{Prio}(i)$ is
  being executed (or ready to be executed).

- **Check$_i$** - an instance of task type $P_i$ is currently ready for execution
  (possibly executing) and is being analysed for schedulability.

- **Busy$_i$** - a task of type $P_j$ with priority $\text{Prio}(j) \geq \text{Prio}(i)$ is currently
  executing.

- **Error$_i$** - the analysed task queue is not schedulable with $Sch$.

The analysis of an instance of $P_i$ starts when a transition from $\text{Idle}_i$ or $\text{Busy}_i$
to $\text{Check}_i$ is taken. The transitions in $E_i(Sch)$ have the following intuitive
interpretations:
- **Idle** - is (re-)entered when the task instance being checked in **Check**, or a sequence of tasks arrived in **Busy**, has finished execution. In both cases the enabling condition \( c=r \) ensures that the location is reached when all tasks \( P_j \) with \( \text{Pri}(j) \geq \text{Pri}(i) \) have finished their executions.

- **Busy** - the ingoing transitions to **Busy** are taken when a task \( P_j \) such that \( \text{Pri}(j) \geq \text{Pri}(i) \) is released. The additional self-loop, is taken to decrement both \( c \) and \( r \) with the constant value \( C \). This does not change the truth-value of any of the guards in which \( c \) and \( r \) appear, as the values are always compared to each other.

- **Check** - transitions entering **Check** from **Idle** or **Busy** are taken when a task instance of \( P_i \) is (non-deterministically) chosen for checking. Self-loops in **Check** are taken to update \( r \) at the release of higher-priority tasks. New instances of \( P_i \) in **Check** are ignored as they are considered by the non-deterministic choice in location **Busy**.

- **Error** - is reached when the analysed task instance reaches its deadline (encoded \( d = D(i) \)) before completion (encoded \( c < r \)). In addition, **Error** is entered if the set of released tasks is guaranteed to be non-schedulable (encoded \( r > R_i^{max} \), the value of \( R_i^{max} \) is discussed in below).

In addition to these transitions, in Fig 2 we have omitted self-loops in all locations, which synchronize with \( E(A) \) whenever a task of priority lower than \( \text{Pri}(i) \) is released. They can be ignored as these tasks do not affect the response time of \( P_i \).

The constant \( C^{max} \) can be any value greater than 0. We use \( C^{max} = \max_i(C(i)) \). To find a value for \( R_i^{max} \), we need the result of the previous analysis steps. Recall that the analysis of all \( P_i \in P \) is performed in priority order, starting with the highest priority. Thus, when \( P_i \) is analysed we can find the maximum value assigned to \( r \) in the previous analysis steps. Let \( r^{max} \) denote this value. Recall that \( r - c \) is always the time remaining until the released tasks complete their executions (except in location **Idle** and **Error** where \( r \) is not updated). For the set of released tasks to be schedulable we have that \( r - c < r^{max} + D(i) \). It follows that \( r < r^{max} + D(i) + C^{max} \) since \( c \leq C^{max} \). We set the constant \( R_i^{max} = r^{max} + D(i) + C^{max} \) and use \( r > R_i^{max} \) to detect non-schedulable tasks sets in \( E(\text{Sch}) \).

The last step of the encoding is to construct the product automata \( E(A) \| E_i(\text{Sch}) \) for each \( P_i \in P \), and check by reachability analysis that location **Error** is
not reachable in the product automaton. We now show that \( E(A) \parallel E_i(\text{Sch}) \) is bounded.

**Lemma 6** The clocks \( c \) and \( d \) and the data variable \( r \) of \( E_i(\text{Sch}) \) in \( E(A) \parallel E_i(\text{Sch}) \) are bounded.

**Proof** The clocks \( d \) and \( c \) are bounded by the constants \( D(i) \) and \( C^{\text{max}} \) respectively. The data variable \( r \) is bounded by \( R^{\text{max}} + \max\{ j : \text{Pri}(j) > \text{Pri}(i) \} \ C(j) \).

\( \square \)

**Lemma 7** Let \( A \) be an extended timed automaton and \( \text{Sch} \) a fixed-priority scheduling strategy. Assume that \((l_0, u_0, q_0)\) and \((l_0, \text{Idle}_i, v_0)\) are the initial states of \( A \) and the product automaton \( E(A) \parallel E_i(\text{Sch}) \) respectively where \( l_0 \) is the initial location of \( A \), \( u_0 \) and \( v_0 \) are assigning all clocks with 0 and \( q_0 \) is the empty task queue. Then the following holds:

\[
(l_0, u_0, q_0)(\rightarrow)^*(l, u, \text{Error}) \text{ iff } ((l_0, \text{Idle}_i), v_0)(\rightarrow)^*(l, \text{Error}_i, v)
\]

for some \( l, u, v, i \).

**Proof** We assume that the task queue takes the form: \([P_1...P_n]\) where \( P_i \) denotes a released instance of task type \( i \) with \( r \) remaining computing time \( c(P_i) \) and relative deadline \( d(P_i) \), and the variable (and clock) assignment \( v \) in the product automaton takes the form \((u, v)\), where \( u \) is an assignment for \( A \) and \( v \) is an assignment for \( E_i(\text{Sch}) \). Whenever it is understood, we shall write \( c, d, r \) to denote \( c(P_i), v(P_i), v(r) \).

We show the existence of simulations between the states of \( A \) and \( E(A) \parallel E_i(\text{Sch}) \). Let \( S_1 = \{(l, u, q), ((l, \text{Idle}_i), (u, v)) | \text{empty}(q)\} \), \( S_2 = \{(l, u, q), (l, \text{Busy}_i), (u, v)\} \), \( S_3 = \{(l, u, q), (l, \text{Check}_i), (u, v)\} \) \( (\sum_{i=1}^{k} c(P_i)) = r - c \), \( S_4 = \{(l, u, q), (l, \text{Check}_i), (u, v)\} \) \( (\sum_{i=1}^{k} c(P_i)) > d(P_k) \text{ and } r - c > d(P_k) \), \( S_5 = \{(l, u, q), (l, \text{Error}_i), (u, v)\} \) \( c(P_k) > 0, d(P_k) = 0 \) and \( S = S_1 \cup S_2 \cup S_3 \cup S_4 \cup S_5 \).

We prove that \( S \) and \( S^{-1} \) are simulations. First we prove that \( S \) is a simulation.

Assume that \( ((l, u, q), ((l, \text{Idle}_i), (u, v))) \in S_1 \). We consider the two types of transitions:

- \( \text{(Action)} \) Assume \( (l, u, q) \xrightarrow{a} (l', u', \text{Sch}(M(P') :: q)) \). Further assume that this transition is induced by \( l \xrightarrow{0.a} l' \) and \( u \models g \). Then the product automaton can also make the following \( a \)-transitions: \( ((l, \text{Idle}_i), (u, v)) \xrightarrow{a} \).
(\langle \ell', \text{Busy}_i \rangle, (u', v[c := 0, r := C(M(\ell'))]))). Here a new task is inserted into the queue, and the variable r is set to the WCET of the new task. Therefore, \((\sum_{i=1}^{k} c(P_i)) = C(M(\ell')) = r - c\), and then \((\ell', u', \text{Sch}(M(\ell') \downarrow q)), (\ell', \text{Busy}_i), (u', v[c := 0, r := C(M(\ell'))]) \in S_2\). Alternatively, if \(M(\ell') = P_k\), the product automaton can also have the transition: \((\ell, \text{Idle}_i), (u, v)) \rightarrow (\langle \ell', \text{Check}_i \rangle, (u', v[c := 0, r := C(P_k), d := 0]))\). Here \(P_k\) is inserted into the queue, and the variable \(r\) is set to the \(C(P_k)\). Therefore, \((\sum_{i=1}^{k} c(P_i)) = C(P_k) = r - c\), and then \((\ell', u', \text{Sch}(M(\ell') \downarrow q)), (\ell', \text{Check}_i), (u', v[c := 0, r := C(P_k), d := 0])\) \(\in S_3\).

- (Delay) Assume \((l, u, q) \rightarrow_{sch} (l, u+t, \text{Run}(q, t)), (u+t) \models I(l)\). Then the product automaton can make the following delay transition: \((\ell, \text{Idle}_i), (u, c, r, d)) \rightarrow (\ell, \text{Idle}_i), (u + t, c + t, r, d + t))\). Running the empty queue for \(t\) time units results in an empty queue, therefore \((\ell, \text{Idle}_i), (u + t, v + t)) \in S_1\).

The rest of the proof that \(S\) as well as \(S^{-1}\) is a simulation is similar. \(\Box\)

Thus, we have shown that the scheduling problem can be solved by a reachability problem for timed automata, and from Lemma 6 we know that the reachability problem is bounded. This completes the proof of Theorem 5.

4 Analysing Data-Dependent Control

In this section we extend the result of the previous section to handle extended time automata in which the tasks may use (read and update) data variables, shared between the tasks and the automata. This results in a model with data-dependent control in the sense that the behaviour of the control automaton, and the release time-point of tasks may depend on the values of the shared variables, and hence on the time-points at which other tasks complete their executions. We first present the model of ETA extended with data variables [AFP’02].

4.1 Extended Timed Automata with Data Variables

Syntax. Assume a set of variables \(\mathcal{D}\) ranged over by \(u\), which takes their values from finite data domains, and are updated by assignments in the form \(u := \mathcal{E}\), where \(\mathcal{E}\) is a mathematical expression. We use \(\mathcal{R}\) to denote the set of all possible assignments. A task \(P\) is now characterized by a triple \(P(C, D, R)\),
where $C$ and $D$ are the execution time and the deadline as usual, and $R \subseteq \mathcal{R}$ is a set of assignments. We use $R(P)$ to denote the set of assignments of $P$, and we assume that a task assigns the variables according to $R(P)$ by the end of its execution.

The data variables assigned by tasks may also be updated and tested (or read) by the extended timed automata. Let $A = \mathcal{R} \cup \{x := 0 \mid x \in \mathcal{C}\}$ be the set of updates. We use $r$ to stand for a subset of $A$. To read and test the values of the data variables, let $B(D)$ be a set of predicates over $D$. Let $B = B(D) \cup B(C)$ be ranged over by $g$ called guards.

Unfortunately the analysis of data-dependent control structures cannot be based on the WCET of tasks only for the obvious reason that if a task updates shared variables by the end of its execution (before WCET) it can trigger releases of the other tasks and lead to a negative schedulability result. This means that a system may be schedulable when all the tasks actually consume the WCET and it may not be schedulable when some of the tasks consume less than the WCET.

In the following, we present a solution for the schedulability analysis problem for the case when the execution times, denoted $C(P)$ for task $P$, are constants. The case when the execution times are intervals i.e. best and worst execution times is considered later.

**Operational Semantics.** To define the semantics, we use valuations to denote the values of variables. 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 $\sigma$. For a non-negative real number $t$, we use $\sigma + t$ to denote the valuation which updates each clock $x$ with $\sigma(x) + t$, and $\sigma[r]$ to denote the valuation which maps each variable $\alpha$ to the value of $E$ if $\alpha := E \in r$ (note that $E$ is zero if $\alpha$ is a clock) and agrees with $\sigma$ for the other variables. We are now ready to present the semantics of extended timed automata with data variables by the following rules:

- $(l, \sigma, q) \xrightarrow{a} \text{Sch}(m, \sigma[r], \text{Sch}(M(m) :: q))$ if $l \xrightarrow{g, \sigma[r]} m$ and $\sigma \models g$
- $(l, \sigma, q) \xrightarrow{t} \text{Sch}(l, \sigma + t, \text{Run}(q, t))$ if $(\sigma + t) \models I(l)$ and $C(\text{Hd}(q, t)) > t$
- $(l, \sigma, q) \xrightarrow{t} \text{Sch}(l, (\sigma[A(\text{Hd}(q))] + t, \text{Run}(q, t))$ if $(\sigma + t) \models I(l)$ and $C(\text{Hd}(q)) = t$

where $M(m) :: q$ denotes the queue with $M(m)$ inserted in $q$ and $\text{Hd}(q)$ denotes the first element of $q$. 
4.2 Schedulability Analysis

As in the previous section, we shall encode the ETA $A$ and the fixed-priority scheduling strategy $Sch$ into timed automata and check for reachability of predefined error states. The encoding $E(A)$ is the same as in the previous section. However, the encoding of $Sch$ will be different with data-dependent control, as the result of the schedulability analysis depends on the data-variables that may be updated whenever a task completes its execution. In the rest of this section we describe how to construct $E(Sch)$:

**Theorem 6** For an extended timed automaton $A$ with data variables, and a fixed priority scheduling strategy $Sch$, $E(Sch)$ can be constructed as timed automaton containing $n + 1$ clocks, where $n$ is a number of task types used in $A$.

**Proof** Follows from Lemma 8 and 9 shown later in this section. \hfill $\Box$
The construction of $E(Sch)$ is illustrated in Fig.3. It consists of two parallel automata: $E_{SP}(Sch)$ - encoding the scheduling policy (containing $n$ clocks), and $E_{DC}$ - encoding a generic deadline checker (containing one clock). As in the previous section, the two scheduling automata (in this case both $E_{SP}(Sch)$ and $E_{DC}$) synchronize with $E(A)$ on the action release when an instance of task $P_i$ is released. In addition, $E_{SP}(Sch)$ and $E_{DC}$ synchronize on finished whenever an instance of $P_i$ finishes its execution.

**Encoding of Scheduling Policy $E_{SP}(Sch)$**. We first introduce some notation. Let $P_{ij}$ denote instance $j$ of task $P_i$. For each $P_{ij}, E_{SP}(Sch)$ has a state variable $\text{status}(i,j)$ that is initially set to free. Let $\text{status}(i,j) = \text{running}$ denote that $P_{ij}$ is executing on the processor, $\text{preempted}$ that $P_{ij}$ is started but not running, and $\text{released}$ that $P_{ij}$ is released but not yet started. We use $\text{status}(i,j) = \text{free}$ to denote that $P_{ij}$ is not released yet. Note that for all $(i,j)$ there can be only one $j$ such that $\text{status}(i,j) = \text{preempted}$ (i.e. only one instance of the same task type is started), and for all $(i,j)$ there can only be one pair $(k,l)$ such that $\text{status}(k,l) = \text{running}$ (i.e. only one task is running in a one-processor system).

For each task type $P_i$ we use three variables:

- $c_i$ - clock measuring the time passed since $P_i$ started its execution. We reset $c_i$ whenever an instance of $P_i$ is started.

- $r_i$ - data variable accumulating the response time of $P_i$ from the moment it starts to execute. $r_i$ is set to $C(i)$ when an instance of $P_i$ is started, and updated to $r_i + C(j)$ when a higher-priority task $P_j$ is released.

- $n_i$ - data variable keeping track of the number of $P_i$ currently released.

In Fig. 4, we show how the above variables are used in $E_{SP}(Sch)$. At time point $x$ state variable status has the values $\text{status}(1,1) = \text{running}, \text{status}(2,1) = \text{preempted}, \text{status}(2,2) = \text{released}$, and $\text{status}(3,1) = \text{released}$.

To represent each task instance in $E_{SP}(Sch)$ we use a triple $(c_i, r_i, \text{status}(i,j))$, and the task queue $q$ will contain such triples. Note that the maximal number of instances of $P_i$ appearing in a schedulable queue is $[D(i)/C(i)]$. Thus, the size of the queue is bounded to $\sum_{P_i \in P} [D(i)/C(i)]$. We shall say that queue is empty, denoted $\text{empty}(q)$, if $\text{status}(i,j) = \text{free}$ for all $(i,j)$.

For a given scheduling strategy Sch, we use the predicate $\text{Run}(m,n)$ to denote that task instance $P_{mn}$ is scheduled to run according to Sch. For a given fixed priority scheduling policy Sch, it can be coded as a constraint over the state
Schedulability Analysis Using Two Clocks

\[ n_1 = 1, \quad c_1 = 0, \quad r_1 = C_1 \]
\[ n_2 = 2, \quad c_2 = r_2, \quad r_2 = C_2 \]
\[ n_3 = 1, \quad c_3 = 0, \quad r_3 = C_3 \]

Figure 4: Task execution scheme where Prior(1) > Prior(2) > Prior(3).

variables. For example, for deadline-monotonic scheduling\(^5\), \( \text{Run}(m, n) \) is the conjunction of the following constraints:

- \( r_k \leq D(k) \) for all \( k, l \) such that \( \text{status}(k, l) \neq \text{free} \): all response time integers are less than deadlines
- \( \text{status}(m, n) \neq \text{free} \): \( P_{mn} \) is released or preempted
- \( D(m) \leq D(i) \) for all \( i \): \( P_m \) has the highest priority

We use \( \text{Run}(m) \) to denote that a task instance of \( P_m \) is scheduled to run according to Sch. The predicate \( \text{finished}(m, n) \) denotes that \( P_{mn} \) has finished its execution. We define \( \text{finished}(m, n) \) to be \((c_m = r_m) \land (\text{status}(m, n) \neq \text{free})\). Finally, we use \( \text{non-schedulable}(q) \) to denote that the queue \( q \) is non-schedulable in a sense that there exists a pair \((i, j)\) for which \( r_1 > D(i) \) and \( \text{status}(i, j) \neq \text{free} \).

The automaton \( E_{\text{Sch}}(\text{Sch}) \) contains three type of locations: \text{Idle}, \text{Running}_i, and \text{Error}. Note that \( \text{Running}_i \) is parameterized with \( i \) representing the running task type. Location \text{Idle} denotes that the task queue is empty. \text{Running}_i \) denotes that task instance of type \( P_i \) is running, that is, for some \( j \) \( \text{status}(i, j) = \text{running} \). For each \( \text{Running}_i \) we have the location invariant \( c_i \leq r_i \). \text{Error} denotes that the task queue is non-schedulable with \( \text{Sch} \). There are five types of edges labelled as follows:

\(^5\)In deadline-monotonic scheduling, task priorities are assigned according to deadlines, such that Prior(i) > Prior(j) iff D(i) < D(j).
1. Idle to Running; edges labelled with action release, and reset \( r, c_i := C(i), n_i := 1, \text{status}(i, j) := \text{running} \).

2. Running\(_i\) to Idle: edges labelled with guard \( \text{empty}(q) \) and reset \( n_i := 0, R(P_i) \).

3. Running\(_i\) to Running\(_m\): two types of edges:
   
   (a) the running task \( P_{ij} \) is finished and \( P_{mn} \) is scheduled to run by \( \text{Run}(m, n) \). There are two cases:
      
      i. \( P_{mn} \) was preempted earlier: encoded by guard \( \text{finished}(i, j) \land \text{status}(m, n) = \text{preempted} \land \text{Run}(m, n), \) action finished\(_i\), and reset \( \{ \text{status}(i, j) := \text{free}, n_i := n_i - 1, \text{status}(m, n) := \text{running}, R(P_i) \} \)

      ii. \( P_{mn} \) was released, but never preempted (not started yet): encoded by guard \( \text{finished}(i, j) \land \text{status}(m, n) = \text{released} \land \text{Run}(m, n), \) action finished\(_i\), and reset \( \{ \text{status}(i, j) := \text{free}, n_i := n_i - 1, r_m := C(m), c_m := 0, \text{status}(m, n) := \text{running}, R(P_i) \} \)

   (b) a new task \( P_{mn} \) is released, which preempts the running task \( P_{ij} \): encoded by guard \( \text{status}(m, n) = \text{free} \land \text{Run}(m, n), \) action \( \text{released}_m \), and reset \( \{ \text{status}(m, n) := \text{running}, n_m := n_m + 1, r_m := C(m), c_m := 0, \text{status}(i, j) := \text{preempted} \} \cup \{ r_k := r_k + C(m) \mid \text{status}(k, l) = \text{preempted} \} \) (we increment the response times of all preempted tasks by the execution time of the released higher-priority task).

4. Running\(_i\) to Running; edges representing the case when a task release does not preempt the running task \( P_{ij} \): encoded by guard \( \text{status}(k, l) = \text{free} \land \text{Run}(i, j), \) action \( \text{released}_k \), and reset \( \{ \text{status}(k, l) := \text{released}, n_k := n_k + 1 \} \cup \{ r_k := r_k + C(m) \mid \text{status}(k, l) = \text{preempted} \}

5. Running\(_i\) to Error: an edge labelled by the guard \( \text{nonschedulable}(q) \).

**Encoding of Deadline Checker** \( E_{DC} \). It is similar to the encoding of \( E_i(\text{Sch}) \) described in the previous section, in the sense that it checks for deadline violations of each task instance independently. The clock \( d \) is used in \( E_{DC} \) to measure the time since the analysed instance of \( P_i \) was released for execution. \( E_{DC} \) also uses a data variable, named \( \text{instance} \). From location \( \text{Idle} \) the automaton non-deterministically starts to analyse a task on the edge to \( \text{Check}_i \), at which
clock $d$ is reset and instance is set to $n_i$, i.e. the current number of released instances of task $P_i$. In Check, instance is decremented whenever an instance of $P_i$ finishes its execution. The analysed task finishes when instance $= 1$ and the location Idle is reentered. However, if $d$ is greater than $D(i)$, the task failed to meet its deadline and the location Error is reached.

The next step of the encoding is to construct the product automaton $E(A) || E_{SP}(Sch) || E_{DC}$ in which the automata can only synchronize on identical action symbols. We now show that the product automaton is bounded.

**Lemma 8** The clocks $c_t$ and $d$, and the data variables $r_i$ and $n_i$ of $E_{SP}(Sch) || E_{DC}$ in $E(A) || E_{SP}(Sch) || E_{DC}$ are bounded.

**Proof** First note that the integers $r_k$ are bounded by $D(k) + \max_i(C(i))$ due to the fact that all edges incrementing $r_k$ (by some $C(i)$) are guarded by the constraint $\text{Run}(m, n)$ requiring $r_k \leq D(k)$. The bound for $n_k$ is $\lfloor D(k)/C(k) \rfloor$. The clocks $d$ and $c_k$ are bounded by $\max_i(D(i))$ and $r_k$, respectively. □

**Lemma 9** Let $A$ be an extended timed automaton and Sch a fixed-priority scheduling strategy. Assume that $(I_0, u_0, q_0)$ and $(\langle l_0, \text{Idle}, \text{Idle} \rangle, v_0)$ are the initial states of $A$ and the product automaton $E(A) || E_{SP}(Sch) || E_{DC}$ respectively where $I_0$ is the initial location of $A$, $u_0$ and $v_0$ are clock assignments assigning all clocks with 0 and $q_0$ is the empty task queue. Then the following holds:

$$(I_0, u_0, q_0)(\longrightarrow)^*(\langle l, u, \text{Error} \rangle)_{iff}(\langle l_0, \text{Idle}, \text{Idle} \rangle, v_0)(\longrightarrow)^*(\langle l, m, n \rangle, v)$$

for some $u, v$ and $l, m, n$ where either $m$ or $n$ is Error.

**Proof** The lemma can be proved by establishing the simulation between the states of $A$ and $E(A) || E_{SP}(Sch) || E_{DC}$. It is similar to the proof for Lemma 7. □

**Interval execution times.** We may extend the model to handle tasks whose execution time is an interval of the form $[C_{tB}, C_{tW}]$, where $C_{tB}$ and $C_{tW}$ denote the best and worst case execution times of task $P_i$ respectively. Unfortunately the schedulability checking problem for such systems is undecidable [KY03].

In the following, we present an analysis method using over-approximation. The idea is to modify the scheduler automaton so that the variables are updated as shown in Figure 5. As before, we use $e_i$ to keep track of the accumulated execution time of $P_i$, and a pair of data variables $r_{iB}$ and $r_{iW}$ to sum up the
best and the worst completion time of \( P_1 \). Obviously \( r_{iB} \) and \( r_{iW} \) should be set to \( C_{1B} \) and \( C_{1W} \) respectively when task \( P_1 \) starts to execute. Observe that each preemption will enlarge the difference \( r_{0W} - r_{0B} \) for the preempted task \( P_0 \) with lower priority by the difference \( C_{1W} - C_{1B} \) for the finishing task \( P_1 \) with higher priority. Accordingly, we modify the scheduler automaton as follows: on edges labeled \text{Finished} \_j from locations \text{Running}(P_j) \ the guard should be \( r_{jB} \leq c_j \leq r_{jW} \) and variable updating should be \( r_{kB} := r_{kB} + C_{jB}, r_{kW} := r_{kW} + C_{jW} \) for all \( k \) such that \text{status}(P_k) = \text{preempted}. The rest of the scheduler automaton remains the same as before.

It is easy to see that the presented algorithm is an over-approximation. For example, consider the system shown in Figure 6. Tasks \( P_L, P_M \) and \( P_H \) have priorities low, medium and high respectively. Task \( P_L \) starts executing at time 0, and is being preempted by the task \( P_M \) at time 1. \( P_M \) has execution times in the interval \([2,5]\) and by the end of its execution it sets the boolean variable \text{flag} \ to true, which is initially set to false. If \( P_M \) completes its execution before 3 time units, it can trigger the higher priority task \( P_H \) that also preempts the execution of \( P_L \). Obviously, the worst-case response time of \( P_L \) is 13, which means that it finishes its execution within its deadline. However, the algorithm will compute the worst-case response time of \( P_L \) as a sum of worst-case exe-
Execution times of $P_L$, $P_M$ and $P_H$, which equals 15 and exceeds the deadline of $P_L$.

5 Implementation

The algorithm described in Section 3 has been implemented in TIMES, a tool for modelling and schedulability analysis of embedded real-time systems [AFM+02]. The modelling language of TIMES is ETA as described in Section 4.1 of this paper. The tool currently supports simulation, schedulability analysis, checking of safety and liveness properties, and synthesis of executable C-code [AFP+02].

A screen-shot of the TIMES tool analysing a simple control system is shown in Fig.7. In the main window, a control automaton is displayed. To the left, a table shows the specified task parameters. The task parameters currently supported are: behaviour ($B$)$^6$, priority ($P$), computation time ($C$), deadline ($D$), and period ($T$). The system in Fig.7 consist of tasks with fixed priorities and data-independent control. In this case, the schedulability analysis is performed as described in Section 3.

The system analysed in Fig.7 is a simple controller of a motor, periodically polling a sensor and at requests providing a user with sensor statistics. In the initial location, an instance of task ReadSensor is released. The controller waits 10 time units for a user to push the button. If the button is not pushed, the controller releases the two tasks AnalyzeData and ActuateMotor. If the button is pushed when the controller operates in its initial location, an instance of task ComputeStatistics is released for execution, and the controller waits 16 time units before releasing task ReadSensor again.

The system has been analysed with two algorithms implemented in the TIMES tool. An implementation based on the original decidability result described in [FPY02] consumes 2.7 seconds, whereas an implementation of the algorithm presented in Section 3 of this paper terminates in 0.1 seconds on the same machine$^7$. Thus, the time consumption is reduced significantly for this system.

In addition to schedulability analysis, it is possible to adjust the algorithms.

$^6$The behaviour field is one of: periodic ($P$), sporadic ($S$), or controllable ($C$) if the time points for the task release are specified by an automaton.

$^7$The measurements were made on a Sun Ultra-80 running SunOS 5.7. The UNIX program time was used to measure the time consumption.
Figure 7: The TIMES tool performing schedulability analysis.

presented in this paper, and implemented in TIMES, to compute the worst-case response time of tasks in a schedulable system. In general, the response time of a task is a non-integer value. We take the worst-case response time to be the lowest integer value greater or equal to the longest response time of a task. The worst-case response time of task $P_i$ can be obtained from the maximum value appearing in the upper bound on the clock $d$ in the symbolic states generated during the schedulability analysis of task $P_i$ (i.e. in the reachability analysis). In Fig. 7 the numbers in the task table column $D$ are the worst-case response times of the tasks in the system. Thus, if any of them is decreased, the system becomes non-schedulable.

6 Conclusions and Related Work

In this paper we have shown that for fixed priority scheduling strategy, the schedulability checking problem of timed automata extended with tasks can be solved by reachability analysis on standard timed automata using only two
additional clocks. We have also shown how to extend the result to systems with data-dependent control, i.e. systems in which the release time-points of a task may depend on the values of shared variables, and hence on the time-point at which other tasks finish their execution. In this case the encoding into reachability problem for standard timed automata uses $n + 1$ clocks, where $n$ is the number of tasks types. Both these encodings use much fewer clocks than the analysis suggested in the original decidability result, and we believe that we have found the optimal solutions to the problems. The presented encodings seem to suggest that the general schedulability problem of ETA can be transformed into a reachability problem of standard timed automata, instead of timed automata with subtraction operation on clocks. This is indeed the case, but the number of clocks used in the standard timed automaton will be the same as in the encoding using timed automata with subtraction.

The schedulability checking algorithms described in this paper have been implemented in the TIMES tool. An experiment shows that the new techniques substantially reduces the computation time needed to analyse an example systems with fixed priority scheduling strategy.

**Related work.** Well established scheduling theory and scheduling algorithms are described in various publications. In the area of real time scheduling methods such as rate monotonic scheduling [But97] are widely applied in the analysis of systems with deterministic behaviours restricted to periodic tasks. However, for systems with non-periodic tasks and non-deterministic behaviours, there are still no satisfactory procedures to perform schedulability analysis. One of the approaches to achieve schedulability is based on controller synthesis paradigm [AGS02, AGP+99]. The methodology described in [AGS02] relies on the idea that one can build schedulable system successively restricting guards of the controllable actions in its model in an appropriate way. However, concepts related to implementation description are not addressed in this work. In the area of non-preemptive scheduling timed automata has been used mainly for job-shop scheduling [Feh99, AM01, HLP01]. The idea is to get schedules out of traces produced during reachability analysis for pre-defined locations specifying scheduling goal. Stop-watch automata have been used to solve preemptive scheduling problem [MV94, Cor94, CL00]. But since reachability analysis problem for this class of automata is undecidable in general there is no guarantee of termination for the analysis without the assumption that task preemptions occur only at integer points. Tools have been developed to support the design
and analysis of embedded systems with tasks. Examples hereof include TAXYS [CPP+01] and TIMES [AFM+02].

References


Paper C:

Handling Precedence and Resource Constraints in Schedulability Analysis Using Timed Automata

Elena Fersman.
Handling Precedence and Resource Constraints in Schedulability Analysis Using Timed Automata

Elena Fersman
Uppsala University
Department of Information Technology
P.O. Box 337, S-751 05 Uppsala, Sweden
Email: elenaf@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 generic task model. 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, describ-
Handling Precedence and Resource Constraints

82

Handling Precedence and Resource Constraints

ing 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 [LL73], and Joseph and Pandya’s response-time analysis [JP86]. In the past years, these classic works have been extended to deal with more complex constraints e.g. offset analysis [PH98] and unfolding [BLMSv98] for precedence constraints and priority ceiling protocols [SRL90, RSL98] 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. [AGP99, EWY98, AGS00, FPY02, FMPY03]. 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 [FPY02], 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 [FMPY03], 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 [FMPY03] 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 schedulability 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 [But97], 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
Handling Precedence and Resource Constraints

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 [GL95]. 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: $\{(T_i, OP_i, A_i)\}$ where
Handling Precedence and Resource Constraints

- $OP_i$ is the semaphore operation to be performed on a semaphore. Let $OP_i = p(s)$ mean locking of $s$ and $OP_i = v(s)$ mean unlocking of $s$.
- $T_i$ is the computation time for performing $A_i$.
- $A_i$ is a sequence of assignments updating the set of variables associated with semaphores $s$ when $OP_i = v(s)$, 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. $\sum T_i$. 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 $(p(s), T_i, A_i)$ should be followed by $(v(s), T_j, A_j)$ 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 [But97]. 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 [But97] and periodic tasks with release jitters [Tin93] 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 [EWY98] and [FPY02], 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 $C$ for clocks. We use $a, b, \tau$ etc. to range over $Act$, where $\tau$ denotes a distinct internal action, and $x_1, x_2$ etc to range over $C$. We use $B(C)$ 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 $x_i, x_j \in C$ are clocks, $\sim \in \{\leq, <, =, \geq, >\}$, and $C, D$ are natural numbers. We use $B_I(C)$ for the subset of $B(C)$ where all atomic constraints are of the form $x \prec C$ and $\prec \in \{<, \leq\}$. The elements of $B(C)$ are called clock constraints.

**Definition 12** A timed automaton extended with tasks, over actions $Act$, clocks $C$ and tasks $P$ is a tuple $\langle N, l_0, E, I, M \rangle$ where

- $\langle N, l_0, E, I \rangle$ is a timed automaton
- $M : N \leftrightarrow P$ 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

$^1$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.
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 [LPY95]) using the notion of synchronisation function [HK89]. For example, consider the parallel composition \( A \parallel B \) of \( A \) and \( B \) over the same set of actions \( Act \). The set of nodes of \( A \parallel B \) 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 [HK89], the action set of the parallel composition will be \( Act \) and thus the task assignment function for \( A \parallel B \) is the same as for \( A \) and \( B \).

### 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 \( \sigma \). Naturally, a semantic state of an automaton is a triple \( (l, \sigma, q) \) where \( l \) is the current control location, \( \sigma \) 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, \( \text{EDF}([P(3.1, 10), Q(4, 5.3)]) = [Q(4, 5.3), P(3.1, 10)] \). 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 \( \text{Run}(q, t) \) should be obvious and it can be defined inductively. For example, let \( q = [Q(4, 5), P(3, 10)] \). Then \( \text{Run}(q, 6) = [P(1, 4)] \) in which the first task is finished and the second has been executed for 2 time units.

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

### Timing constraints

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

**Definition 13** Given a scheduling strategy \( \text{Sch}^2 \), the semantics of an extended timed automaton \( \langle N, I_0, E, I, M \rangle \) with initial state \( (I_0, u_0, q_0) \) is a transition system defined by the following rules:

- \((l, u, q) \xrightarrow{a} \text{sch}(m, u[r \rightarrow 0], \text{Sch}(M(m) \Downarrow q)) \) if \( l \xrightarrow{g, a, r} m \) and \( u \models g \)

- \((l, u, q) \xrightarrow{t} \text{sch}(l, u + t, \text{Run}(q, t)) \) if \( (u + t) \models I(l) \)

where \( M(m) \Downarrow q \) denotes the queue with \( M(m) \) inserted in \( q \).\(^2\)

\(^2\)Note that we fix \( \text{Run} \) to be the function that represents a one-processor system.
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 \( SC_{CB/CR/CW} \) 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 \( G_{ij} \) for every pair \( P_i, P_j \) such that \( P_i \rightarrow P_j \), initialized with 0. \( G_{ij} = 1 \) means that \( P_i \) is completed and enables \( P_j \). Thus when \( P_i \) is completed, \( G_{ij} \) is switched to 1 and 0 again when \( P_j \) is completed. We shall use the predicate \( \text{ready}(P) \) to denote that \( P_i \) is ready to run according to the precedence constraints. Naturally \( \text{ready}(P_i) \) holds if \( G_{k,i} = 0 \) for no \( P_k \). Note that when \( \text{ready}(P_i) \) holds, there may be several instances of \( P_i \), that are released. In that case, \( \text{ready}(P_i) \) 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:

\[
(l, \sigma, q) \xrightarrow{t} \text{Sch}(l, \sigma + t, \text{Run}(q, t)) \text{ if } (\sigma + t) \models I(l) \text{ and } t < c(\text{Hd}(\text{ready}(q)))
\]

\[
(l, \sigma, q) \xrightarrow{t} \text{Sch}(l, \sigma' + t, \text{Run}(q, t)) \text{ if } (\sigma + t) \models I(l) \text{ and } t = c(\text{Hd}(\text{ready}(q)))
\]

where

- \( \text{ready}(q) \) is the sub-queue of \( q \) containing all elements \( P \) of \( q \) such that \( \text{ready}(P) \).
- \( \sigma' \) is the new valuation defined as follows:

\[
\sigma'(G_{ij}) = 1 \text{ if } P_i = \text{Hd}(\text{ready}(q)), \quad \sigma'(G_{ij}) = 0 \text{ if } P_j = \text{Hd}(\text{ready}(q))
\]

and \( \sigma'(G_{ij}) = \sigma(G_{ij}) \) otherwise.
In case the task queue is empty we interpret the conditions $t < c(Hd(q))$ and $t = c(Hd(q))$ as true. Hence the transition rules become equal and correspond to a delay transition in ordinary timed automata.

**Resource constraints** To resolve resource constraints and avoid unbounded priority inversion, various resource access protocols have been developed (see e.g. [But97]). 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) [SRL90].

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) [RSL98] 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 $BC$ be the highest priority. Assume that for each task $C8$, we have a semaphore access pattern which is a list $(\sum_{i} OP_{i}, A_{i})$, and the list is ordered according to $C8/D6$. Without losing generality, we assume that $OP_{i}$ are operations on binary semaphores $S_{i}$ with $S_{i} = 1$ meaning that it is free and locked otherwise. We shall use $sem(P)$ to denote a set of semaphores currently locked by $P$.

From the access patterns, we can calculate the ceiling $C(s)$ 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 $Pr(P)$ should be replaced with $C(s)$ if $Pr(P) < C(s)$.

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.

1. $(l, \sigma, ((T, OP, A) :: l, d) :: q) \xrightarrow{1}_{Sch} (l, \sigma, ((T - t, OP, A) :: l, d - t) :: q)$ if $t \leq T$
2. $(l, \sigma, ((0, p(s), A) :: l, d) :: q) \xrightarrow{0}_{Sch} (l, \sigma', q)$ if $\sigma(s) = 1$

where $\sigma' = \sigma[s := 0,$

$sem(P) := sem(P) \cup \{s\}$, $Pr(P) := newPr(P, s)$]
Handling Precedence and Resource Constraints

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 of the automaton where the task queue contains a task which misses its given deadline.

Definition 14 (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 $e(P) > 0$. Naturally an automaton $A$ with initial state $(l_0, u_0, q_0)$ is non-schedulable with $Sch$ iff $(l_0, u_0, q_0) \rightarrow^{Sch^*} (l, u, Error)$ 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 [FPY02]. 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 [FMPY03], 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 $E(Sch)$, 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 $\text{release}_i$ 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 \( E(\text{Sch}) \) (the scheduler), encodes the task queue, the scheduling strategy \( \text{Sch} \) and the \textit{task constraints}.

### 4.1 Timing Constraints

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

![Scheduler automaton for tasks with timing constraints.](image)

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 \( E(\text{Sch}) \) are as follows:

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

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

We use the predicate 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: \( \exists P_i \in q \text{ such that } d_i > D_i. \)

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: \( P_i = Hd(Sch(q)) \). For example, Sch can be:

• Highest priority first (FPS):
  \[ P_i \in q, \forall P_k \in q \, \text{Pr}(P_i) \leq \text{Pr}(P_k) \text{ where Pr}(P) \text{ denotes the fixed priority of } P. \]

• 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 \( E(Sch) \), see [FPY02]. 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 \( Hd(Sch(q)) \), the automaton changes its state to Run(P_i), and the execution time clock \( c_i \) is reset. When the value of \( c_i \) reaches \( C_i \) in location Run(P_i), the task \( P_i \) finishes its execution and is being removed from the queue, the automaton changes location to 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 Run(P_i) to 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 \( P_j \) \( E(Sch) \) changes its state to Run(P_j). We use the predicate preemted(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 \( D6/CT/CP/CS/DD/B4 \).

Recall from previous section, the boolean variable \( BZ/CX/BN/CY \) denotes that an instance of task \( C8/CX \) is preceded by an instance of task \( C8/CY \). Then the precedence constraints for each task \( C8/CZ \) is simply the conjunction of booleans \( BZ/CX/BN/CZ \) for all edges \( C8/CX/AX/C8/CZ \) in the given precedence graph. Let \( \text{Ready}(P_k) \) denote the conjunctive formula. Then the union of \( \text{Ready}(P_k) \) for all \( P_k \)’s represents the subset \( \text{ready}(q) \).

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

![Figure 2: Scheduler automaton handling precedence constraints. Changes from the encoding without precedence constraints are shown in the grey rectangles.](image)

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

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

• Finished$\rightarrow$Idle is taken when a subset of the queue with satisfied precedence constraints becomes empty.

• Finished$\rightarrow$Run($P_j$) and Arrived($P_i$)$\rightarrow$Run($P_j$) are taken when $\text{ready}(q)$ is non-empty and then the highest priority task is chosen for execution from this subset.

• Run($P_j$)$\rightarrow$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.

The number of clocks used in the encoding can be reduced in the following way.

For every task that must precede other tasks (predecessor) we are interested in the time point when it finishes its execution. Therefore, in the analysis we need one extra clock for each predecessor. The number of clocks can be reduced further for precedence relations of the form $P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_n$, where task $P_{k-1}$ is an immediate predecessor of $P_k$. For each such sequence we can use only one clock to calculate when any task in the sequence finishes its execution due to the fact that the tasks will execute consequently, and the clock can be reused for any predecessor in the sequence. Hence, for the analysis we need $k$ extra clocks, where $k$ is the number of such sequences.

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.

The following variables and functions are used in the encoding:

• Function $\text{ceil}(s)$ 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 $prio_i$. 

Figure 3: Encoding of highest locker protocol.
- 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 \( time_e \) and increment the counter after each such transition. Invariant \( c_i \leq time_e[k_i] \) 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 \( time_e[k_i] \), 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 \( time_e[k_i] \), 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 \( i := \text{Hd}(\text{Sch}(q)) \). 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. \( i := \text{Hd}(\text{ready}(q)) \).

- **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 \( time_e \) 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 $\text{Run}(P_i)$ 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 $\text{locker}[\text{sem}_i[k_i]] = \text{NONE}$ and $\text{locker}[\text{sem}_i[k_i]] = i$.

The number of clocks used in the encoding can be reduced in the following way. For schedulability analysis of such models, it is important to know the status of semaphores at any time point. Hence, for each task that uses one or more semaphores, we need to use one clock that would run along the task access pattern and trigger the events of locking and unlocking the semaphores. This results in $l$ extra clocks, where $l$ is the number of tasks that use shared resources protected by semaphores.

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 [AFM+02]. The tool currently supports simulation, schedulability analysis, calculation of response times, checking of safety and liveness properties, and synthesis of executable C-code [AFP+02] 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 $\text{RelP}$ and $\text{RelQ}$ 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 $(\text{Idle}, [x = 0, n = 0], [])$ may demonstrate the following sequence of transitions:
Figure 4: A screen-shot of the TIMES tool.

\[
(Idle, [x = 0, n = 0], []) \xrightarrow{0} \\
(ReadP, [x = 0, n = 0], [P(2, 8)]) \xrightarrow{1} \\
(ReadQ, [x = 1, n = 0], [P(1, 7), Q(2, 20)]) \xrightarrow{2} \\
(Idle, [x = 3, n = 1], [Q(1, 18)]) \xrightarrow{0} \\
(ReadP, [x = 0, n = 1], [P(2, 8), Q(1, 18)]) \xrightarrow{1} \\
(ReadQ, [x = 1, n = 1], [P(1, 7), Q(1, 17), Q(2, 20)])
\]

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.

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 s1 and s2. Task P locks semaphore s1 for the first half of its execution and s2 for the second half. Task Q locks both semaphores for the whole its exe-
Handling Precedence and Resource Constraints

Figure 5: Timed automaton with tasks P and Q.

<table>
<thead>
<tr>
<th>Type of constraint</th>
<th>WCRT(P)</th>
<th>WCRT(Q)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timing only</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Precedence</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>Resource</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Precedence and resource</td>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 1: WCRT for tasks with different constraints.

...cution. 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.

6 Conclusions and Related Work

In scheduling theory [But97], 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 [FMPY03], it is shown that for fixed priority scheduling strategy and tasks with only timing constraints (i.e. execution time and deadline), the schedulabil-

...
Handling Precedence and Resource Constraints

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

Figure 7: Precedence graph.

The schedulability analysis problem for the extended model can be solved efficiently using the same technique as in [FMPY03]. 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. [But97]. In the past years, these classic works have been extended to deal with more complex constraints e.g. offset analysis [PH98] and unfolding [BLMSv98] for precedence constraints and priority ceiling protocols [SRL90, RSL98] 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[AM01, Feh99, HLP01]. 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 [AGP+99, AGS00]. 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 [AGS00]. 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**


Paper D:

Code Synthesis for Timed Automata

Tobias Amnell, Elena Fersman, Paul Pettersson, Hongyan Sun, and Wang Yi.
Abstract. We present a framework for the development of real-time embedded systems based on timed automata extended with a notion of real-time tasks. It has been shown previously that reachability and schedulability for such automata can be checked effectively using model checking techniques. In this paper, we propose to use the extended automata as design models. We describe how to compile design models to executable programs with predictable behaviours. The compiling procedure ensures that the execution of the generated code satisfies mixed timing, resource and logical constraints imposed on the design model. To demonstrate the applicability of the framework, a prototype C-code generator based on the legOS operating system has been implemented in the TIMES tool and applied to develop the control software for a production cell. The production cell has been built in LEGO® equipped with a Hitachi H8 based LEGO® Mindstorms control brick.

1 Introduction

A key facility of many commercial tools for development of embedded software is to automatically synthesise executable code (i.e. code generation) from design models. However, few of the existing tools are capable of producing software with predictable timing behaviours, i.e. code which is known a priori to satisfy given timing constraints. In fact, most of the commercial tools trust the software developers to resolve the timing issues e.g. whether all tasks will meet their deadlines, when the generated code is supposed to run on a specific target platform. In contrast, research tools such as UPPAAL [LPY97] and KRONOS [BDM+98] are often dedicated to analysis and abstraction on high-level design descriptions. This has proven useful for finding errors and checking correctness properties in many case studies, e.g. [BGK+96], [LPY98], [SMF97] and [DKRT97]. However, it provides little or no support for producing the actual program code to be executed in the final implementation.
In this work, we aim at synthesis of executable code with predictable behaviours, from high level design specifications. We combine results and ideas from model-checking, scheduling, and synchronous programming to develop a framework which supports, on one hand, formal specification, validation, analysis of design, and on the other hand, code generation which ensures that timing constraints are met when the generated code is executed on the target system. As a design language, we use timed automata extended with real-time tasks [FPY02]. In such extended automata, a transition is associated with a task (or several tasks) that is an executable program with given parameters, such as execution time, deadline, fixed priority etc. Intuitively, whenever an automaton takes a transition, its associated task is released for execution. Thus a discrete transition denotes an event releasing a task and the clock constraints on the transition (i.e. the guards) specify the possible arrival times of the associated task. When a task is released, it is inserted into the ready queue of the operating system and executed according to its scheduling policy.

It has been shown that the schedulability checking problem for the model of extended automata is decidable for both non-preemptive [EWY99] and preemptive [FPY02] scheduling policies. An automaton is schedulable with a given scheduling policy if, for all possible sequences of events accepted by the automaton the released tasks can be computed within their deadlines. The check is performed by transforming the original scheduling problem to a reachability problem of timed automata extended with subtraction operations on clocks. This means that, given a design model of a timed system, described as a set of automata running in parallel, it can be checked whether it is schedulable prior to its implementation provided that the computing times on the target hardware for the associated tasks are known.

In this paper, we show how to compile a (verified) design model to an executable program preserving the mixed timing and logical constraints imposed on the model. Inspired by the design philosophy of synchronous languages e.g. Esterel [BG92], we assume that the target real-time operating system and hardware ensure the synchrony hypothesis, i.e. the run-time of certain system functions is neglectable compared with the execution times and deadlines of the application tasks. Based on this assumption, we transform the control structure of an automaton to operating system functions and the associated tasks to lightweight threads. The compiling procedure should be applicable to a variety of target platforms that guarantee the assumption.

To demonstrate the applicability, we have developed a prototype C-code generator on the legOS operating system for the LEGO® Mindstorms control brick,
equipped with an 8-bit Hitachi micro-processor. The code generator has been integrated in the TIMES tool [AFM+02] and applied to develop the control software for a production cell built in LEGO® and controlled by LEGO® Mindstorms control bricks. The control program for the most central component, the two-armed robot, is designed, analysed and generated using the TIMES tool. The generated code has been compiled, downloaded, and executed on LEGO® Mindstorms bricks.

The rest of this article is organised as follows: Section 2 describes the syntax and semantics of design models that are timed automata extended with tasks. To extract executable behaviours from design models, we present a deterministic semantics for the extended automata. Section 3 shows how to implement the deterministic semantics on a small generic embedded operating system. Section 4 describes the modelling and design of a production cell using the extended automata. Section 5 is devoted to the analysis and synthesis of the control software for the production cell. Section 6 concludes the paper.

2 The Design Language

In [EWY99] and [FPY02], an extended version of timed automata with real time tasks is presented. The idea is to annotate each transition of an automaton with a task (an executable program with computing time and deadline), that will be triggered when the transition is taken. The triggered tasks will be scheduled to run according to a given scheduling policy. This provides a general task model for real time systems, which can be used to solve scheduling problems.

In this paper, we adopt a more expressive version of timed automata with tasks. We allow that an automaton and the annotated tasks may have shared variables. The shared variables may be updated by the execution of a task (or the automaton) and their values may effect the behaviour of the automaton.

2.1 Syntax

Assume a set of variables \( D \) ranged by \( u, v \). We assume that the variables take values from finite data domains. The variables may be updated by assignments in the form: \( u := E \) where \( E \) is a mathematical expression. We use \( R \) to denote the set of assignments (e.g. \( u := u + 1 \)).
Real Time Tasks. Let $\mathcal{P}$ ranged over by $P, Q, R$, denote a finite set of task types (executable programs written in a programming language). A task type may have different instances that are copies of the same program with different inputs. We assume that the execution times and hard deadlines of tasks in $\mathcal{P}$ are known\(^1\).

Further assume that a task may update the data variables by the end of its computation using assignments in the form $u := \mathcal{E}$ (computed by the task and the value of $\mathcal{E}$ is returned when the task is finished). Thus, each task $P$ is characterised as a triple denoted $(C, D, A)$ with $C \leq D$, where $C$ is the execution time of $P$, $D$ is the relative deadline for $P$ and $A$ is the set of assignments updating data variables. The deadline $D$ is a relative deadline meaning that when task $P$ is released, it should finish within $D$ time units. We shall use $C(P)$, $D(P)$ and $A(P)$ to denote the execution time, relative deadline and set of assignments of $P$ respectively.

Timed Automata. Assume a finite set of actions $\mathcal{A}$ and a finite set of real-valued variables $\mathcal{C}$ for clocks. We use $a, b, \tau$ etc to range over $\mathcal{A}$, where $\tau$ denotes a distinct internal action, and $x_1, x_2$ etc to range over $\mathcal{C}$. We use $B(\mathcal{C})$ 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 $B_I(\mathcal{C})$ for the subset of $B(\mathcal{C})$ where all atomic constraints are of the form $x \prec C$ and $\prec \in \{<, \leq\}$. The elements of $B(\mathcal{C})$ are called clock constraints. These are the basic syntactical objects needed to define timed automata i.e. finite state automata extended with clocks.

We shall allow automata to test (or read) and update data variables. To read and test the values of data variables, we assume a set of predicates $B(\mathcal{D})$ (e.g. $u \leq 10$). Let $B = B(\mathcal{D}) \cup B(\mathcal{C})$ be ranged over by $g$ called guards. Further let $A = \mathcal{R} \cup \{x := \theta \mid \theta \in \mathcal{C}\}$ be called resets. We use $r$ to stand for a subset of $A$.

**Definition 15** A timed automaton extended with tasks, over actions $\mathcal{A}$, clocks $\mathcal{C}$, data variables $\mathcal{D}$ 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,
- $I \subseteq N$ is the initial location.

\(^1\)Note that tasks may have other parameters such as fixed priority for scheduling and other resource requirements e.g. on memory consumption. For simplicity, in this paper, we only consider computing time and deadline.
\(E \subseteq N \times B \times \text{Act} \times 2^A \times N\) is the set of edges, and

\(I: N \mapsto B_1(C)\) is a function assigning each location with a clock constraint (a location invariant).

- \(M: \text{Act} \mapsto \mathcal{P}\) is a partial function assigning actions with tasks \(^2\).

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). When the task is finished, the data variables will be updated by the assignments \(A(P)\).

Note that a boolean can be used to denote the completion of a task. The arrivals of events may be guarded by the boolean, which implies that any task to be triggered by the events can not be started before the completion of this task. Thus, we can describe precedence constraints over tasks.

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 [LPY95]) using the notion of synchronisation function [HL89]. For example, consider the parallel composition \(A || B\) of \(A\) and \(B\) over the same set of actions \(\text{Act}\). 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 [HL89], the action set of the parallel composition will be \(\text{Act}\) and thus the task assignment function for \(A || B\) is the same as for \(A\) and \(B\).

### 2.2 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 corresponds to the arrival of new task instances.

\(^2\)Note that \(M\) is a partial function meaning that some of the actions may have no task. Note also that we may also associate an action with a set of tasks instead of a single one. It will not introduce technical difficulties.
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 $\sigma$. Naturally, a semantic state of an automaton is a triple $(l, \sigma, q)$ where $l$ is the current control location, $\sigma$ denotes the current values of variables, and $q$ is the current task queue. We assume that the task queue takes the form: $[P_1(c_1, d_1), P_2(c_2, d_2) ... P_n(c_n, d_n)]$ where $P_i(c_i, d_i)$ denotes a released instance of task type $P_i$ with remaining computing time $c_i$ and relative deadline $d_i$.

Assume that there is a processor running the released task instances according to a certain scheduling strategy $\text{Sch}$ e.g. FPS (fixed priority scheduling) or $\text{EDF}$ (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 $\delta$ time units is to execute the task in the first position of the queue for $\delta$ time units. Thus the delay transition will decrease the computing time of the first task with $\delta$. If its computation time becomes $\beta$, the task should be removed from the queue (shrinking).

- $\text{Sch}$ is a sorting function for task queues (or lists), that may change the ordering of the queue elements only. For example, $\text{EDF}([P(3.1, 10), Q(4, 5.3)]) = [Q(4, 5.3), P(3.1, 10)]$. We call such sorting functions scheduling strategies that may be preemptive or non-preemptive.

- $\text{Run}$ 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 $\text{Run}(q, t)$ should be obvious and it can be defined inductively. For example, let $q = [Q(4, 5), P(3, 10)]$. Then $\text{Run}(q, 6) = [P(1, 4)]$ in which the first task is finished and the second has been executed for 2 time units.

Further, for a real number $t \in \mathbb{R}_{\geq 0}$, 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 $\sigma$ satisfies the guard $g$ and $\sigma[r]$ to denote the valuation which maps each variable $\alpha$ to the value of $\mathcal{E}$ evaluated in $\sigma$ if $\alpha := \mathcal{E} \in \tau$ (note that $\mathcal{E}$ is zero if $\alpha$ is a clock) and agrees with $\sigma$ for the other variables. Now we are ready to present the operational semantics for extended timed automata by transition rules:
Definition 16 (Operational semantics) Given a scheduling strategy \( \text{Sch} \), the semantics of an extended timed automaton \( \langle N, I_0, E, I, M \rangle \) with initial state \( (I_0, \sigma_0, q_0) \) is a transition system defined by the following rules:

- \( (l, \sigma, q) \xrightarrow{a, \tau}_{\text{Sch}} (m, \sigma[r], \text{Sch}(M(a) :: q)) \) if \( l \xrightarrow{\sigma[a, \tau]} m \) and \( \sigma \models g \)

- \( (l, \sigma, q) \xrightarrow{t}_{\text{Sch}} (l, \sigma + t, \text{Run}(q, t)) \) if \( (\sigma + t) \models I(l) \) and \( C(Hd(q)) > t \)

- \( (l, \sigma, q) \xrightarrow{t}_{\text{Sch}} (l, (\sigma[A(Hd(q)]) + t, \text{Run}(q, t)) \) if \( (\sigma + t) \models I(l) \) and \( C(Hd(q)) = t \)

where \( M(m) :: q \) denotes the queue with \( M(m) \) inserted in \( q \) and \( Hd(q) \) denotes the first element of \( q \).

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.

There are two kinds of delay transitions as defined in the second and the third rules. In the second rule, the transition corresponds to the execution of the first task in the queue. The transition is enabled when the invariants are satisfied and the task has not completed the execution during the delay. The transition results in updates of the clock variables and a decrease of the remaining execution time and deadline time of the task at the head of the queue.

In the third rule, the transition corresponds to completing the execution of the task at the head of the queue. The delay is the same as the remaining execution time. When the transition is taken the variables are updated according to the assignment of the task. Note that we assume a fixed execution time \( C(P) \) for a task. It is possible to extend the model and the analysis so that the execution times vary in an interval between best and worst case execution time.

We shall omit \( \text{Sch} \) from the transition relation whenever it is understood from the context.

---

3Note that we fixed \( \text{Run} \) to be the function that represents a one-processor system.

4In case the task queue is empty we interpret the conditions \( C(Hd(q)) > t \) and \( C(Hd(q)) = t \) as true. Hence the two last transition rules becomes equal and corresponds to a delay transition in ordinary timed automata.
2.3 Analysis of Design Model

It has been shown that reachability and schedulability are decidable problems for the class of timed automata extended with tasks both for non-preemptive [EWY99] and preemptive scheduling policies [FPY02]. In this section we give the definitions of these and define the boundedness problem, which should also be checked prior to code-synthesis. For a more detailed description on how to check schedulability of task extended timed automata we refer the reader to [FPY02] and [FMPY03].

We use the same notion of reachability as for ordinary timed automata:

**Definition 17 (Reachability)** We shall write \( (l, \sigma, q) \rightarrow (l', \sigma', q') \) if \( (l, \sigma, q) \xrightarrow{a} (l', \sigma', q') \) for an action \( a \) or \( (l, \sigma, q) \xrightarrow{t} (l', \sigma', q') \) for a delay \( t \). For an automaton with initial state \( (l_0, \sigma_0, q_0) \), \( (l, \sigma, q) \) is reachable iff \( (l_0, \sigma_0, q_0) \rightarrow^* (l, \sigma, q) \).

Thus a state \( (l, \sigma, q) \) is reachable if there is a sequence of transitions starting in the initial state \( (l_0, \sigma_0, q_0) \) and ending in \( (l, \sigma, q) \). We shall use reachability analysis to check safety properties of design models to conclude that undesired situations (states) are not reachable. For example, we may check that the size of the task queue for all reachable states is bounded. Note that the boundedness is a useful property, which may be used for estimation of memory consumption. Further, we may use reachability analysis to check the schedulability of a design model prior to the final implementation.

**Definition 18 (Schedulability)** A state \( (l, \sigma, q) \) where \( q = [P_1(c_1, d_1), \ldots, P_n(c_n, d_n)] \) is a failure denoted \( (l, \sigma, \text{Error}) \) if there exists \( i \in [1..n] \) such that \( c_i \geq 0 \) and \( d_i < 0 \), that is, a task failed in meeting its deadline. An automaton \( A \) with initial state \( (l_0, \sigma_0, q_0) \) is non-schedulable with scheduling policy \( \text{Sch} \) if and only if \( (l_0, \sigma_0, q_0) \rightarrow^* \text{Sch}(l, \sigma, \text{Error}) \) for some \( l \) and \( \sigma \). Otherwise, we say that \( A \) is schedulable with \( \text{Sch} \).

Intuitively, a system is schedulable if for all reachable states, all tasks are guaranteed to meet their deadlines. It should be noticed that schedulability implies boundedness but not the other way around, as clearly many bounded ready queues are not schedulable.

In order to check the above properties of a system design, given as an extended timed automaton \( C \) (possibly a parallel composition \( \tilde{C}_1 \parallel \ldots \parallel \tilde{C}_n \)), we will need to take the behaviour of its environment into consideration. Recall that in the extended model of timed automata adopted in this paper, new task in-
stances are released at the action transitions, corresponding to input signals received from the environment of the controller C. To model the environment, we compose in parallel with C timed automata models $E_1 \parallel \ldots \parallel E_m$ of its environment, and analyse properties of the complete system design $S^{Design} = C \parallel E_1 \parallel \ldots \parallel E_m$.

We shall see in the next section that the code-synthesis is guaranteed to preserve safety, schedulability and boundedness properties. This means that these properties of a system design can be checked prior to its implementation.

### 2.4 Deterministic and Executable Semantics

We shall consider $S^{Design}$ as a design model. Code synthesis is to generate executable code from the design model that implements the controller. In general the behaviour of the design model according to the operational semantics (cf. Definition 2) is non-deterministic. Our goal is to extract deterministic behaviour for the controller, preserving safety properties satisfied by the design model. Thus, our problem is essentially to resolve non-determinism.

The sources of non-determinism in the operational semantics are time-delays and external actions. We use time non-determinism to mean that an enabled transition can be taken at any time-point within the time zone, while we use external non-determinism to mean that several actions may be simultaneously present from the environment which results in several enabled transitions. We say that a transition $\rightarrow = l \xrightarrow{g,\sigma,q} m$ is enabled in state $s = (l,\sigma,q)$, denoted $\text{Enabled}(\rightarrow, s)$, when its guard is true i.e. $\sigma \models g$ and the environment is ready to synchronise on the action $a$.

We shall refine the design model for the controller as follows:

- External non-determinism is resolved by assigning priorities to transitions in the controller. Let $Pr : E \mapsto Z$ be a function assigning a unique priority to each edge, which defines an order that the guards of edges are evaluated in. If several transitions are enabled in a state, they should be taken in priority order.

- Time non-determinism is resolved by implementing the so-called maximal-progress assumption [Yi91]. Maximal-progress means that the controller should take all enabled action transitions until the system stabilises, i.e. no more action transitions are enabled. Similar ideas have been adopted.
We can now give a refined (deterministic) version of the operational semantics for extended timed automata:

**Definition 19 (Deterministic semantics)** Let \( A = \langle N, l_0, E, I, M \rangle \) be an extended timed automaton. Given a scheduling strategy \( \text{Sch} \) and a function \( \Pr \) that assign priorities to edges the deterministic semantics is a labelled transition system defined by the rules:

1. \((l, \sigma, q) \xrightarrow{a}(m, \sigma[r], \text{Sch}(M(a) :: q))\) if \( \text{Enabled}(l \xrightarrow{g,a,r} m, (l, \sigma, q)) \) and there is no \( \rightarrow \in E \) such that \( \Pr(\rightarrow) > \Pr(l \xrightarrow{g,a,r} m) \) and \( \text{Enabled}(\rightarrow, (l, \sigma, q)) \)

2. \((l, \sigma, q) \xrightarrow{t}(l, \sigma + t, \text{Run}(q, t))\) if \((\sigma + t) \models I(l) \) and \( C(\text{Hd}(q)) > t \) and for all \( \rightarrow \in E \) and \( d < t \) such that \( \neg\text{Enabled}(\rightarrow, (l, \sigma + d, q)) \)

3. \((l, \sigma, q) \xrightarrow{t}(l, (\sigma[A(\text{Hd}(q))] + t, \text{Run}(q, t))\) if \((\sigma + t) \models I(l) \) and \( C(\text{Hd}(q)) = t \) and for all \( \rightarrow \in E \) and \( d < t \) such that \( \neg\text{Enabled}(\rightarrow, (l, \sigma + d, q)) \)

Clearly the behaviour of the controller according to the refined semantics is deterministic. Safety properties in the design model are preserved since the behaviour defined by deterministic semantics is included in the behaviour defined by the operational semantics. That is, all sequences of transitions (i.e. executions) of a design model for a controller according to the deterministic semantics can also be taken according to the operational semantics.

Note that according to the semantics, we may have automata that exhibit zeno-behaviours, that is, infinite sequences of action transitions within a finite time delay. Such automata correspond to non implementable design models, and they should be discovered by schedulability analysis as non schedulable. To simplify the analysis, we adopt the following syntactical restriction. We shall consider only automata in which each cycle contains at least one edge associated with a task.

### 3 Code Synthesis

Now we show how to transform a design model to an executable program according to its deterministic semantics. We assume a generic target platform which guarantees the synchronous hypothesis and on which the associated tasks
will consume their given computing times to execute. As our transformation preserves the deterministic semantics, the schedulability of a design model will be preserved by the generated code when it is executed on the target platform.

We assume that the target OS provides the following generic features:

- threads with unique priority levels,
- a scheduler based on fixed priority assignment,
- interrupt handling routines.

The first two requirements are needed to map the notion of tasks from the design model to the generated code. Tasks have unique priorities in the model. Thus when they are mapped to threads in the implementation the threads must also have unique priorities. The last requirement is needed to encode control automata, which is done by interrupts using interrupt handling routines.

The intended target platform for the generated code is a single embedded processor, but the organisation of the code is more easily illustrated by assuming two processors as shown in Figure 1. The code of the controller and the tasks execute on two separate logical processors, a control processor and a task processor. The interaction between the two is limited to the scheduling queue and shared variables. The control processor receives events from the environment and inserts tasks into the scheduling queue. The task processor executes the task at the head of the scheduling queue and updates the shared variables when a task is done.

To realise the execution model on a single processor, we execute the code of the controller automata as a separate thread with a priority higher than all the other threads.

### 3.1 Handling Tasks and Variables

Tasks are executed in threads, one thread for each task type, with priorities lower than the controller thread. Scheduling of task threads and management of the ready queue are handled by the target operating system. We assume boundedness of the queue length, meaning that the memory allocated for the task queue can be fixed at compile-time and no exception handling for queue overflow is needed at run time.

Data variables in the design model are mapped to global integer variables in the generated code. To encode clocks, let $\varphi$ be a global system clock. For each
clock $x$ in the timed automata, let $x_{\text{reset}}$ be an integer variable holding the system time of the last clock reset. The value of the clock is then $(sc - x_{\text{reset}})$, and a reset can be performed as $x_{\text{reset}} := sc$.

3.2 Encoding and Executing the Controller Automata

In the generated program, the controller automata are encoded as four look-up tables and two functions, as shown in Figure 2. Three of the look-up tables are static and used to encode the edges, the locations, and the synchronisations respectively. The fourth table is dynamic and used to hold the set of currently active edges, i.e. edges leaving the current control location (or locations, if the controller consists of parallel automata). We use ACTIVE to denote the list of active edges.

The table containing all edges is sorted in priority order. For each edge there are five fields: guard, assign, from, to and sync. The guard and assign fields are references to code that when invoked will evaluate the guard and perform the assignments of the edge. The fields from and to are references into the table of locations. The field sync is either empty if the edge has no synchronisation label, or a reference to a list of edges having the complementary synchronisation label.

In the table containing all locations, we store for each location a list of all edges leaving the location. In the table containing synchronisation labels, we
store references to all edges at which the label occurs, and also the set of tasks associated with the label. Note that some of the information stored in the tables (such as the synchronisation label of an edge) is intentionally duplicated to improve efficiency.

The list of active edges, ACTIVE, is managed by the procedure shown in Table 1 where we use $\pi$ to denote the complementary synchronisation label of $a$, e.g. $\text{sync!}$ for $\text{sync?}$. We use $\text{aut}$ to denote a function that takes a location as parameter and returns an unique identifier for the automaton that the location belongs to. We use $\text{sort}$ to denote a sorting function that takes as input two parameters: a list of edges, and a function assigning unique priorities to edges, and returns as output a list of edges sorted according to the assigned priorities.

The procedure in Table 1 is executed by the controller thread whenever an event (such as timeout or arrival of an external event) has occurred. When executing, no new events are processed and the timers are not updated, i.e. the whole procedure is executed in a critical region. Note that this means that the tasks will not be able to update shared variables while the procedure is executing.

Initially the list of active edges consists of all edges leaving the initial locations. The procedure scans ACTIVE in priority order and evaluates the corresponding guards. If a guard is found to be satisfied, there are two cases:

- no synchronisation - the assignment is performed and the information in the table of locations is used to update the list of active edges and to release any tasks associated with the edge.

- synchronisation - (i.e. the edge is labelled with an action label) the information in the table of synchronisations is used to find an active edge
Table 1: Procedure, in pseudo-code, that executes the encoded controller.

### Initially:

\[
\text{ACTIVE} := \{ l \xrightarrow{g.a,r} l' \mid l = l_0 \}
\]

\[
\text{ACTIVE} := \text{sort(ACTIVE, Pr)}
\]

### Algorithm:

```
START:
for each \( l \xrightarrow{g.a,r} l' \) in ACTIVE do
  if \( \sigma \) satisfies \( g \) then
    if \( a = r \) then
      \( \sigma := \sigma[r] \)
      remove \( \{ n \rightarrow n'|n = l \} \) from ACTIVE
      add \( \{ n \rightarrow n'|n = l' \} \) to ACTIVE
      ACTIVE := \text{sort(ACTIVE, Pr)}
      \( q := \text{Sch}(M(a) :: q) \)
      goto START
    else
      if exists \( m \xrightarrow{g'.a',r'} m' \) in ACTIVE s.t. \( \sigma \) satisfies \( g' \) and \( \text{aut}(m) \neq \text{aut}(l) \) then
        if \( a \) is sending then
          \( \sigma := \sigma[r]; \sigma := \sigma[r'] \)
        else
          \( \sigma := \sigma[r']; \sigma := \sigma[r] \)
        \end{if}
        remove \( \{ n \rightarrow n'|n = l \lor n = m \} \) from ACTIVE
        add \( \{ n \rightarrow n'|n = l' \lor n = m' \} \) to ACTIVE
        ACTIVE := \text{sort(ACTIVE, Pr)}
        \( q := \text{Sch}(M(a) :: M(\bar{a}) :: q) \)
        goto START
      \end{if}
  \end{if}
fi
fi
od
```

belonging to another control automata with complementary action label and satisfied guard. If such an edge is found, the compound transition is performed i.e. the assignments of the two edges are performed, ACTIVE is updated and the tasks associated with action label are released.
When a transition has been executed the procedure returns to \texttt{START} and re-examines the (updated) list of active edges to check for another transition to be taken. The procedure will continue to do so as long as enabled edges exist in \texttt{ACTIVE}. When there are no more enabled edges, the procedure will terminate and let the OS execute task threads. As an effect, the procedure implements a so-called \textit{run-to-completion} step ensuring that the generated code has the maximal-progress behaviour assumed by the deterministic semantics (see Section 2.4). Note that since \texttt{ACTIVE} is always kept sorted and the procedure always scans from the head of the list, the implementation is also deterministic with respect to external actions.

### 3.3 Correctness

In this section we argue for the correctness of the code synthesis by emphasising how the deterministic semantics (c.f. Definition 19) of timed automata with tasks is realised in the generated code. The three components of a semantic state \((l, \sigma, q)\) are mapped to the following parts of the synthesised code:

- **\(l\):** the source locations of the edges in \texttt{ACTIVE} (the list of active edges),
- **\(\sigma\):** integer variables for the data variables in \(\sigma\), and the difference between reset-time and system time for the clocks in \(\sigma\),
- **\(q\):** the scheduling queue of the operating system.

Note that there is no explicit representation of control location \(l\) in the synthesised code, instead the current control location is implicitly represented by the edges in \texttt{ACTIVE}.

Initially, when the synthesised code starts to execute, the semantic state \((l_0, \sigma_0, q_0)\) is represented in the code as:

- **\(l_0\):** \texttt{ACTIVE} is initialised with the edges leading out from the initial locations,
- **\(\sigma_0\):** data variables are initialised to their initial values and the reset-time of clocks are initialised to the system time at startup,
- **\(q_0\):** the scheduling queue is empty.

The transition rules defined in the deterministic semantics (Definition 19) are mapped in the following way:
1. Discrete transition - \((l, \sigma, q) \xrightarrow{a} (l', \sigma', q')\) is handled by the procedure in Table 1. All edges that lead out from the source location are removed from \textsc{active} and all edges that lead out from the target location are added to \textsc{active}. Data variables are updated according to the assignments. Clocks are reset by assigning their reset-time to the current value of the system clock.

2. Delay transition without task termination - \((l, \sigma, q) \xrightarrow{t} (l', \sigma', q')\) corresponds to the execution of a task. The location component of the state does not change during task execution. Data variables are not updated. Clocks are updated by the delay \(t\) during task execution since the difference between the last reset-time and the system clock will increase. In the queue \(q\), the remaining execution time of the task at the head of the task queue, as well as the deadline of all tasks, will be \(t\) time units shorter.

3. Delay transition with task termination - \((l, \sigma, q) \xrightarrow{t} (l', \sigma', q')\) corresponds to execution of a task. The location component of the state does not change during task execution. Data variables are updated by the assignment of the task. Clocks are updated by the delay \(t\) during task execution since the difference between the last reset-time and the system clock will increase. The queue is reduced by removing the task at the head of the queue, and the relative deadline of the other tasks will be \(t\) time units shorter.

The refinements introduced to resolve non-determinism are implemented as follows:

\textit{Priority} - Recall that all edges are assigned unique priorities. The controller thread evaluates the guards of the active edges in priority order from high to low. The first transition whose guard is satisfied is taken. In the case of synchronisation the complementary transition with the highest priority whose guard is satisfied is taken. The order of the active edges is maintained when replacing outgoing edges of the source location with those of the target location by re-sorting before proceeding.

\textit{Maximal Progress} - The code has maximal-progress behaviour since, when a discrete transition has been taken the list of active edges is checked again from the beginning. Only when no edge has a satisfied guard the controller will suspend and let other threads with lower priority (i.e. tasks) execute.
3.4 Prototype for legOS

The code synthesis described above has been implemented in a version of the TIMES tool [AFM+02] to generate code for the legOS [Lt02] operating system — a small open source operating system for the Hitachi H8 processor — embedded into the LEGO® Mindstorms RCX control brick. The OS is implemented in C and for simplicity we use C as implementation language for the tasks as well.

The Hitachi H8 processor in the RCX unit is equipped with 32 kB of RAM, a typical setup for the type of embedded systems that our code generation is intended for. The I/O interface consists of three sensor inputs and three actuator outputs.

In legOS threads are separate control flows that are scheduled on the CPU by the operating system. The scheduler implements preemptive fixed priority scheduling and handles up to 20 priority levels. Threads with equal priorities are executed in a round robin fashion. This limits the number of task types in the design to at most 19, since each task type needs a unique priority level and the highest priority level is reserved for the controller thread.

Another limitation in the current version of legOS is in the interrupt handling. The OS provides so-called wake-up functions to program event handling. A wake-up function is a general mechanism provided by legOS that lets a thread wait for some condition (such as the release of a semaphore, a timeout, a key press etc.). A thread registers a boolean function that the scheduler will execute when it looks for the next thread to run, which it will do when the current thread is suspended or periodically every 20 ms.

To make sure that the control procedure of Table 1 is executed at every event handling, we encode the entire control procedure as a wake-up function that always returns false, and is associated to a task at highest priority level. In this way, the control procedure is ensured to be executed by the operating system (atomically and at OS priority level) just after every event handling, and just before the operating system determines the next task thread to be executed.

For each task type a corresponding thread is created at boot time. The task threads contains a loop repeating a suspend call, and the task body to be executed. The suspend call registers a wake-up function that checks for a non-zero value of an element in the integer array release list. This array contains one

---

5The period of the scheduler (the time-slice) can be modified by recompiling the OS.
element for each task type. When the controller releases a task the correspond-
ing element is incremented, so that the wake-up function of the task thread will
return and resume the task thread. When the task body has executed the corre-
sponding element in the release list is decremented, and the thread is suspended
again at the start of the loop. Note that since priorities are fixed, the release list
can be used as a representation of the current state of the ready queue.

Sensor readings in legOS are available to the generated program as variables
which are updated independently by the hardware. Sensor variables are either
read by the task code, or used directly in guards of the control automata. This
means that checking if a transition is enabled becomes a condition only involv-
ing variables (external and other), i.e. essentially the same as a guard.

In the current prototype we associate tasks with locations instead of edges. The
tasks are released when the location is entered. Such design models can eas-
ily be transformed into models with tasks on edges by associating the task of
a location with all edges leading to the location. Furthermore we impose the
syntactic restriction that a control automaton may not use both a sending and
a receiving action label with the same name (e.g. not both sync! and sync? in
the same automaton).

4 Modelling and Design of a Production Cell

In this section, we show how to use our design language to model and design
the control software for an industrial robot. The production cell is a unit in a
metal plate processing plant in Karlsruhe. The model has been developed by
FZI in Karlsruhe [LL95] as a benchmark example of concurrent and safety-
critical system software development. In this paper, we use a LEGO® model
of the production cell based on the model developed by FZI but with some
simplifications. The LEGO® production cell system contains four subsystems,
the feed belt, the robot, the press and the deposit belt, corresponding to the
physical components as shown in Figure 3. Each of the subsystems performs a
specific operation during the plate processing:

The feed belt transfers a plate from the entry position to the end position where
the robot arm named arm A can pick up the plate. The press forges the plate
delivered by arm A of the robot, and the forged plate is unloaded from the press
by arm B of the robot. The robot moves to the position where arm A points
to the feed belt and picks up a plate. It then rotates until arm A points to the
press where arm A delivers a plate. When a plate has been forged in the press,
it moves to the position where arm B points to the press and picks up the forged plate. Afterwards, the robot rotates until arm B points to the deposit belt where it delivers the plate. The deposit belt receives a forged plate from arm B of the robot.

4.1 Overall Control Structure

Each subsystem of the feed belt and the robot comprises sensors and actuators for the physical component in the subsystem plus a controller (i.e. a control program) for controlling the sensors and actuators. In our version of the production cell, both the press and the deposit belt contain only the physical components (without any sensor and actuator attached) as media for the robot to convey a plate. The status of a plate in both components are modelled in the robot controller by means of internal variables. The overall control structure of the production cell is sketched as in Figure 4.

Robot Controller. The robot controller is composed of two processes (or sub controllers), RobotControl and MoveTo as shown in Figure 4. The RobotControl process controls the overall motions of the robot, and communicates with both the feed belt controller and the local process MoveTo. The MoveTo process controls the robot movements according to the rotation sensor (i.e. sensor 3 in Figure 3) information and the requests from RobotControl.
The RobotControl process communicates with the feed belt controller by means of the shared variable platePos which indicates whether there is a plate on the feed belt. It communicates with MoveTo by means of communication channels i.e. startM to start the robot movement and stopM to stop the robot movement.

Feed Belt Controller. The feed belt controller is composed of the BeltPosition process and the Alarm process. The BeltPosition process updates the positions of the plates on the feed belt periodically after they are detected by the light sensor. The Alarm process handles the light sensor (i.e. sensor 1 in Figure 3) at the entry position of the feed belt. The light sensor senses the arrival of a plate on the feed belt. The communication between BeltPosition and Alarm is through the shared variable platePos which indicates the position of a plate on the belt. Note that the feed belt controller cannot stop the belt. The bricks will be picked up “on-the-fly” by the magnet on arm A.

4.2 Robot Controller Model

The robot controller is modelled as two extended timed automata, named RobotControl and MoveTo. The shared variable platePos is modelled by an integer array Pos, and the communication channels startM and stopM are respectively modelled by synchronisation channels start and stop, together
with a shared integer variable `Goal` that sets the goal position that the robot should reach.

**Automaton RobotControl.** Figure 5 shows the concrete model of the automaton `RobotControl`. Recall that in our prototype implementation tasks are associated with locations. In the automata shown in this section and in Appendix A, tasks are indicated as labels with bold font in the locations.

In Figure 5, `Pos[0]` is the first element in array `Pos`. `Pos[0] > -1` indicates that there is a plate on the feed belt, `Pos[0] == -1` indicates that there is no plate on the feed belt, and `Pos[0] == 0` indicates that the plate is at the position where arm A of the robot can pick it up.

The boolean variable `PlateInPress` is used to model the status of the press (TT means that there is a plate on the press and FF means no plate on the press). The constant `PRESS_T` is used to model the time needed for the press to forge a plate, and the constant `PREPARE_T` is used to model the time needed for the press to be ready to process a new plate. The clocks `PressTime` and `PrepareTime` are respectively used to measure whether `PRESS_T` and `PREPARE_T` are reached.
The boolean variables PlateOnA and PlateOnB are used respectively to indicate the status of arm A and arm B. PlateOnA==TT denotes that arm A of the robot holds a plate, and PlateOnA==FF denotes that arm A is empty. PlateOnB==TT denotes that arm B of the robot holds a plate, and PlateOnB==FF denotes that arm B is empty. The boolean variable TaskDone is shared with the tasks released by the process. TaskDone==TT denotes that the current task e.g. the pickup task of arm A is finished, and TaskDone==FF denotes that the current task has not finished yet.

The constants WAIT_POS, FB_POS, PRESS_A_POS, PRESS_B_POS and DB_POS are five goal positions for the robot to reach in the robot working space.

The automaton is initially at location AtW8, corresponding to that the robot is waiting for the control commands (or control events). The rest of the locations can be classified into three groups corresponding to the robot actions:

- The robot is moving from one position to the other, e.g. the locations Moving2FB and Moving2PrB. At the former location the robot is rotating so that arm A points to the feed belt, and at the latter the robot is rotating so that arm B points to the press. While waiting in these locations MoveTo control the movement of the robot.

- The robot is waiting for an event, e.g. the locations W8AtFB and AtPressB. The former represents that the robot stops at the position where arm A points to the feed belt and waits for the event Pos[0]==0, and the latter that the robot stops at the position where arm B points to the press and waits for the event PlateInPress==TT.

- The robot is carrying out the pickup or drop tasks, e.g. the locations PickUpA and DropA. At the former location arm A is picking up a plate from the feed belt, and the latter arm A is dropping the plate onto the press.

On each of the edges to PickUpA, PickUpB, DropA and DropB, there is a executable task associated. For example, on the edge to PickUpA the task PickUpA is released for arm A to pick up a plate from the feed belt, while the edge to DropA is associated with the task DropA by which arm A drops a plate onto the press.

**Automaton MoveTo.** The automaton MoveTo is shown in Figure 6. The variable RobotAngle stores the sensor information about the current robot position. The value in ROTATION_1 stores the updated robot position while the
The automaton is initially in location Entry corresponding to that the robot stops and waits for the signal from RobotControl over channel start. Having received the signal the task RdAngSen is released that reads the sensor information about the current robot position into the variable RobotAngle and the automaton moves to location Read. The task RdAngSen has deadline 16, see Appendix A, so the invariant x ≤ 30 together with the guards x = 30 on the outgoing edges will guarantee that RobotAngle is updated before the location is left.

Based on the value in RobotAngle, the automaton moves either to location MR if RobotAngle ≤ Goal or to location ML if RobotAngle ≥ Goal, or to location Entry if RobotAngle = Goal.

On the edge to location MR, the associated task MvRight commands the motor to rotate the robot clockwise. When the robot is in the desired position, i.e. ROTATION_1 = Goal, the automaton takes a transition to location SR.

Similar to MR, on the edge to ML, the associated task MvLeft commands the motor to rotate anti-clockwise the robot. When the robot is in the desired position, i.e. ROTATION_1 = Goal, the automaton takes a transition to location SR.

On both edges to SR, an associated task StopRobot is released to stop the motion of the robot. The automaton takes an urgent transition to location En-
try when the robot is stopped, at the same time it informs RobotControl over channel stop.

### 4.3 Feed Belt Controller Model

Similar to the robot controller, we use two timed automata extended with real-time tasks to model the two processes of the feed belt controller. The two automata BeltPosition and Alarm are given in Appendix A.

**Automaton Alarm.** The automation Alarm is shown in Figure 8 in Appendix A. Before it enters the initial location Calib the task Calib is released that performs calibration of the background light by measuring the mean light value, and saves it in the shared variable normalLight. Finally the task sets the boolean variable AlarmTaskDone to TT. When the sensor is calibrated, i.e., AlarmTaskDone==TT, the automaton takes a transition to location WaitFrontEdge.

In location WaitFrontEdge, the automaton waits until the light value measured by sensor 1 (cf. Figure 3) falls below 80% of the normal light. This indicates that the front end of a plate has passed through. The automaton then moves to location WaitBackEdge, where it waits for the whole plate to pass through the sensor, i.e. the measured light value rises to above 90% of the normal light. Once the whole plate has passed through the sensor, the automaton takes a transition to location AtStart and releases the AtStart which inserts a value into the array Pos.

In location AtStart, the clock ArrivalTime is used to measure the time elapsed since the plate was detected. When it reaches to the value equal to SAFE_CALIB, it is safe to accept a new plate and return to Calib. The time constant SAFE_CALIB is obtained by experiments.

**Automaton BeltPosition.** The automaton BeltPosition is shown in Figure 9 in Appendix A. It is initially in location Idle. When there is a plate on the belt, indicated by Pos[0]>=1, the automaton moves to location Active and releases task UpdatePos.

In location Active, the task UpdatePos is released periodically to update the positions of the plates in array Pos. The period is represented by an invariant $x \leq UpdateTime$, where $x$ is again a clock. As soon as there is no plate
on the belt, which is indicated by $\text{Pos}[0] = -1$, the automaton returns to the initial location Idle.

We could also have a process to handle the light sensor (i.e. sensor 2 in Figure 3) at the end position of the feed belt to sense the arrival of a plate. In the current implementation we ignore this sensor and use $\text{Pos}[0]$ to indicate whether a plate has reached the end position or not.

5 Analysis and Code Synthesis for the Production Cell

The analysis and code synthesis presented in Section 3 have been implemented in the TIMES$^6$ tool [AFM+02]. The tool performs analysis of a system model by transformation from the model of timed automata with tasks to the model of timed automata extended with subtraction operations in the clock assignments. During the transformation, a scheduler automaton is created and composed in parallel with the other automata. Its purpose is to ensure that the released tasks are scheduled according to the chosen policy, and to indicate if a task fail to meet its deadline. For a detailed description about the encoding see [FPY02].

The parameters and tasks used in the analysis of the production cell model are shown in Table 2 of Appendix A. Note that in the production cell model, tasks execute according to the fixed priorities listed in column $P$ of Table 2. The code performed by the tasks is given in column Description and Interface and is performed by the scheduler automaton at the time-point when a task finishes its execution.

Environment Model: In order to analyse the behaviour of the production cell model, the abstract behaviour of its environment is modelled with the two timed automata Brick and Robot shown in Figure 10 and Figure 11 of Appendix A. The automaton Brick models bricks arriving on the feed belt, and Robot models the position of the robot, and how it behaves when the robot rotates. The interaction between the environment and the control program is modelled using three shared variables: DIR used to control the rotation of the robot, POS used to model the robot position, and LIGHT_2 used to sense the value of the light sensor at the start of the feed belt.

$^6$More information about the TIMES tool can be found at the web site www.timestool-.com.
Analysis: The two most important properties w.r.t. code-synthesis are schedulability and boundedness, described in section 2.3. We have used TIMES to check that the production cell model is schedulable with fixed priority scheduling in the sense that all released tasks are guaranteed to meet their deadlines in all possible runs of the system. In addition, we have checked that the ready queue of the system is guaranteed to be bounded to three, meaning that the number of simultaneously released tasks will never be more than three. We have also checked that for each task type, the bound is one, i.e. there is never more than one task instance of each type released simultaneously. A number of other correctness properties of the production cell behaviour have also been checked, e.g. that the robot will always be ready to pick up a brick before the brick reaches the end of the feed belt, and that whenever the robot tries to pick up a brick from the belt, there is a brick available. By reachability analysis, we found that the nodes W8P1 and W8P2 of automaton RobotControl are unreachable. This information has been used in the code synthesis to produce smaller code. In Appendix A.1 we list the verified properties in the input format currently accepted by the TIMES tool.
Analysis Results: The system has been verified on a machine equipped with two 1.8 GHz AMD processors and 2 GB of main memory, running Mandrake Linux. TIMES consumes 207 MB of memory and 11 minutes to perform exact analysis of the most time and space consuming property above (i.e. any property that generates the full state space, e.g. the schedulability analysis). If the same property is checked using the over approximation option of TIMES (based on the convex-hull approximation described in [DT98]), the analysis requires 13 MB and 9 seconds on the same machine.

During the analysis we found and corrected several problems in the model, e.g. the deadlines of tasks DropB, DropA, PickUpB, UpdatePos, and RdAngSen were adjusted\(^7\); the priorities of the tasks released by automaton MoveTo were adjusted to preserve the intended execution order\(^8\); the constant PREPARE_T were found to be too short; the behaviour of automaton MoveTo was modified as the boundedness analysis showed that two instances of task ReadAngleSen could erroneously be ready for execution simultaneously.

During the debugging, we often found the simulator of TIMES very useful. In particular the Gantt chart view, shown in the lower right part of the screen shot in Figure 7 proved useful for tracing the executions of the tasks. In the Gantt chart view it is illustrated how tasks are executed according to the chosen scheduling policy [AFM’02].

5.1 Generated legOS-code

We have used TIMES to automatically generate C-code that has been compiled and executed on a LEGO® RCX-brick running the legOS operating system. The code generated from the analysed model of the production cell is listed in Appendix B. The source code consists of 550 lines of C-code which results in an executable file of 4892 bytes.

The code consists of two parts: one generic run-time system (listed in Appendix B.2) that is part of any generated code for the legOS target, and one design specific part (listed in Appendix B.1). The design specific code for the production cell model is essentially an encoding of the control automata in five look-up tables, as described in Section 3. The edge table has four fields: ac–

\(^7\)Initially, all deadlines were set to 10ms. The deadlines of the tasks were changed to 13ms, 16ms, 17ms, 17ms, and 16ms respectively.

\(^8\)The task in MoveTo should execute so that task RdAngSen precedes task MvRight or MvLeft.
tive, from, to, and sync where the boolean value active indicates if the edge is currently in the list ACTIVE. The other fields are indexes into the location and synchronisation tables. The design specific part of the code also include the task bodies as part of the program text.

The information in the look-up tables is used by the run-time kernel, which basically is an implementation of the event handling procedure shown in Table 1. The main function is check_trans() that loops through the edge table and evaluates the guards of the edges marked as active. When a guard is found to be satisfied, and the edge does not have a synchronisation channel, the transition is taken, meaning that the assignments are performed (by a call to the function assign() with the edge identifier as parameter), the field active in the edge table is cleared for all edges leaving the source location and set for all edges leaving the target location. Finally, the task of the target location (if any) is released. When a transition has been taken the loop starts over, check_trans() will only return when no more transitions can be taken.

If the edge is labelled with a synchronisation channel, i.e. the sync field in the edge table contains a non-negative synchronisation identifier, a separate function check_sync() is called. This function determines, by using the table chanusage which stores back references from channel names to edges, if there is an enabled edge to synchronise with. If a synchronisation is possible the assignments of both edges are performed.

The code also includes macro and type definitions as shown in Appendix B.3.

6 Conclusions

In this article, we have shown how to synthesise executable code with predictable behaviour from high level design descriptions of embedded real-time systems. As design language, we proposed to use the model of timed automata extended with real-time tasks. For this model, it has been shown previously that design problems such as schedulability and reachability checking are decidable and can be efficiently checked using model-checking techniques.

The technique for code-synthesis developed in this paper is based on a deterministic subset of the operational semantics of extended timed automata, which is guaranteed to preserve timing, resource, and logical constraints imposed on the model. Based on the presented deterministic semantics, we have shown that executable code can be generated for a generic target platform assuming that it
guarantees the synchronous hypothesis and that the associated tasks consume their given computing times. As prerequisite for code generation, we require that the model has been analysed to be schedulable (i.e. all tasks must be guaranteed to meet their deadlines). Other analysis results can be used to further optimise the generated code by e.g. limiting the amount of allocated memory, or to exclude code segments which are guaranteed to be unreachable (i.e. dead code).

A prototype C-code generator for the legOS operating system has been implemented in the TIMES tool, and applied in a case study in which the control software of a production cell is designed. The production cell is built in LEGO® and controlled by a Hitachi H8 based LEGO® Mindstorms control brick. The design, analysis, and code generation was successfully completed in the TIMES tool, and the generated C-code was compiled and downloaded to the control brick, and executed to control the production cell.

As future work we plan to adjust the code generation to target other operating systems than legOS, such as RT-Linux. Furthermore, we consider to extend the code synthesis to also generate code for the scheduling kernel and the necessary run-time system functions. This would make it possible to generate a tailored made run-time system optimised to support only functions being used in the designed system, and further optimised based on results from analysing the design model.

Acknowledgement

The TIMES tool has been implemented in large parts by Leonid Mokrushin. The authors would like to thank him for his excellent work. We would also like to thank the anonymous reviewers of the Nordic Journal of Computing for their valuable comments on our work.

References


A The Analysis Model
Code Synthesis for Timed Automata

Figure 8: The automaton Alarm.

Figure 9: The automaton BeltPosition.

Figure 10: The automaton Brick.
Figure 11: The automaton Robot.
Table 2: Task parameters used for analysis and synthesis.

<table>
<thead>
<tr>
<th>Name</th>
<th>C</th>
<th>D</th>
<th>P</th>
<th>Description and Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Alarm</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AtStart</td>
<td>4</td>
<td>10</td>
<td>12</td>
<td>Append value to Pos.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Pos[numBricks]:=BELT_LENGTH, numBricks:=numBricks+1</td>
</tr>
<tr>
<td>Calib</td>
<td>4</td>
<td>10</td>
<td>11</td>
<td>Read a calibrated value for background light.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AlarmTaskDone:=TT, normalLight:=100</td>
</tr>
<tr>
<td><strong>BeltPositions</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UpdatePos</td>
<td>5</td>
<td>17</td>
<td>5</td>
<td>Update values in Pos.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( \forall i \text{ Pos}[i]:=(\text{Pos}[i] \geq 0?\text{Pos}[i]-1:-1) )</td>
</tr>
<tr>
<td><strong>RobotControl</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DropA</td>
<td>4</td>
<td>16</td>
<td>2</td>
<td>De-activate magnet on arm A.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TaskDone:=TT</td>
</tr>
<tr>
<td>DropB</td>
<td>4</td>
<td>13</td>
<td>1</td>
<td>De-activate magnet on arm B.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TaskDone:=TT</td>
</tr>
<tr>
<td>PickUpA</td>
<td>4</td>
<td>10</td>
<td>4</td>
<td>Activate magnet A. Remove value from Pos.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( \forall i \text{ Pos}[i]:=(\text{Pos}[i]+1), \text{Pos}[\text{BRICKS}-1]:=-1, \text{numBricks}:=\text{numBricks}-1, \text{TaskDone}:=\text{TT}, )</td>
</tr>
<tr>
<td>PickUpB</td>
<td>4</td>
<td>17</td>
<td>3</td>
<td>Activate magnet on arm B.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TaskDone:=TT</td>
</tr>
<tr>
<td><strong>MoveTo</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MvRight</td>
<td>4</td>
<td>10</td>
<td>8</td>
<td>Start motor to rotate robot rightwards.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RobotAngle:=\text{ROTATION}_1</td>
</tr>
<tr>
<td>MvLeft</td>
<td>4</td>
<td>10</td>
<td>7</td>
<td>Start motor to rotate robot leftwards.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ROB_DIR:=\text{CW}</td>
</tr>
<tr>
<td>RdAngSen</td>
<td>4</td>
<td>16</td>
<td>9</td>
<td>Read rotation sensor and convert to degrees.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RobotAngle:=\text{ROTATION}_1</td>
</tr>
<tr>
<td>StopRobot</td>
<td>4</td>
<td>10</td>
<td>10</td>
<td>Stop rotation motor.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ROB_DIR:=\text{STOP}</td>
</tr>
</tbody>
</table>
### A.1 Query File

```c
/*
Robot can become ready to pick up a brick from the feedbelt.
*/
E<> ( RobotControl.W8AtFB and Pos[0]==0 )

/*
Scheduler can execute task to pick up a brick from the feed belt.
*/
E<> ( SCHEDULER.RUN_PickUpA )

/*
Robot arm B can reach the position of the press.
*/
E<> ( RobotControl.AtPressB )

/*
Robot arm A can reach the position of the press.
*/
E<> ( RobotControl.AtPressA )

/*
Scheduler can execute task (released by robot controller) to drop brick
from arm A to the press.
*/
E<> ( SCHEDULER.RUN_DropA )

/*
Robot can never reach a state where it is waiting for press to complete,
with arm B at the position of the press. Thus, location W8P1 is dead-code
in the robot controller model.
*/
A[ ]not( RobotControl.W8P1 )

/*
Robot can never reach a state where it is waiting for press to become
ready (with arm A at the position of the press). Thus, location W8P2 is
dead-code in the robot controller model.
*/
A[ ]not( RobotControl.W8P2 )

/*
Scheduler can execute task (released by robot controller) to picked up
brick from press with arm B.
*/
E<> ( SCHEDULER.RUN_PickUpB )
```
Scheduler can execute task (released by robot controller) which drop brick from arm B on the deposit belt.

A \text{ } ! ( \text{SCHEDULER.RUN\_DropB})

The robot controller is never waiting unnecessarily at the pick-up position of the feed belt. If it is there, it is because a brick has been detected on the feed belt (but perhaps not yet arrived to the pick-up position).

A [ \] (\text{RobotControl.W8AtFB imply Pos[0]==0})

When the controller is waiting for a plate there should be no plate on arm A.

A [ ] \text{not( RobotControl.AtW8 and RobotControl.PlateOnA==TT)}

Whenever task PickUpA is executing, a brick is at the pick-up position of the belt.

A [ ] (\text{SCHEDULER.RUN\_PickUpA imply Brick.AtPickUp})

The pos of robot is always in the interval [FB\_POS, DB\_POS].

A [ ] (\text{ROTATION\_1==FB\_POS and ROTATION\_1==DB\_POS})

The variables sharedPos[0] and numBricks are correctly updated.

A [ ] \text{not( Pos[0]==-1 and numBricks>=1)}

Boundedness Analysis: The nr of simultaneously released task is bounded to 3.

A [ ] (\text{SCHEDULER.n0 + SCHEDULER.n1 + SCHEDULER.n2 + SCHEDULER.n3 + SCHEDULER.n4 + SCHEDULER.n5 + SCHEDULER.n6 + SCHEDULER.n7 + SCHEDULER.n8 + SCHEDULER.n9 + SCHEDULER.n10 } <= 3)

Scheduleability Analysis: all tasks are always guaranteed to meet their deadlines.

A [ ]
A[ ]not( SCHEDULER.error )
# Generated Code

## B.1 Production Cell Code

```c
numBricks = 0;

#define T23 22
#define T21 20
#define T20 19
#define T18 17
#define T14 13
#define T12 11
#define T10 9
#define T9 8
#define T7 6
#define T5 4
#define T2 1
#define NB_T ASK 11
#define tid_NOP tid_offset+11
#define tid_UpdPos tid_offset+10
#define tid_StopRobot tid_offset+9
#define tid_PickUpB tid_offset+7
#define tid_PickUpA tid_offset+6
#define tid_MvRight tid_offset+5
#define tid_DropA tid_offset+2
#define tid_Calib tid_offset+1
#define tid_AtStart tid_offset+0
#define tid_offset 200

BEL T_LENGTH 8
#define BRICKS 8

Transition ids
#define T0 0
#define T1 1
#define T2 2
#define T3 3
#define T4 4
#define T5 5
#define T6 6
#define T7 7
#define T8 8
#define T9 9
#define T10 10
#define T11 11
#define T12 12
#define T13 13
#define T14 14
#define T15 15
#define T16 16
#define T17 17
#define T18 18
#define T19 19
#define T20 20
#define T21 21
#define T22 22
#define T23 23
#define T24 24
#define T25 25
#define T26 26
#define T27 27
#define T28 28
#define T29 29
#define T30 30
#define T31 31
#define T32 32
#define T33 33
#define T34 34
#define T35 35
#define T36 36
#define T37 37

char release_list[NB_TASK] =
{0,1,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0};
```

---

### EDIT WITH CARE!

---

```c
#include <unistd.h>
#include <sys/time.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include "legos_definitions.h"
```

---

```c
int normalLight;
int numBricks = 0;

// Task identifiers (tid)
#define tid_BrakeEntry 200
#define tid_BrakeEntryCalib 201
#define tid_BrakeEntryDropA 202
#define tid_BrakeEntryDroplB 203
#define tid_BrakeEntryDroplA 204
#define tid_BrakeEntryMvlr 205
#define tid_BrakeEntryMvRed 206
#define tid_BrakeEntryStopRobot 207
#define tid_BrakeEntryNOP tid_offset+11

#define NB_TASK 11

// Transition ids
#define T10 0
#define T11 1
#define T12 2
#define T13 3
#define T14 4
#define T15 5
#define T16 6
#define T17 7
#define T18 8
#define T19 9
#define T20 10
#define T21 11
#define T22 12
#define T23 13
#define T24 14
#define T25 15
#define T26 16
#define T27 17
#define T28 18
#define T29 19
#define T30 20
#define T31 21
#define T32 22
#define T33 23
#define T34 24
#define T35 25
#define T36 26
#define T37 27
#define T38 28
#define T39 29
#define T40 30
#define T41 31
#define T42 32
#define T43 33
#define T44 34
#define T45 35
#define T46 36
#define T47 37

// Constant values
#define FB_POS 0
#define WAIT_POS 45
#define PRESS_B_POS 0
#define PRESS_A_POS 90
#define DB_POS 0
#define PREP_T 200
#define PRESS_T 250
#define TT 1
#define FF 0
#define RobCtrlTMPPRESS 1000
#define RobCtrlGOPRESS_T -750
#define Enter_SAFE_TIME 4000
#define Enter_CALIB 0
#define Enter_SAFE_CALIB 4000
#define UpPosUpTime 333

// Clock variables
#include "clock.h"
#include "time.h"
```

---

```c
#define time_t clock_RobCtrl_PressTime;
#define time_t RobCtrl_PrepTime;
#define time_t MoveTo_x;
#define time_t Enter_ArrivalTime;
#define time_t UpPosUpTime;

#define int RobotAngle=45;
```
void Prodcell_init() {
    RobCtrl_PlateOnB=0;
    int RobCtrl_PlateOnA=0;
    int RobCtrl_PlateInPress=0;
    Goal=0;
    int TaskDone=1;
    int AlarmTaskDone=0;
    Pos[5]={-1,-1,-1,-1,-1,-1};
    Code Synthesis for Timed Automata 145
    }
Code Synthesis for Timed Automata

```c
// Plate at the start

// Insert "brick at first non used position
Post(numBricks) =belt_length;

//mostat

// Read shared variables
normalLight = readsum/ SAMPLES;
AlarmTaskDone = TT;

// Drive the magnet on arm A
motor_a.speedoff();

// Update shared variables
TaskDone = TT;

// Drive the magnet on arm B
motor_b.speedoff();

// Update shared variables
TaskDone = TT;
```

```c
// Start robot motor moving left
motor_c_dir = motor_c.speedoff();

// Start robot motor moving right
motor_c_dir = motor_c.speedoff();
```

```c
T5T.id_NOP()// Idles/ .
T6T7.id_UpdPos //Active/ .

// Include "legos_kernel.h"

int AtStart() {
    TASKBEGIN(AStart)
    // Plate at the start
    
    // Insert "brick at first non used position
    Post(numBricks)=belt_length;
    numBricks++;
    TASKEND
    }

int Calib() {
    TASKBEGIN(Calib)
    // Take the average light value over 500 ms
    #define SAMPLES 10
    int i;
    int readsum = 0;
    extern int normalLight;
    for (i = 0; i < SAMPLES; i++) {
        readsum += LIGHT_2;
        mvalue = 50;
    }
    // Update shared variables
    normalLight = readsum / SAMPLES;
    AlarmTaskDone = TT;
    TASKEND
    }

int DropA() {
    TASKBEGIN(DropA)
    // Drive the magnet on arm A
    motor_a.speedoff();
    
    // Update shared variables
    TaskDone = TT;
    TASKEND
    }

int DropB() {
    TASKBEGIN(DropB)
    // Drive the magnet on arm B
    motor_b.speedoff();
    
    // Update shared variables
    TaskDone = TT;
    TASKEND
    }

int MvLeft() {
    TASKBEGIN(MvLeft)
    // Start robot motor moving left
    motor_c_dir = motor_c.speedoff();
    TASKEND
    }

int MvRight() {
    TASKBEGIN(MvRight)
    // Start robot motor moving right
    motor_c_dir = motor_c.speedoff();
    TASKEND
    }
```
motor_c_speed(100);
} TASK_END

int PickUpA() {
    TASK_BEGIN(PickUpA)
    int i, temp;

    // Activate the magnet on arm A
    motor_a_speed(MAX_SPEED);

    // Shift the values in the pos array.
    // Update the first (shared) element last.
    temp = Pos[1];
    for (i = 1; i < numBricks; i++) {
        Pos[i] = Pos[i+1];
    }
    numBricks--;
    Pos[BRICKS-1] = -1;

    // Update shared variables
    Pos[0] = temp;
    TaskDone=true;
} TASK_END

int PickUpB() {
    TASK_BEGIN(PickUpB)

    // Activate the magnet on arm B
    motor_b_speed(MAX_SPEED);

    // Update shared variables
    TaskDone=true;
} TASK_END

int RdAngSen() {
    TASK_BEGIN(RdAngSen)

    // The rotation sensor is update every
    // 1/16 th of a turn. Cog-wheel mounted
    // between the rotation axis and the sensor
    // give a precision of 360 degree.

    // Update shared variables
    RobotAngle = -ROTATION_1;
} TASK_END

int StopRobot() {
    TASK_BEGIN(StopRobot)

    // Stop motor moving robot.
    motor_c_dir(brake);
} TASK_END

int UpdPos() {
    TASK_BEGIN(UpdPos)

    // Update the brick positions in the position array.
    int i;
    for (i = 1; i < BRICKS; i++) {
        Pos[i] = ((Pos[i] >= 0) ? (Pos[i]-1) : -1);
    }

    // Update shared variables
    Pos[0] = ((Pos[0] > 0) ? (Pos[0]-1) : -1);
} TASK_END

int main(int argc, char **argv) {
    Prodcell_init();
    execi(&AtStart, 0, NULL, 12, SMALL_STACK);
    execi(&Calib, 0, NULL, 11, SMALL_STACK);
    execi(&DropA, 0, NULL, 2, SMALL_STACK);
    execi(&DropB, 0, NULL, 1, SMALL_STACK);
    execi(&MoveLeft, 0, NULL, 7, SMALL_STACK);
    execi(&MoveRight, 0, NULL, 8, SMALL_STACK);
    execi(&PickUpA, 0, NULL, 4, SMALL_STACK);
    execi(&PickUpB, 0, NULL, 3, SMALL_STACK);
    execi(&RdAngSen, 0, NULL, 9, SMALL_STACK);
    execi(&StopRobot, 0, NULL, 10, SMALL_STACK);
    execi(&UpdPos, 0, NULL, 5, SMALL_STACK);

    // Reset clocks
    reset(RobCtrl_PressTime);
    reset(RobCtrl_PrepTime);
    reset(MoveTo_x);
    reset(Enter_ArrivalTime);
    reset(UdpPos_x);

    execi(&controller, 0, NULL,
    PRIO_HIGHEST, SMALL_STACK);
    return 0;
}

B.2 Run-time system

// legos_kernel.h: The kernel functions that
// executes the controller.

// Determines if it is possible to
// synchronize on a channel.
// parameter: sync, channel id, i.e. index
// into the chanusage array.
// return: 0, if synchronisation is not
// possible, transition id of the
// complementary transition if it is.
int check_synch(unsigned char sync) {
    while (chanusage[sync] < NB_TRANS) {
        if (trans[chanusage[sync]].active &
            eval_guard(chanusage[sync])) {
            return chanusage[sync];
        }
        sync++;
    }
    return false;
}

// Update the list of active transitions,
// and release task of target location.
// parameter: trn, the transition taken.
void clear_and_set(unsigned char trn) {
    int i;
    // Clear outgoing transition from source
    j=trans[trn].from;
    do {
        if (trans[j][trn].active)
            break;
    } while (loc++<j[tid_offset]);

    // Set outgoing transition from target
    j=trans[j].to;
    do {
        if (trans[j][trn].active)
            break;
    } while (loc++<j[tid_offset]);
}
// Release task of target location
release_list[loc][id][tid_offset]++;

// Check if an active transition is enabled,
// and if to take it. Will continue until
// stable state (no more enabled
// transitions) is reached.

// parameter: data, unused (should be null).
// return: false, when in stable state
wakeup_t check_trans(wakeup_t data) {
    int trn, compl_trans;
    for (trn=0; trn < NB_TRANS; trn++) {
        if (trans[trn].active) {
            if (eval_guard(trn)) {
                if (trans[trn].sync - 1) {
                    if ((compl_trans =
                        check_synch(trans[trn].sync))) {
                        assign(trn);
                        assign(compl_trans);
                    } else {
                        assign(compl_trans);
                        assign(trn);
                    }
                    clear_and_set(trn);
                    clear_and_set(compl_trans);
                    trn-=1;
                } else {
                    assign(trn);
                    clear_and_set(trn);
                    trn-=1;
                }
            } else {
                clear_and_set(trn);
            }
        }
    }

    return false;
}

// Wake-up function for threads (tasks).
// parameter: data, unused (should be null)
wakeup_t task_release(wakeup_t data) {
    return release_list[data];
}

// Thread body for automata controller
// return: 0 (always)
int controller() {
    while (true) {
        wait_event(&check_trans, 0);
        return 0;
    }
}

B.3 Header file

// legosDefinitions.h: Define types and
// macros used by generated code and kernel.
typedef struct trans_s{
    char active;
    char from;
    char to;
    signed char sync;
} trans_t;

#ifdef true
#define true (1==1)
#endif

#ifdef false
#define false (0==1)
#endif

#define SMALL_STACK 128