ELENA FERSMAN

A Generic Approach to Schedulability Analysis of Real-Time Systems

UPPSALA UNIVERSTITET

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

iii
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

Figure 1: Real-Time System Components.

The expected result must be computed and delivered. 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 system 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 2: Illustration of different types of real-time systems.

In this thesis we will study hard real-time systems. A key issue in the develop-
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 [HAM+99, HLS00, EES+03].

*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.

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 $P$ 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

---

**Figure 5:** An example of a precedence graph.

**Figure 6:** Semaphore access patterns.
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.

![Figure 7: Blocking on an exclusive resource.](image)

### 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)
scheduling policy. 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

Table 1 can be executed according to the schedule shown in Figure 9 and the code for such system will look as follows:

![Figure 9: An example of cyclic schedule for the task set shown in Table 1. Minor cycle is 10, major cycle is 40.](image-url)
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 optimal 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 response 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:

\[
I_i = \sum_{j \in hp(i)} \left\lfloor \frac{R_i}{T_j} \right\rfloor C_j
\]

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

\[
R_i = C_i + \sum_{j \in hp(i)} \left\lfloor \frac{R_i}{T_j} \right\rfloor C_j
\]
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 $P_j$ 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
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.

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:

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

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:

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

where $cs_{P_{h,s}}$ 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 $P_i$, $use_s(P_k)$ 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
Introduction

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 \( n + \sum_{i=0}^{n-1} \left\lfloor \frac{D(P)}{C(P)} \right\rfloor \) 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 \( P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_n \),
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 consequentially, 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 \( \{(T_i, OP_i, A_i)\} \), where \( OP_i \) is the semaphore operation to be performed on a semaphore (i.e. locking or unlocking), \( T_i \) is the computation time for performing \( OP_i \), and \( A_i \) 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.
Introduction

References


