Diss. ETH No. 13564

On the Modeling and Evaluation of Real-Time Systems

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY
ZURICH

for the degree of
doctor of technical sciences

presented by
MARTIN NAEDELE
Dipl.-Ing., Ruhr-Universitaet Bochum, Germany
born February 15, 1972
citizen of Germany

accepted on the recommendation of
Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Manfred Morari, co-examiner

2000

Abstract

Real-time systems need to be correct with respect to both functional and temporal behavior. The use of models to validate specifications and to evaluate design alternatives is nowadays a necessity for the rapid and efficient development of complex and heterogeneous real-time systems.

Many model-based evaluation methods exist. Some use simulation, others analyze the state space or the numerical parameters of a system. Each method has its individual input model. The evaluation methods and their input descriptions are adapted to specific system characteristics and evaluation purposes. In the area of schedulability analysis, this work proposes a characterization of task arrival and processor service behavior that unifies several previous methods and extends them towards the analysis of bursty and utilization-bound behavior.

In order to evaluate all relevant aspects of a system or its respective model, several different evaluation methods need to be applied. This, however, requires multiple input models for the analysis of functional behavior, the documentation of system structure, and the validation of scheduling algorithms and timing behavior. The use of multiple input models is both error-prone, as the validity with respect to the complete system of results obtained from investigating certain views in isolation is not clear, as well as inefficient and costly, as several different models need to be built and maintained.

This work proposes a novel approach for constructing integrated system models based on Petri nets, an expressive and clearly defined visual and operational specification formalism. A pragmatic generic real-time system model structure is presented which can be used for scheduling and timing simulation as well as for formal verification of functionality. Solutions are proposed for the issues that have previously prevented Petri nets from being used for the modeling of systems with preemptive scheduling. For a model following the proposed generic structure it is shown that certain analytical statements obtained from model abstractions remain valid for the complete model.

A methodology has been developed to systematically construct models that confom to this structure. The model building process makes use of design patterns, a concept that is transfered from the software engineering domain to the area of modeling with dataflow visual languages such as Petri nets. Petri net design patterns capture behavior instead of structure and are thus orthogonal and complementary to the traditional approach of reducing model complexity by hierarchical structuring.

Zusammenfassung

Echtzeitsysteme müssen sowohl in Bezug auf ihr Zeitverhalten als auch auf ihre Funktionalität korrekt arbeiten. Die Benutzung von Modellen zum Validieren von Spezifikationen und zum Evaluieren von Entwurfsalternativen ist heutzutage eine Notwendigkeit für die schnelle und effiziente Entwicklung komplexer, heterogener Echtzeitsysteme.

Es existieren viele verschiedene modellbasierte Evaluationsmethoden. Einige von ihnen beruhen auf der Simulation, andere auf der Analyse des Zustandsraumes oder der numerischen Parameter des betrachteten Systems. Jede Methode verwendet ein eigenes, spezifisches Model als Ausgangsbasis. Die Auswertungsverfahren und die ihnen als Eingaben dienenden Systembeschreibungen sind auf spezifische Systemeigenarten und Untersuchungsziele hin ausgerichtet. Für das Gebiet der Analyse von Ablaufplänen stellt die vorliegende Arbeit eine neue Art der Beschreibung von Task-Anforderungen und Abarbeitung durch den Prozessor vor, die verschiedene frühere Ansätze vereinheitlicht und so erweitert, dass auch ein charakteristisches Verhalten analysiert werden kann, das Bursts aufweist oder durch eine allgemeine Grenzauslastungsfunktionen beschrieben werden kann.

Um alle relevanten Aspekte eines Systems bzw. seines Modells zu untersuchen müssen verschiedene Auswertungsmethoden eingesetzt werden. Dies wiederum verlangt mehrere verschiedene Datenmodelle als Eingabebeschreibungen für die Analyse des funktionalen Verhaltens, die Dokumentation der Systemstruktur, sowie die Validierung der Ablaufplanungsalgorithmen und des Zeitverhaltens. Die Verwendung mehrerer verschiedener Eingabemodelle kann einerseits leicht zu Fehlern führen, da die Gültigkeit von Ergebnissen, die durch die isolierte Untersuchung einer Systemansicht gewonnen wurden, in Bezug auf das Gesamtsystem nicht klar ist, und ist andererseits teuer und ineffizient, da mehrere unterschiedliche Modelle erstellt und gepflegt werden müssen.

In der vorliegenden Arbeit wird ein neuartiger Ansatz zur Erstellung integrierter Systemmodelle vorgeschlagen, der auf Petri-Netzen basiert. Petri-Netze stellen einen ausdrucksstarken und klar definierten visuellen und operationalen Spezifikationsformalismus dar. Eine pragmatische und allgemein anwendbare Modellstruktur für Echtzeitsysteme wird vorgestellt, die für die Simulation von Ablaufplanung und Zeitverhalten ebenso wie für die formale Verifikation der Systemfunktionalität verwendet werden kann. Für die Problembereiche, die bisher die Modellierung von Systemen mit präemptiven Ablaufplanungsverfahren mit Petri-Netzen verhindert haben, werden Lösungen angeboten. Ausserdem wird gezeigt, dass für ein Modell, das der vorgeschlagenen generischen Modellstruktur folgt, basierend auf Teilmodellen bestimmte analytische Aussagen gemacht werden können, die dann auch für das Gesamtmodell gültig sind.

Es wurde eine Methodik für die systematische Erstellung von Modellen entwickelt, die der oben angesprochenen Form folgen. Dieser Modellierungsprozess verwendet Entwurfsmuster, ein Konzept, das aus der Softwaretechnik in das Gebiet der visuellen Datenflusssprachen, zu denen auch Petri-Netze gehören, übertragen wurde. Entwurfsmuster für Petri-Netze drücken Verhalten statt Struktur aus und sind somit orthogonal und komplementär zur hierarchischen Strukturierung, die traditionell zur Verringerung der Modellkomplexität verwendet wird.

Contents

1

Introduction

Embedded computer systems can be found in many parts of our infrastructure, such as telecommunications, power generation, and transportation, or home appliances. Concrete examples include telecommunication switches, plant control and safety systems, air and spacecraft computers, air bags, engine controllers, and medical devices. It is estimated that for every Personal Computer in use world-wide thousands of embedded computer systems are deployed and that the embedded systems market is growing at around 30% per year [83]. Many of these embedded computer systems are real-time systems, which means that they have strict requirements with respect to both functional and timing correctness.

Functional correctness: This requirement is concerned with the algorithmic aspects of the system. The key question is whether the *correct result* is produced.

Timing correctness: This refers to the requirement of meeting deadlines. The key question is whether the result is produced *in time*.

1.1 Model-based system evaluation

The development, even the specification, of a real-time system like the Space Vehicle described in [137] is a very challenging task, because it involves the interaction of multiple engineering and scientific disciplines, combines deterministic computing and non-deterministic environmental stimuli, and does not leave any margins for design or implementation errors. System modeling and model-based system evaluation are important tools to deal with the different aspects of complexity encountered in the specification and design process. Throughout the development cycle, models serve different purposes: visualization and animation of functionality, analysis of properties and verification of functional and timing requirements, evaluation of performance, documentation of subsystems, components, and interfaces for the implementation and maintenance of a specific system, and documentation of domain knowledge and general mechanisms in order to facilitate reuse. Model-based system evaluation helps to explore design alternatives and to find faults in the specification already at an early stage of the design process and thus reduces development effort, time, and costs (see Fig. 1).

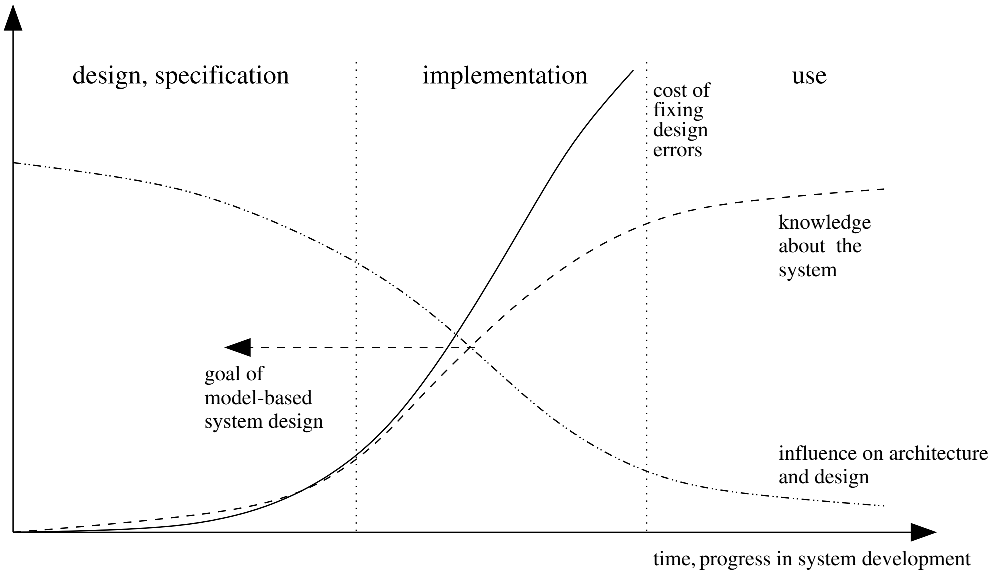

Fig. 1: Model-based system evaluation helps to gain more insight into system behavior and potential problems in earlier stages of the development process when changes are less costly.

1.2 Methods for real-time system evaluation

There are two fundamentally different approaches to the evaluation of specifications: formal analysis and simulation.

Analytical approaches are based on certain mathematical theorems and apply formulas and proof techniques to the system specification in order to obtain statements that cover every possible behavior of the system, provided that it conforms to the assumptions made. In order to be tractable, analytical approaches usually make considerably simplifying assumptions about certain aspects of the specification (e.g. [105]). Common analytical methods for the verification of real-time systems are: schedulability analysis, model checking, stochastic and queuing theoretical analysis.

The basic idea of the simulation approach is to execute the system model, possibly several or many times, and to analyze the resulting execution traces. Simulations may be conducted with respect to, e.g., functional behavior, meeting of deadlines (scheduling simulation), or throughput (performance simulation). In case a system contains considerable non-determinism, the union of all execution traces obtained by simulation will only cover a part of the possible behavior of the system.

Simulation, in contrast to formal analysis, can thus not be used to prove all-quantified statements about certain properties ("the property is always/never valid"), as the counter-example may be hidden just in that part of the possible behavior that was not covered by the simulations conducted [114]. Simulation can only support existence-quantified statements ("there exist situations in which a certain property is/is not valid"). Nevertheless, even for the development of real-time systems, there are evaluations settings in which simulation is an appropriate approach [83]:

1. Once the existence of a problem has been detected by analytical means, simulations can be used to discover the sequence of events that leads to the occurrence of the problem and guide the designer towards a better solution.

2. For performance analysis usually the major interest is in the typical behavior. This can be obtained from simulation even for systems for which the specification is too complex for a stochastic, queuing theoretical analysis. Both for throughput analysis of manufacturing and computer systems as well as for the rejection rate analysis of real-time guarantee algorithms simulation is a well-established technique [130, 54, 145, 65].

3. An executable, animatable specification is useful to validate system requirements [177, 114], that is, not to prove that the system behaves correctly but to investigate whether it serves the right purpose (prototyping).

4. Certain systems may be too complex (e.g. due to overhead, overload, interaction with a dynamic environment/plant for which no analytical model exists, conditional dependencies, distributed systems, imprecise computing, urgency dependent selection of alternative tasks, runtime modification of the system itself, fault injection and simulation) for the analytical means available today, so that, though a formal analysis would be more desirable, simulation of the specification is the only way to gain at least some confidence in the correct behavior of the systems under construction [25, 113, 178].

1.3 Problems of model-based evaluation

In the course of a system development several of the methods presented in the previous section need to be applied to evaluate different aspects of the specification, each with its own system model as input. This is in contrast to the goal of model-based evaluation to get a maximum of insight from a minimum of modeling and implementation effort.

After more than two decades of research that focused on very specific problem settings and isolated methods, in 1996 a group of leading real-time researchers identified the need for a new approach:

> "There are many techniques developed and used in practice for improving reliability, ranging from static analysis based on formal methods and scheduling theory to dynamic analysis based on testing and run-time monitoring/checking. Although these techniques are quite effective when applied independently to small scale systems, **it is important to have a common framework on which these different techniques can be uniformly applied. With a common framework, one model of the system is specified and then used for both static and dynamic analysis. This would save the efforts of developing and specifying models since each technique requires its own model of the system. Furthermore this would make it unnecessary to check consistency between models used by different techniques since there is only one model.**" [83]

This work proposes a model-based approach to the evaluation of real-time systems which uses one integrated, common behavioral model of the system under development for both formal verification methods and evaluation by simulation. From the modeling purposes stated at the beginning of this chapter, a set of requirements for the common real-time modeling framework can be deduced [137]. The model and its underlying formalism should

- be executable for (animated) simulation,
- have clearly defined semantics for analysis,
- be able to express control and data flow as well as data transformations,
- have a notion of time to express timing parameters and constraints,
- be able to represent the system structure, as reuse will most likely happen along structural and subsystem boundaries,
- provide abstraction mechanisms such as hierarchy,
- provide means for visual input of the specification and visual output of the evaluation results, and
- not be biased towards a certain application or engineering domain.

Any actual modeling formalism to be used as specification language in the context of such an integrated modeling approach must be able to express the specific characteristics of real-time systems and it must be usable and practical for complex, large scale models. This motivates and necessitates research on the following topics:

Abstraction: New structuring and abstraction mechanisms are necessary which facilitate the reading and understanding of complex visual models.

Construction: A methodology for systematic model construction must be developed which guides the domain expert towards an analyzable and simulatable model.

Structure: A generic model structure needs to be found that facilitates the application of formal verification methods to (parts of) complex models and also allows to represent state-of-the-art, that is, preemptive scheduling strategies in the model.

Validity of analysis: The proposed methodology is based on a common, integrated model which contains all information about the system. Certain formal analysis, however, can only be conducted on parts of this model or requires an abstraction to a less expressive formalism. It must thus be investigated to what extend and which results from the formal verification of analyzable submodels are still valid for the complete system.

Accurate schedulability analysis requires that the task specification and scheduler abstractions used as analysis input correspond closely to the actual system, otherwise the results are unreliable or lead to over-dimensioned and thus unnecessarily expensive systems. Early theoretical results on schedulability [105] made very restrictive assumptions about the task arrival behavior, such as strict periodicity, and a lot of effort was invested to make actual tasks fit into this mould [142]. Practical model-based system evaluation, however, requires models that address realistic situations. Recent research, e.g. [112], focuses on the development of new analyzable task models to capture the characteristics of tasks with more realistic arrival behavior. One class of arrival behavior that has not yet been thoroughly investigated are bursty tasks and tasks limited by utilization bounds. Task arrival limited by utilization bounds means that sporadic tasks are not described using worst-case execution times and minimum interarrival times for each arrival but are defined in terms of the maximum computational capacity needed within certain time spans, independent of the actual number of arrivals that request this capacity (see Fig. 2). Also, processor capacity needs not be constant over time in the general case and can be described in a similar way.

Examples for the occurrence of these characteristics are multi-media applications and distributed, network connected systems. As these application domains will become more and more important in the near future the mathematical characterization and schedulability analysis of such systems is of high interest.

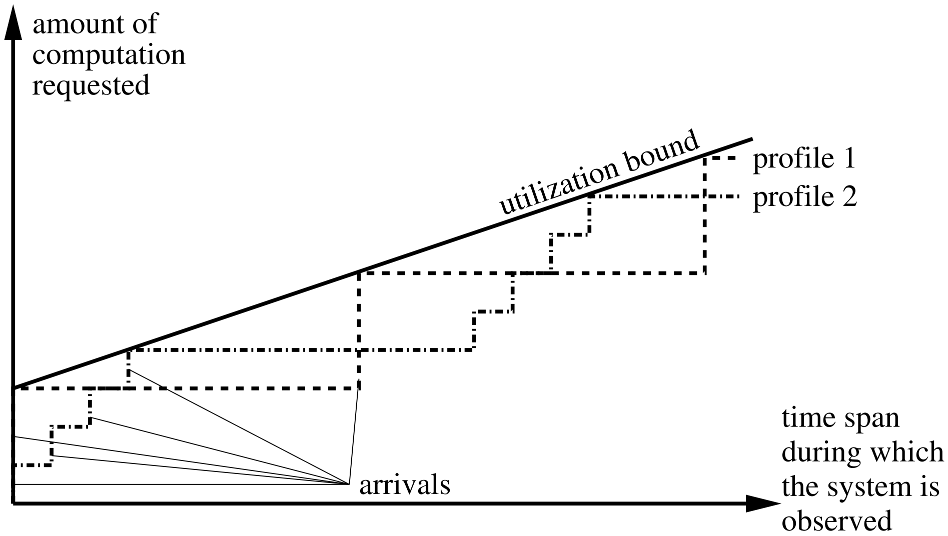

Fig. 2: Two different task arrival profiles which are characterized by a single utilization bound function.

1.4 Main results

This work addresses the research issues on modeling and analysis raised above. For model construction, Petri nets have been selected as the common modeling formalism. The following results have been achieved:

Modeling issues

- The concept of design patterns for dataflow visual languages such as Petri nets, has been introduced as a new abstraction to conquer model complexity. Design patterns focus on functionality and behavior and are thus orthogonal to other abstractions such as spatial ordering and hierarchy.
- A pattern-based methodology for the systematic construction of formal models from real-time domain models (task models) has been developed.
- A novel model structuring style, delegated execution, is proposed which allows to separate the model parts that require expressive formalisms with token data processing and time delays from the submodels that represent the basic functionality of the system and that can be modeled with restricted, analyzable Petri nets.
- based on the delegated execution model structuring the problem of modeling general preemptive scheduling situations in a scalable way has been solved for the first time.

Analysis issues

- The relationship between different model parts and the influence of the use of submodels of different expressive power on the semantics of the whole model have been investigated and the types of analytical statements that can be made from the analyzable submodel have been characterized and proven.
- For schedulability analysis a task and processor model especially suited to bursty tasks and processors with variable performance has been developed and investigated. This model unifies and extends previous task models.

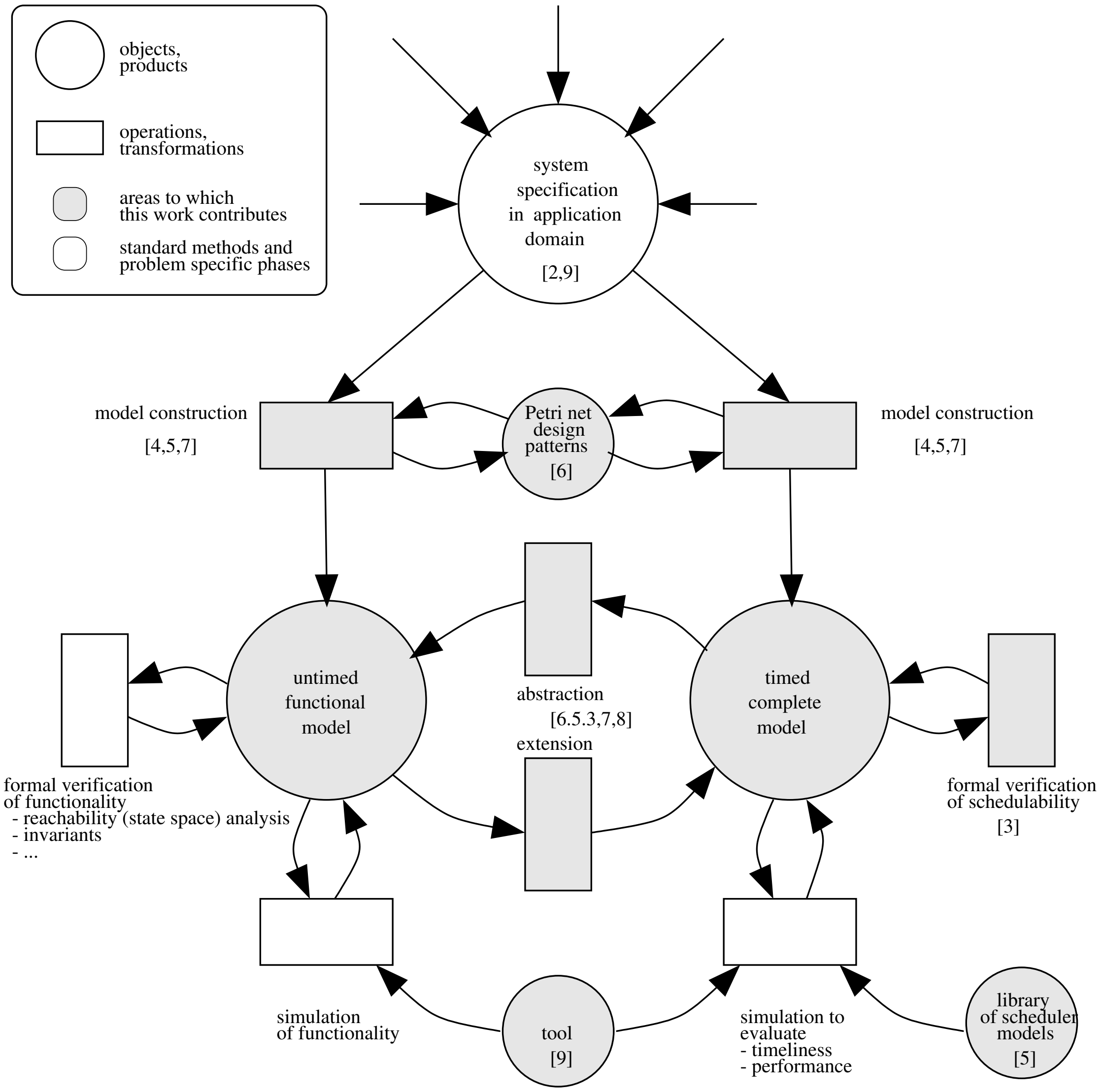

Fig. 3: Phases in the system modeling and evaluation process to which this work contributes. The numbers denote the corresponding sections of this monograph. Note that the methodology allows either to derive the complete system model as an extension of a previously constructed functional model or to go directly to a complete model which is then the basis for abstractions which are evaluated with respect to functionality.

1.5 Overview

Fig. 3 shows the stages of the system modeling and evaluation process and highlights the areas to which this research contributes. The numbers indicate the chapters of this monograph which treat the corresponding topic. The structure is as follows:

In Chapter 2 real-time systems and their relevant attributes are introduced. An overview of state-of-the-art scheduling and resource access algorithms for real-time systems is given, and previous work on modeling and evaluation of real-time systems by formal methods and simulation is discussed.

In Chapter 3 a task model for the schedulability analysis of bursty tasks and processors with variable speed is defined, some properties are proven, and a general schedulability test is derived.

The use of Petri nets as common modeling formalism is motivated in Chapter 4 and a short introduction to the Petri net formalisms used in this work is given.

Chapter 5 defines the general preemptive scheduling problem for Petri nets, explains the problems it raises and presents a novel solution which uses the delegated execution modeling style. Some example models of preemptive real-time scheduling algorithms are also given in this chapter.

In Chapter 6 design patterns and their applications to dataflow visual languages are introduced. A set of general Petri net design patterns and their dependencies are identified and the occurrence and relevance of these patterns in published system models is investigated. A design pattern template for dataflow visual languages is presented and used to document, as examples, five design patterns from different levels of abstraction, some of them specific to the real-time system application domain.

The system evaluation methodology in Chapter 7 addresses the aspects of systematic model construction through patterns, formal analysis of functionality, and evaluation of timing properties by simulation.

Chapter 8 investigates the validity of formal verification results obtained from submodels and abstractions of a common model constructed according to the modeling approach proposed in this work.

In Chapter 9 the application of the methodology for model construction and evaluation is demonstrated by means of the mine drainage system example which has been used previously in other works on modeling methodologies for real-time systems.

Chapter 10 summarizes the results achieved and suggests areas for further research.

2

Real-time systems

The characteristic feature of real-time systems is that their specifications impose strict requirements with regard to the timing of their actions. Real-time systems usually consist of a set of tasks with individual timing constraints and certain dependence relations between those tasks. Scheduling algorithms are used to ensure, either during the system design stage or online during system execution, that all timing and dependence constraints are actually met when the system is operating.

The following sections of this chapter explain the main attributes that are characteristic for real-time tasks, give an overview of standard real-time scheduling algorithms, and discuss some previous work on specification and evaluation methods for real-time systems.

2.1 Hard, firm, and soft real-time systems

Frequently, 'real-time' is associated with 'fast'. While performance and throughput may be important factors for real-time systems, too, especially in communications applications, the requirement for high throughput does not constitute a real-time system, only the importance of meeting timing constraints does [150].

Real-time tasks are classified into three categories according to how serious, with respect to the system purpose, a violation of a timing constraint is:

Hard real-time tasks must meet their timing requirements under all circumstances, otherwise a catastrophe, within the scope of the application, will be the consequence. A car airbag is an example for a hard real-time system.

Soft real-time tasks provide the less benefit to the system the more they miss their deadlines, but the consequence is never catastrophic [56]. Examples for soft real-time applications can be found in the communications and multimedia area, e.g. video conferencing.

Firm real-time tasks may, e.g. due to system overload, be rejected without consequences straight away when they arrive. If a firm task is guaranteed after arrival, it becomes a hard task.

Utility functions, see e.g. [32], can be used to express these three categories mathematically and graphically and to construct even more specialized task types.

2.2 Task parameters

Usually the code associated with a task will be executed multiple times while the system is operating. Each of these executions is termed a task *instance* or *job*. The term task is used to designate the set of attributes that are valid for each instance.

The most crucial task parameter, the one that turns a task into a real-time task, is the *deadline* of the task. The deadline, which is specified either *relative* to the task's arrival time or as an *absolute* point in time, determines the upper bound on when the task's execution has to be completed, taking interference from other parts of the system into account.

The *response time* of a task is the time span it actually takes from task arrival until task completion within the system environment. A task completes in time if its maximum response time is shorter than its relative deadline.

In order to compute the response time of a task it has to be known how long its execution would take under ideal conditions, that is, without any interference from other tasks. This parameter is the *worst-case execution time* c_{wcet}. For some scheduling algorithms it is also necessary or helpful to know the minimum and average task execution times.

The *slack* of a job is the amount of time a job may be delayed and still complete in time. Thus the slack is the difference between completion time (arrival time + response time) and absolute deadline. If the slack is negative, the deadline has been missed.

Depending on the application, the real-time requirement might also include requirements for fault-tolerance mechanisms, such as multiple, redundant task executions (e.g. [65]).

2.2.1 Periodic, aperiodic, and sporadic tasks

Tasks are categorized according to the arrival characteristics of their jobs:

Periodic tasks have jobs that arrive with regularly spaced intervals in between. The length of the time between arrivals is called the *period*. The time until the first job arrives after system start is called the *initial phase* of the task. In realistic systems even periodic tasks have a certain *jitter* with respect to the arrivals of their instances. Usually the executions of different instances of the same periodic task do not overlap, and frequently the relative deadline is equal to the task period. Typical periodic tasks are sensing and computation tasks of digital control systems.

Aperiodic tasks have jobs that can arrive at any time during system execution. Typical aperiodic tasks are event handlers reacting on stimuli from the environment.

Sporadic tasks are aperiodic tasks with a specified *minimum interarrival time*, so that the worst-case load induced by these tasks can be calculated.

Whereas the arrival of a job of an *independent task* is determined by an external event or the progress of time only, the start of a job in a task system with *precedence constraints* between tasks also requires the completion of its predecessor tasks.

2.2.2 Urgency, priority, and importance

Most real-time scheduling algorithms rely on a concept of task *priority* to determine the execution sequence of pending jobs. The priority of a task is a scalar metric - by convention a lower numerical value indicates a higher priority - for use by the scheduler to establish an order on the jobs waiting to be executed. Several factors influence the priority a job has: its *urgency* is a measure for how close it is to its deadline, while its *importance* (criticality in [157]) is derived from its role in the application at a problem domain level. An important task with a far out deadline and thus low urgency may therefore have a lower priority than an urgent but unimportant task. The distinction between tasks that are important and those that are merely urgent becomes significant when dealing with overload situations [142].

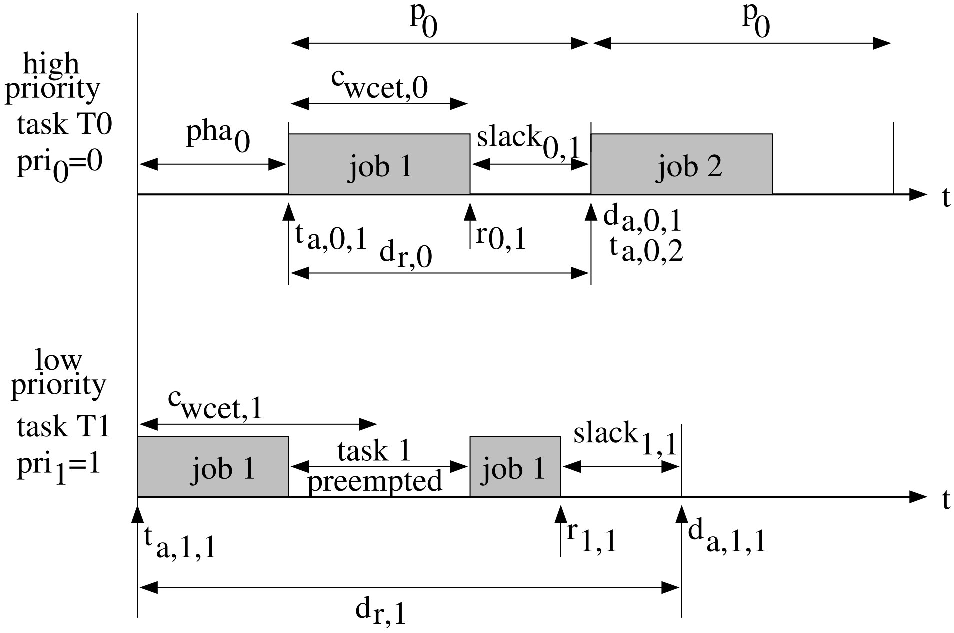

Fig. 4: Example for real-time task parameters and timeline presentation.

Ex. 2.2.1: *Fig. 4 shows a part of a timeline diagram (Gantt chart) of the execution of two tasks, Tab. 1 explains the abbreviations used. One sees two instances of the periodic task T_0 with period $p_0 = 50$ which has the higher priority $pri_0 = 0$ of the two tasks. Its initial phase is $pha_0 = 25$, which is by definition identical with the arrival time of its first instance $t_{a,0,1}$. The relative task deadline of $d_{r,0} = 50$, equal to the period, results in an absolute deadline for the first instance of $d_{a,0,1}$=75. As this is the task with the highest priority in the system, its response time $r_{0,1} = 30$ is equal to its execution time c which is here assumed to be equal to the worst-case execution time $c_{wcet,0} = 30$. Due to the arrival of the higher priority task, the instance of task T_1 which had arrived and started earlier*

Abbreviation	Explanation
pri_i	priority of task T_i
pha_i	initial phase of task T_i
p_i	period of task T_i
$c_{wcet,i}$	worst-case execution time of task T_i
$t_{a,i,j}$	arrival time of job j of task T_i
$d_{a,i,j}$	absolute deadline of job j of task T_i
$d_{r,i}$	relative deadline of task T_i
$r_{i,j}$	response time of job j of task T_i
$slack_{i,j}$	slack of job j of task T_i

Tab. 1: Explanation of the task parameter abbreviations used in Fig. 4.

is interrupted (preempted) at time 25 and resumes execution at 55. Its response time $r_{1,1} = 70$ *is thus longer than its pure execution time.*

2.3 Scheduling

The explanations in the following sections refer mostly to the scheduling of single processor systems. Even though some of the algorithms have been extended to the multiprocessor case, the latter is much more complicated and less theoretical results exist.

2.3.1 Scheduled systems versus cyclic executives

There are two principal approaches to the design of real-time executables [107]:

The *cyclic executive* approach [29, 107, 11] is used to build systems that consist only of purely periodic tasks. The code of all tasks is combined using a fixed sequence of procedure calls which is executed within a single cyclical process. A cyclic executive system has two global timing attributes: the major and minor cycle lengths. The start time of each minor cycle is enforced by an external timer. The length of the minor cycle, or frame, will usually be identical to the period of the most frequent task in the system. Less frequent tasks are only executed every n-th minor cycle according to their required period. This requires the periods of all tasks in the system to be harmonic. In case the load allocated to a certain frame is larger than the length of the frame, some less frequent tasks need to be divided into slices that are distributed throughout several minor cycles. The major cycle is the least common multiple of all task periods. While the code executed during consecutive minor cycles will usually be different due to the slicing of the tasks with non-minimal periods, each major cycle is absolutely identical. One major cycle corresponds to one complete execution of the fixed sequence of procedure calls mentioned above.

The advantages of this approach are that it is conceptually simple, needs no kernel, allows exact, deterministic control of which job is executing at a certain point in time which also allows to do without explicit protection of shared resources, and incurs practically no run-time overhead for scheduling and inter-task communication. On the down-side, the requirement of harmonic task periods is restrictive, the exact slicing of the tasks is very labor intensive, and this work needs to be repeated whenever any one task implementation changes. Added problems arise from the fact that a basic assumption of this approach is that the actual task execution times are known exactly, and that the system goes into an undefined state after a frame overrun has occurred.

In a *scheduled system* a runtime instance, the scheduler of the operating system, controls the invocation order, invocation times and possibly (see next section) interruption of tasks. The advantages of scheduled systems are that each task is dealt with individually. Due to this modularity design and maintenance are easier. The scheduler as control instance can stop and remove tasks that have run out of control and thus increases the reliability of the system. It can also react flexibly on system and environment changes at run-time. Disadvantages are that some load is added for the execution time of the scheduler, context switching etc, and that, though the relevant behavior, that is, the meeting of deadlines is of course guaranteed [107], the actual execution sequence of jobs is not always deterministic and predictable.

Many different scheduling algorithms exist [152]. Some basic ones will be presented in the next sections. One main distinction is whether the execution sequence is already fixed at design time (*offline scheduling*) [176] or whether events during the execution can still influence the sequencing (*online scheduling*) [153].

2.3.2 Preemptive and non-preemptive scheduling

If the scheduler makes a decision about the next task to run only after the currently executing job has finished this is called *non-preemptive scheduling*. With *preemptive scheduling* the scheduler evaluates the set of jobs ready to run after each relevant event, such as new task arrivals, resource releases, and task completions. This evaluation may result in the currently executing task being stopped - preempted - and another task to be run instead, as shown in Fig. 4. Preemption may either be *resumable* or the preempted job is aborted and has to be restarted completely. Due to the higher loads that can be guaranteed to meet their deadlines, the theoretical results that support design and analysis (see Sections 2.3.3 and 2.4.1), and the fact that many commercial real-time operating systems such as pSOS[1], QNX[2], VxWorks[3] support it, preemptive scheduling is the standard technique to be used for single processor systems. Multiprocessor scheduling in the general case, that is, allowing arbitrary computation times or task ordering is already NP-complete for the non-preemptive case [152]. Using preemptive scheduling would further increase the complexity and difficulty of this problem, thus multiprocessor real-time systems are mostly scheduled non-preemptively.

2.3.3 Fixed and dynamic priorities

As has been stated above, the scheduler uses priorities to determine the next job to be scheduled. Depending on whether the priorities of the tasks are determined at design time and never changed during execution or whether the priority of each job is recalculated at each scheduling event and may change even during the execution of one job, one distinguishes between *fixed* and *dynamic* priority scheduling.

The standard techniques for assignment of fixed priorities to periodic tasks are rate monotonic scheduling (RMS) [105] and deadline monotonic scheduling (DMS) [101, 10] and their extensions [142, 99, 72, 92]. See [7] for an overview. With these algorithms the task priorities are assigned according to the inverse of the task periods or relative deadlines respectively, with the task with the shortest period/deadline obtaining the highest priority.

[1] www.isi.com
[2] www.qnx.com
[3] www.wrs.com

Dynamic priority scheduling in the form of earliest deadline first scheduling (EDF)[105] has recently been rediscovered [153] and the first EDF based operating systems are appearing on the market now [124]. With EDF scheduling, at each scheduling event the priorities of all jobs are recalculated according to their absolute deadlines.

2.3.4 Combining periodic and aperiodic tasks

Periodic tasks can be scheduled at design time. Sporadic tasks can be scheduled by making the wasteful worst-case assumption that the period is equal to the minimum interarrival time. The arrival of aperiodic tasks is by definition not known a priori. Consequently, service for aperiodic tasks must be somehow fitted into the planned schedule of the periodic tasks. Several different algorithms have been proposed to serve aperiodics within a schedule of periodics managed by a fixed or dynamic priority scheduling algorithm: they include background server, polling server, deferrable server [100, 158], priority exchange [100], sporadic server [149], slack stealing [98, 133], total bandwidth server [33], to name just a few. For a rather comprehensive overview and explanation see [32]. The main idea is to use the idle time of the periodic schedule and to shift around the scheduled periodic jobs within their slack so that they still meet their deadlines but the idle time to serve an aperiodic is available as early as possible after the arrival of that aperiodic.

2.3.5 Accessing shared resources

Frequently tasks need to share access to resources such as memory locations or system services. The code section in which a task accesses a shared resource is called a *critical section*. No two jobs may be inside a critical section with respect to the same resource at any time during execution. When using preemptive scheduling this means that a higher priority task may not be able to preempt a lower priority task holding a resource the higher priority task needs as well. The higher priority task is blocked on the lower priority task. While this is in general unavoidable, the situation is aggravated once a third job with intermediate priority and no resource access arrives. It preempts the lower priority task and thus blocks the higher priority task for a possibly unbounded time. This situation is called *priority inversion*. To cope with it, several resource access protocols have been developed: e.g. priority inheritance protocol [143], priority ceiling protocol [143], and stack resource protocol [12]. For an overview with algorithm explanations see [32]. The general idea is to raise the priority of the blocking task to the level of the blocked task so that the intermediate priority task can no longer preempt the blocker.

2.4 Real-time system evaluation

2.4.1 Scheduling analysis

The seminal work on real-time schedulability analysis for single processor systems is [105]. The decision about schedulability of a system of periodic, independent, fully preemptive tasks with deadlines equal to their periods, without shared resources or scheduler overhead, and with priorities assigned according to the inverse of task periods for the fixed priority case (rate-monotonic scheduling) and according to the 'earliest deadline' for the dynamic priority case is made based on a simple formula involving the number of tasks and their respective partial loads. Both priority assignment schemes were shown to be optimal for the fixed and dynamic case respectively within the given set of assumptions.

Since then, most research has concentrated on relaxing some of the original assumptions for the fixed priority case, such as the relation between deadlines and periods [101, 97, 10, 59], allowing shared resources [143], aperiodic tasks [100, 149, 98, 159, 158], scheduler overhead [89, 28], communications between tasks [142, 144, 162], and variable task arrival patterns [1, 16, 112, 121]. Starting from [74] several groups developed complex iterative formulas taking the consequences of all these complications into account [99, 161]. Some of the results have then been adapted to the dynamic priority case as well [40, 41, 37, 12, 69, 78, 61, 33]. [92] summarizes much of the state of the art in the design and analysis of fixed priority systems and [7] is a detailed historical survey. [153] gives an overview of the work done on dynamic priority scheduling.

Most research on scheduling analysis has been done in the area of preemptive single processor scheduling. Scheduling of multi-processor or distributed systems, even non-preemptively, is NP-complete for all practically relevant situations [152], so mostly heuristic algorithms are used for scheduling and simulations to evaluate these algorithms [151, 179, 131, 130, 174, 160, 79, 145, 39, 65, 109, 117].

Only little work has been done on non-preemptive single processor scheduling (e.g. [86]), because in the general case the workload that can be scheduled is smaller than for preemptive scheduling, and the theoretical results achieved for the more efficient preemptive scheduling are actually practically applicable.

2.4.2 Tool support for scheduling analysis

While formal analysis gives theoretically reliable results, practical systems frequently do not meet the assumptions upon which the scheduling analysis methods are based, and real system configurations are too complex to be formally verified. For this reason most tools to support the design of real-time systems rely on a simulator or a combination of simulator and analyzer.

The first tool to offer support for the calculations of rate-monotonic scheduling analysis and simulation was Scheduler 1-2-3 [164, 163]. The status of the simulator component of this tool set is not clear from the publications. STRESS [9] is mainly a simulator for the experimentation with new scheduling algorithms. It has some analysis functions for standard scheduling algorithms. CAISARTS [82] is a rule-based expert system for schedulability analysis. Its main feature is the ability to provide context-sensitive advice for design improvements. The current status is unknown. [95] is a C^{++} framework for scheduling simulation. [173, 67] describe another simulator for the evaluation of scheduling algorithms. The PERTS tool set [154, 147, 106, 134, 156, 155] consists of a schedulability analyzer (PERTS) and a simulator (DRTSS/PERTSSim). PERTS is comprehensive with respect to standard algorithms, but not extensible. With the live task model of the DRTSS simulator, tasks and resources are represented by C^{++}-classes and contain real code. The PERTS tool set is now maintained and extended as a commercial product[4] [166, 167].

Another model editor, analyzer, and simulator suite is described in [4, 3, 5]. The simulator GHOST [141] also seems to be part of this project. ASSERTS [64] is a simulation environment for distributed real-time systems. TimeWiz[5] is a commercial tool without simulation component for system design based on rate-monotonic and dynamic priority analysis. The current status of perfoRMAx[6], another commercial scheduling analysis tool, is not clear.

[4] www.tripac.com
[5] www.timesys.com
[6] www.aonix.com

STRESS, the framework of [95], [173, 67], GHOST, and ASSERTS use textual pseudo code to structure the task bodies. While this allows for a limited modeling of functionality, the pseudocode is not intended to model actual data processing but just to declare the processing and the input/output sections of the tasks to be analyzed/simulated. The live task model of DRTSS requires the actual task to be implemented in full detail.

Most of the non-commercial tools named above have been discontinued and/or are not publically available, PERTS/DRTSS being the notable exception. For a more detailed description of the individual scheduling analysis and simulation tools see [118], a comparison and critique of the mentioned commercial tools can be found in [137].

2.4.3 Functional specification

The most popular visual and executable specification languages for functional system modeling are Specification and Description Language (SDL) [36], the StateChart [73] and ROOM (UML/RT) [139] extensions of the finite state machine formalism, and Petri nets with their various derivatives. Among the modeling formalisms presented here, only Petri nets have clearly defined semantics that include data transformations and a concept of time [172, 55, 128]. In the following, a brief overview of some of the works that use Petri in the context of real-time and embedded system development is given. Due to the huge amount of published work in this area, the survey can not be nearly complete. However, an attempt has been made to present a representative selection.

Petri nets are used for the visualization and evaluation of system functionality alone, without any scheduling or timing information, in [51, 14, 52, 127]. Tools (PROTOB/Artifex[7], CodeSign/MOSES[8]) based on object-oriented high-level Petri net variants are employed for structural modeling in [15, 53].

[62, 63] describe a net formalism and analysis method to formally verify the existence of a non-preemptive schedule for a modeled task system. The approach neglects shared resources and the analysis encounters certain anomalies due to the specific Petri net firing semantics and modeling style used.

With generalized stochastic Petri nets [110, 49] a combination of formal analysis and simulation is used to evaluate the performance of complex systems. Exponentially distributed stochastic firing times permit to model preemption [110, 180] exploiting the memoryless property of the exponential distribution, but at the same time limit the applicability of this approach to the analysis of real-time systems to specific problem settings [125, 93].

Petri nets with deterministic firing times are used in [25, 168] for the validation of the timing requirements of a system specification on a high level of abstraction. On this level, neither scheduling nor resource access issues are dealt with.

Different scheduling policies are applied to a Petri net model in [13, 104, 24] by means of scheduler specific modifications of the Petri net execution semantics which are not transparent to the modeler and for formal analysis.

In [122] the scheduler is an explicit and integrated part of the system model. The resulting structural complexity is only manageable because the scheduling is non-preemptive and there are no task dependencies and shared resources.

Ada programs are transformed into Petri nets for automated static analysis in the approach of [34]. The formalism used is similar to time Petri nets with a

[7] www.artis.it
[8] www.tik.ee.ethz.ch/~moses

deterministic firing delay. After applying net reduction rules to the task graph the state space is generated and can be used to answer queries about the system behavior, in particular with respect to deadlock detection.

A generic approach to the modeling of fault-tolerant real-time systems with a hierarchical timed Petri net formalism is presented in [38]. The model specifies timing constraints for analysis as well as run-time error handlers for timing error during system execution. A generic transform to add redundancy to a system model is introduced. The methodology is targeted towards coarse-grained modeling of distributed systems, scheduling and resource access issues are not dealt with.

2.4.4 Other formal specifications of functionality

A survey of several additional tools and languages for formally specifying real-time systems, such as algebraic specifications and logical expressions, can be found in [26]. Because they are not visual and/or not executable they will not be further investigated in this monograph.

An interesting integrated approach for functional and scheduling verification based on a process algebra is reported in [94]. The process algebra allows dynamic priorities and data processing. From the specified system a set of predicate equations is obtained by bisimulation and these equations can be solved using linear or constraint programming. The solution is then used to determine the timing parameter values (periods, start times) which make the system schedulable. However, it is not yet clear whether all kinds of real-time systems can be specified and verified with this approach, and also the specification process and the resulting models are unintuitive enough to be serious hurdles for practical application.

In this chapter the main terms to describe real-time tasking situations have been introduced and some standard scheduling algorithms as well as schedulability analysis methods have been presented. Due to the simplifying assumptions and constraints imposed by individual schedulability analysis methods each is only applicable for a certain specific tasking situation. In the next chapter a new method especially suited to the characterization and analysis of bursty tasks and processors will be described.

3

Schedulability analysis for bursty tasks and processors

Real-time systems can be evaluated by analytical means or simulation. The accuracy of analytical results depends strongly on the match between the characteristics of the system or its model, respectively, and the assumptions made by the analysis method that is used. In this chapter a new analytical approach to schedulability analysis is presented that is particularly well suited to bursty tasks and processors with bursty service characteristics.

3.1 Previous task models

Every method for formal schedulability analysis of real-time systems relies on certain mathematical models for the amount of processing capacity requested by arriving tasks over time, R(t), and the amount of processing capacity the processor can provide, C(t), (Fig. 5).

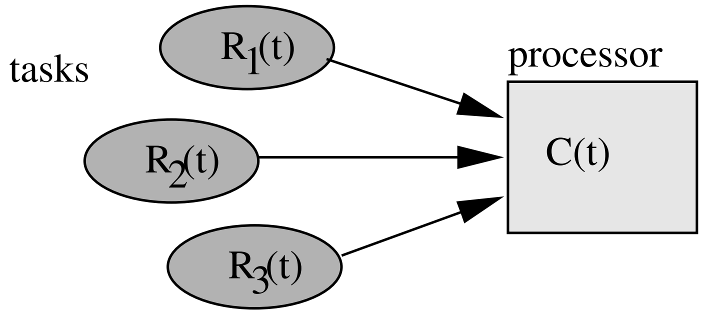

Fig. 5: The input parameters for schedulability analysis are the capacities requested by each task, $R_i(t)$, and the capacity provided by the processor, $C(t)$.

The differences between the various algorithms proposed for schedulability analysis lie mostly in the assumptions made about the shapes of the functions R(t) and C(t), usually with a stronger emphasis on variations of R(t), to capture specific a priori knowledge about the arrival characteristic of a certain task:

The Hyperperiodic Task Model [1] allows a task to be executed more frequently than its maximum interarrival time prescribes, if the necessary processing capacity is available. The General Real-Time Task model of [112] describes situations where regularly every n-th release of a task is known to have considerably higher worst case computation demands than the others. For this particular situation Mok and Chen derived a schedulability test that allows a higher utilization than the well-known test of [105].

Of special interest are models for bursty tasks. These are tasks for which several releases may follow upon each other with a short minimum interarrival time T', but of which it is known that only a certain amount of computation will be requested within a certain longer period. An example for such a bursty task in a satellite control system is given in [8]. Further applications can easily be found in distributed control and multimedia systems. [69] informally introduces a task model for bursty tasks based on event streams and gives algorithms to analyze the schedulability of distributed real-time systems using this task model and earliest deadline first scheduling. The task model presented in this chapter is a general model that integrates those mentioned above. In contrast to the sporadic periodic tasks of [8] the minimum allowed interarrival time need not be fixed, but can be stated in terms of utilization bounds.

In analogy to bursty task arrival also the computational capacity of a processor, C(t), can be considered as bursty from the perspective of a particular task as a consequence of influences of e.g. DMA processing in the background or dead-times caused by the execution of higher priority tasks. It turns out that bursty processor service and task arrival can be treated as mathematically dual.

3.2 Terminology

This section defines the terminology used further on in this chapter and introduces an example which will be extended later.

delivery rate s The nonnegative delivery rate $s(t)$ is a measure for the rate with which task execution progresses, e. g. instructions per second.

request rate r The request rate $r(t) \geq 0$ is a train of impulses scaled with the amounts of computation e, requested by the respective task releases.

accumulated request function R The accumulated request function $R(t) = \int_0^t r(\tau)\,d\tau$, $R(0) = 0$, $R(t) \geq 0$, is a wide-sense increasing function that represents the total amount of computation that has been requested (by a task or from a processor) up to time t.

accumulated capacity function C The accumulated capacity function $C(t) = \int_0^t s(\tau)d\tau$, $C(0) = 0$, $C(t) \geq 0$ is a wide-sense increasing function that represents the total amount of computation the processor could deliver up to time t if it is running at full load.

accumulated computation function R′ The accumulated computation function $R'(t) \geq 0$, $R'(0) = 0$, is a wide-sense increasing function that represents the total amount of computation delivered (from a processor or to a task) up to time t as result of one particular accumulated request function R.

accumulated unused capacity function C′
The accumulated unused capacity function $C'(t) \geq 0$, $C'(0) = 0$, is a wide-sense increasing function that represents the total amount of computational

capacity that was not used up to time t as a consequence of the processor not running at full load.

Fig. 6 visualizes these input and output variables of a conceptual processing node.

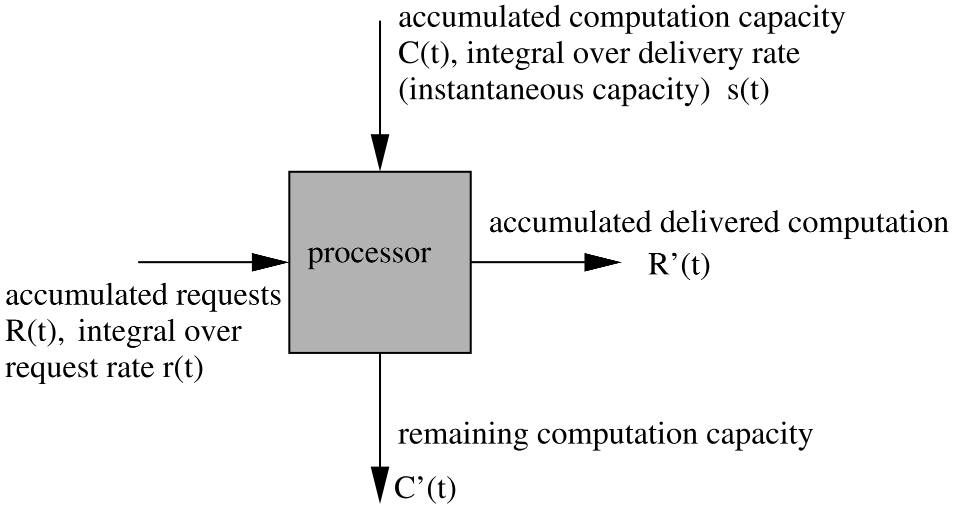

Fig. 6: Accumulated input and output variables of a processing system in the time (t) domain.

Progress on task execution in dependence of the available computation capacity and the actual load are calculated using the following equations:

Cor. 3.2.1: **(Relationship between C, R and R′)** *For any processor with accumulated request function R(t) and accumulated capacity function C(t) it holds that:*

$$R'(t) = \min_{0 \le u \le t} \{R(u) + (C(t) - C(u))\}\,. \tag{3.1}$$

Proof: The actually delivered amount of computation, $R'(t)$, must be limited from above by both the accumulated capacity function C(t) and the accumulated request function R(t). In particular, $R'(t)$ is clearly equal to the maximum amount of computation that could have been delivered by t, C(t), minus the maximum wasted capacity up to t. Capacity is wasted during all intervals when R(t) is smaller than C(t). Thus

$$R'(t) = C(t) - \max_{0 \le u \le t} \{C(u) - R(u)\}\,.$$

$$\iff R'(t) = C(t) + \min_{0 \le u \le t} \{R(u) - C(u)\}$$

$$\iff R'(t) = \min_{0 \le u \le t} \{R(u) + C(t) - C(u)\}\,.$$

■

The unused capacity can then be calculated with the following corollary:

Cor. 3.2.2: **(Relationship between C, R′ and C′)** *For any processor with accumulated request function R(t) and accumulated capacity function C(t) it holds that:*

$$C'(t) = C(t) - R'(t). \tag{3.2}$$

Proof: The proof is clear from the definition of C and Corollary 3.2.1: The accumulated unused capacity is the difference between the accumulated available capacity and the capacity actually used by computations. ■

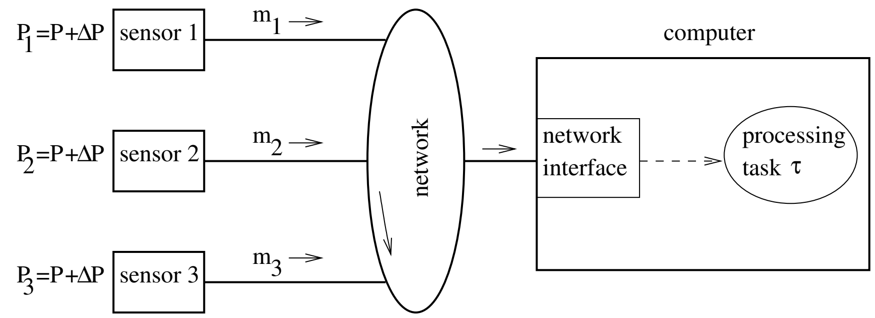

Fig. 7: Example system: Three sensors are connected to a computer via a network.

Ex. 3.2.3: *Consider the system shown in Fig. 7: A number of sensors are connected to a network. Periodically, the sensors send messages to a central processor also on the network. The transmission period P for all sensors is assumed to be identical, but the sensors are not synchronized and the actual time at which they send their message may have some jitter. The message sizes m vary between a known minimum and maximum, and we assume that the computation time required to process a message is proportional to the message length ($e' \propto m$). Thus each message arrival results in the release of a task τ with worst-case computation demand e', $e'_{min} \leq e' \leq e'_{max}$. We assume that the network interface of the computer can read only one message from the network at a time. After the network interface has completely received a message, it requests the execution of the message processing task. The time T' the network interface needs to receive a message of length m ($T'_{min} \leq T' \leq T'_{max}$) thus constitutes the minimum interarrival time for a request of τ with a processing demand e'. Suppose that T' and e' are related by $T' = \frac{1}{u_{var}} * (e' - e'_{min}) + T'_{min}$, with the utilization bound for the variable part of the task specification defined as $u_{var} = \frac{e'_{max} - e'_{min}}{T'_{max} - T'_{min}}$. A long message blocks the network for a longer time than a smaller one before requiring the execution of the control task.*

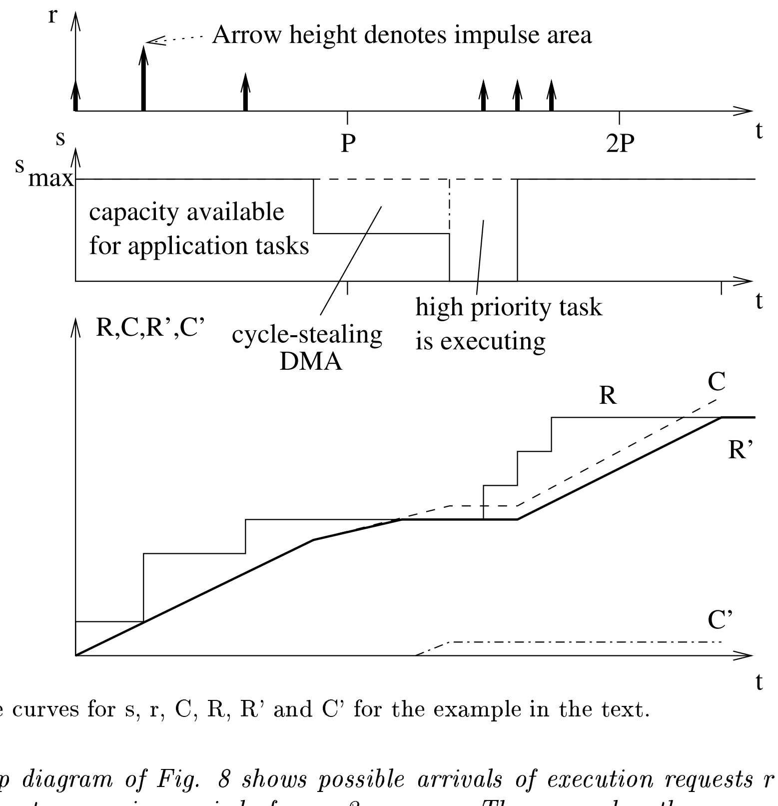

Fig. 8: Possible curves for s, r, C, R, R' and C' for the example in the text.

The top diagram of Fig. 8 shows possible arrivals of execution requests $r(t)$ for τ during two sensing periods for n=3 sensors. The arrow lengths represent the respective amount of computation capacity that is requested. The middle diagram indicates that task τ is not all times executed with the full theoretical speed of the *processor, $s = s_{max}$: At times the processor is slowed down, e.g. due to cycle stealing DMA, and in other intervals a high-priority task is executing, so that no capacity is left for τ. The lower diagram shows the curves for the resulting accumulated variables C, R, R' and C' as defined above.*

The mathematical model of Fig. 5 and 6 which uses time dependent values for R, C, R', and C' is not useful for analysis because it does not necessarily express the worst case constellation between these variables, that is, the situation where the highest possible demand coincides with the lowest available capacity. To describe this situation an abstraction is introduced which characterizes the system in terms of the behavior observable in intervals of length Δ that may start at arbitrary times during system life time (Fig. 9).

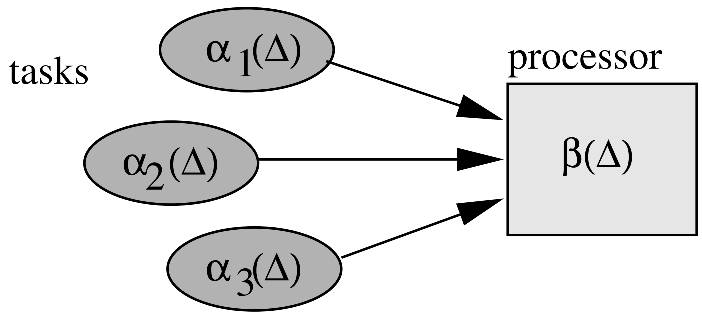

Fig. 9: Task and processor attributes expressed as averages over intervals in order to characterize worst case constellations.

This allows to obtain analytical results that are independent of any particular point in time and thus to derive latency bounds and a schedulability test for the system. Computation and meaning of the functions α and β, in Fig. 9 as well as the schedulability test will be explained in the following sections.

3.3 Modeling bursty tasks

[69] and [112] characterize tasks by the maximum amount of computation requested by a discrete number of arrivals within any interval of a certain size. In order to capture two further categories of task specifications,

- tasks with variable or bursty arrival characteristics, as in the above example, and
- sets of alternative tasks or modes with different arrival profiles (see Sec. 3.3.1.5),

this approach can be formalized and extended for continuous utilization bounds by adapting the concept of arrival curves from the networking domain [45, 46, 96] for task releases, resulting in the following definition:

Def. 3.3.1: **(Request curve)** *The request curve α of a task with an accumulated request function R is a wide-sense increasing, non-negative function. R is said to be constrained by α, iff*

$$R(t) - R(s) \leq \alpha(t - s) \quad \forall s \leq t. \tag{3.3}$$

The intuitive meaning of a request curve α for a task τ is, that within any time interval of length Δ only a total amount of computation $\alpha(\Delta)$ may be requested by the releases of τ, either through frequent requests with small computational demands, rare requests with high computational demands, or any suitable combination. A task requires the maximum possible computation capacity, if its accumulated request function R equals its request curve α. Such a task is called greedy.

The request curve $\alpha(\Delta)$ guarantees an upper bound on the amount of computation requested per interval of size Δ by the task which it describes. For the purpose of analysis one is, of course, interested in the smallest bound that still accurately describes the arrival behavior of the task. A request curve α used for analysis should thus always be subadditive[1] ($\alpha(s+t) \leq \alpha(s) + \alpha(t),\ \forall s,t \geq 0$): A bound imposed by a subadditive request curve may be tight (greedy task), whereas the bound described by a non-subadditive request curve can not be tight without leading to a contradiction and inconsistency in the specification, as a non-subadditive request curve would imply that there are intervals with a higher average request density than any of their sub-intervals. If we assume that the specification is consistent and the request curve is actually a valid upper bound for all interval lengths Δ then a non-subadditive request curve can for the purpose of analysis be replaced by its subadditive closure $\overline{\alpha}$.

Given an accumulated request function in the t-domain, the request curve in the Δ-domain can be constructed as follows:

Cor. 3.3.2: **(Construction of the minimum request curve)** *For a task characterized by an accumulated request function R, the minimum constraining request curve α_{min} can be calculated as*

$$\alpha_{min}(\Delta) = (R\overline{\ominus}R)(\Delta) = \max_{u \geq 0}\{R(\Delta + u) - R(u)\} \tag{3.4}$$

(proposition 4 in [96]).

Ex. 3.3.3: *Consider again example 3.2.3. For simplicity assume that $T'_{min} = 0$ and $e'_{min} = 0$. Fig. 10 shows again the accumulated request function $R(t)$ from Fig. 8 as well as a request curve α bounding it, and a worst-case bound R_{wc} on all curves $R(t)$. The latter reflects the conservative model used by conventional schedulability tests for periodic tasks that do not allow the specification in terms of an utilization bound but use the minimum interarrival time $T_{min} = 0$ and the maximum requested capacity per period ($3 \times e'_{max}$) instead. The request curve α is smaller than R_{wc} and thus clearly advantageous for schedulability considerations. By constructing various different curves $R(t)$ permitted by the assumptions made in example 3.2.3 and checking that they touch but never intersect α one can convince oneself that α is actually the smallest tight upper bound for the requested capacity.*

3.3.1 Constructing the request curve

The construction of request curves using Corollary 3.3.2 is computationally expensive. The following subsections give methods that can be used to determine α_{min} for several common special cases.

3.3.1.1 Periodic task

For a task that is specified by a constant period P and a constant worst-case execution time e, the request curve α is identical to the worst case accumulated request function R, that is, a staircase function with step height e, first step at time $t = 0^+$ and all consecutive steps as early as possible, at $t = (k * P)^+$, k=1,2,3,4. In this case the request curve α has a shape identical to the conventional periodic request bound, e.g. R_{wc} in Fig. 10.

[1] Refer to Appendix A for mathematical definitions used in this chapter.

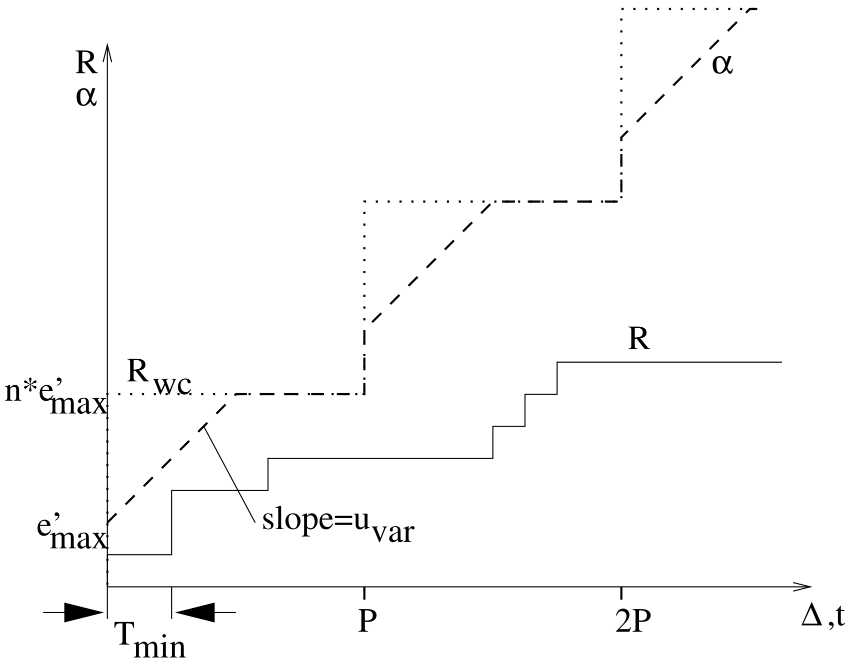

Fig. 10: Construction of the request curve.

3.3.1.2 Multiframe task

A multiframe task [112] consists of N consecutive requests with, possibly different, worst-case execution times $e_0 \ldots e_N$ and minimum separation time P between requests. The whole request pattern repeats cyclically with a hyperperiod of $N \cdot P$.

The request curve can be constructed with the following algorithms, which are equivalent to Corollary 3.3.2 but make use of the fact that there is only a finite number of time intervals Δ which need to be checked. First a table with the N relevant values for one hyperperiod needs to be calculated:

Alg. 1:

```
FUNCTION makeTable(INTEGER N, REAL e[]): REAL[]
  // calculate table for all deltas within the first hyperperiod
  INTEGER delta,i,j,
  REAL sum,max,table[]

  table[0]:=0
  FOR delta=0 TO N-1
    max:=0.0
    // try all starting points for a sequence
    FOR i=0 TO N-1
      // sum all sequences of length delta
      sum:=0.0
      FOR j=0 TO delta
         sum:=sum+e[(i+j) mod N]
      END
      IF sum>max THEN max:=sum
    END
    table[delta+1]:=max
  END
  RETURN table[]
END
```

As the task request pattern is cyclic it is sufficient to look at the N intervals with different sizes Δ ranging from 0 to N-1 (outer loop). For each interval size all N possible starting points (0 to N-1) for a sequence need to be investigated (middle loop). In the inner loop the execution times of the Δ consecutive requests after the starting point of the respective interval are added. For each interval length the largest sum is stored as table entry at index $(\Delta + 1)$, ranging from 1 to N.

Due to the three nested loops this algorithm is of complexity $O(N^3)$. Exploiting the fact that the periodicity of the requests with hyperperiod $N \cdot P$ results in an equal periodicity of the request curve, α is then calculated as follows:

Alg. 2:

```
FUNCTION alpha(REAL delta,REAL[] table, INTEGER P, INTEGER N) : REAL
  RETURN table[CEILING((delta mod NP)/P)]
END
```

First the interval size *delta* of the argument is mapped to a valid table index in the range of 1 to N and then the table entry at the resulting index is returned.

For a multiframe task with the AM-property [112], this reduces to the problem of finding the peak frame of the task. The request curve α is then identical to the accumulated request function R that results from releasing the peak frame at $t = 0^+$ and all other frames in the sequence determined by the task specification, with distance P.

3.3.1.3 General real-time task

The General Real Time Task [112] is defined by a minimum period P of consecutive releases and a function $\phi(i)$ which returns the maximum sum of the execution times of i consecutive releases of the task. This function is thus equivalent to the table that had to be calculated from the specification for the multiframe task in the previous section. The request curve of a general real time task is thus calculated as $\alpha(\Delta) = \phi(\lceil \frac{\Delta \mod NP}{P} \rceil)$ (cf. Alg. 2).

3.3.1.4 Tasks specified by utilization bounds

If a task is specified by a worst-case execution time e'_{max} and a maximum utilization bound u then the request curve immediately follows as $\alpha(\Delta) = e'_{max} + u \cdot \Delta$.

3.3.1.5 Alternative tasks

When designing a system there may be a situation in which several tasks with different arrival characteristics alternatively take the same place in the schedulability analysis, e.g. if

- a functionality of the application operates in several modes, each with its own task structure, or
- for a certain functionality there exist several implementation alternatives and it has at the time of the schedulability analysis not yet been decided which one will actually be used.

In these cases the request curve that covers the whole set of alternatives is computed by taking the maximum of the request curves of all member tasks for each interval Δ.

3.4 Modeling variable processor speed

Usually processors serving task requests are considered to be able to deliver a constant amount of computation per unit time. This is, e.g., the implicit basic assumption for the well-known results of [105] as well as the feasibility tests of [74] and [99].

The more general concept of a delivery curve, which is inspired by the use of service curves in the networking domain in [96], characterizes a processor via the minimum amount of computation that is guaranteed to be delivered in an interval of length Δ starting at any arbitrary point in time t. This leads to the following definition:

Def. 3.4.1: **(Delivery curve)** *The delivery curve β of a processor with accumulated capacity function C is a wide-sense increasing, non-negative function. The processor is said to offer a delivery curve of β, iff*

$$C(t) - C(s) \geq \beta(t-s) \, \forall \, t, s : \, s \leq t. \tag{3.5}$$

The value of the delivery curve $\beta(\Delta)$ for any interval of length Δ can be obtained a posteriori from a record of the actually available instantaneous delivery rate s(t) (see Fig. 11), by looking at a moving window of size Δ. $\beta(\Delta)$ is the minimum over all these windows of the integral over the available capacity:

Cor. 3.4.2: **(Construction of the maximum delivery curve)** *For a processor characterized by an accumulated capacity function C, the maximum constraining delivery curve β_{max} is*

$$\beta_{max}(\Delta) = (C \underline{\ominus} C)(\Delta) = \min_{u \geq 0}\{C(\Delta + u) - C(u)\}. \tag{3.6}$$

Proof: It has to be shown that any service curve obtained from equation 3.6 also fulfills the definition of β, equation 3.5:

$$\beta_{max}(\Delta) = (C \underline{\ominus} C)(\Delta) = \min_{u \geq 0}\{C(\Delta + u) - C(u)\}$$

$$\Rightarrow \; \forall \, \Delta \, \exists \, u_0 \, : \, \beta_{max}(\Delta) = C(\Delta + u_0) - C(u_0)$$

$$\Rightarrow \; \forall \, \Delta \, : \, \beta_{max}(\Delta) \leq C(\Delta + u) - C(u) \; \forall u$$

For $t = \Delta + u$, $s = u$ follows

$$C(t) - C(s) \geq \beta_{max}(t - s), \; \forall t, s$$

It is also necessary to prove that there is no bigger β that would also fulfill equation 3.5. Proof is by contradiction:

Assume there is a β such that there exists a $\delta \geq 0$ with $\beta(\delta) > \beta_{max}(\delta)$. This β must also fulfill equation 3.5:

$$\beta(\delta) \leq C(\delta + s) - C(s) \, \forall s \;\; with \;\; \delta = t - s$$

The all-quantified expression can be converted to a min-expression:

$$\beta(\delta) \leq \min_{s \geq 0} \{C(\delta + s) - C(s)\}$$

The right hand expression is identical to $\beta_{max}(\delta)$ as calculated using equation 3.6. This results in

$$\beta(\delta) \leq \beta_{max}(\delta)$$

which is in contradiction to the initial assumption. ■

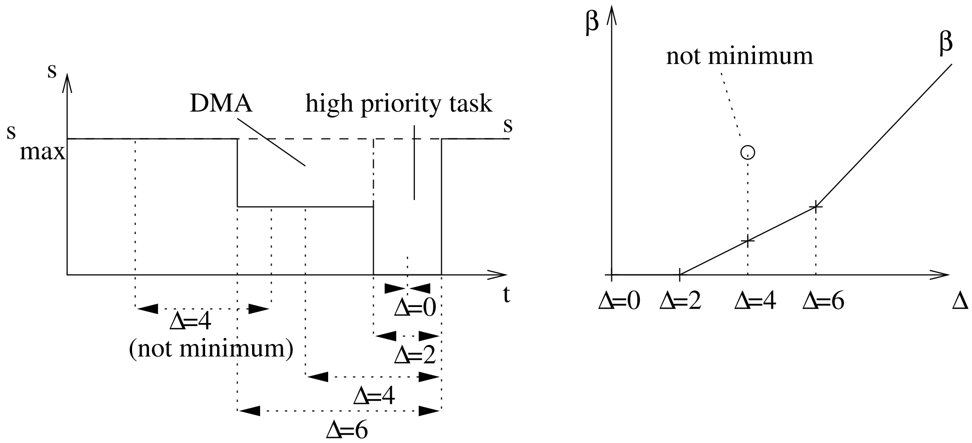

Fig. 11: Construction of the delivery curve.

Ex. 3.4.3: *A delivery curve can be derived either theoretically from the specification of the processing system or from measurements. Fig. 11 illustrates Cor. 3.4.2 by demonstrating the a posteriori construction of the delivery curve from measurements, in this case from the information given about the delivery rate s(t) in Fig. 8. The diagram on the left hand side plots the processor speed/capacity available for τ, the task under consideration, over time. One can see that there are two intervals of reduced capacity in the part of the timeline that is shown. When constructing a schedule for hard real-time systems, it is necessary to take the worst case of processor availability into account. This worst-case scenario for the available accumulated computation capacity after a release of τ should be captured in the delivery curve β. The delivery curve $\beta(\Delta)$ is constructed by integrating s(t) in each interval of length Δ for all Δ. In the diagram this is visualized for some discrete values of Δ ($\Delta = 0, 2, 4, 6$) at selected interval starting points. As can be seen for example for $\Delta = 4$, depending on the interval chosen there will be different results for the integral, that is, the available computational capacity. The delivery curve has to represent the worst-case, so the respective minimum over the values obtained for a certain Δ for all starting points will be selected to construct the curve.*

For a processor characterized by a delivery curve β, a lower bound on the actually delivered computation can be given:

Cor. 3.4.4: (Lower bound on R′)

Given is the delivery curve β of a processor. If $\beta(0) = 0$ then for all t there exists a t_0, $0 \leq t_0 \leq t$ such that

$$R'(t) - R(t_0) \geq \beta(t - t_0), \tag{3.7}$$

which can also be expressed using min-plus-convolution

$$R'(t) \geq (R \underline{\oplus} \beta)(t) = \min_{0 \leq u \leq t} \{R(u) + \beta(t-u)\} \tag{3.8}$$

Proof: Using Corollary 3.2.1 and Definition 3.4.1:

$$R'(t) = \min_{0 \leq u \leq t} \{R(u) + (C(t) - C(u))\}.$$

$$R'(t) \geq \min_{0 \leq u \leq t} \{R(u) + \beta(t-u))\} = (R \underline{\oplus} \beta)(t)$$

■

This means, that for a processor characterized by a delivery curve β the allocation of a certain amount of computation capacity can be guaranteed for a task with requests according to a certain accumulated request function R. As will be shown in the next section this allows to calculate an upper bound on latency for that task.

Ex. 3.4.5: *Fig. 12 illustrates this lower bound on R': The accumulated request function R shown here is caused by three releases of task τ, and the processor is characterized by the delivery curve β constructed above. The accumulated computation function R' is an example of a possible task execution profile. The worst-case situation for any task arrival would be that during the execution of this job the processor passes through the two periods of reduced processing capacity described in the previous example. Of course, this worst case will not always occur. In Fig. 12 it occurs for the second arrival only. The lower bound on delivered computation, $R \underline{\oplus} \beta$ (the dotted line in Fig. 12), is the minimum envelope produced by sliding the $\Delta = 0$ point of the β curve along R. This represents for each release of task τ the respective worst case situation.*

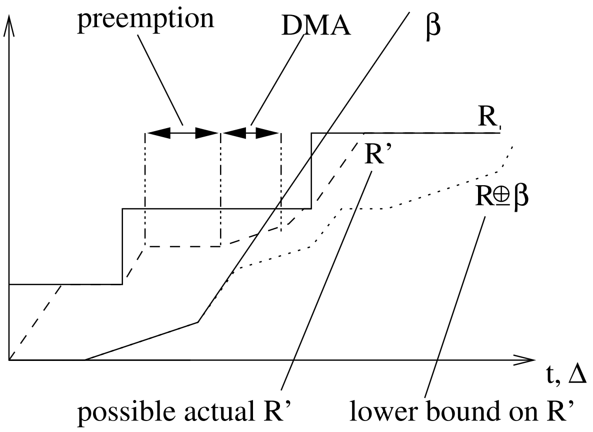

Fig. 12: Illustration of the lower bound for R'.

It is shown in [120] that this bound is tight in the sense that for each super-additive delivery curve β ($\beta(s+t) \geq \beta(s) + \beta(t)$, $\forall s, t \geq 0$) an accumulated capacity function C(t) can be constructed such that $R' = R \underline{\oplus} \beta$ for a certain t and accumulated request function R. Related properties of β are also proven in [120].

3.5 Scheduling analysis

In the previous sections the Δ-domain has been introduced and based on it time independent worst-case characterizations for both tasks and processors have been found. Fig. 13 displays the Δ-domain model of a processing system with input bounds α and β given by a specification and output bounds derived from them. In the following sections it will be shown how to calculate a bound β' on the remaining processing capacity that can be guaranteed after a certain task has been scheduled and how to calculate an upper bound on the latency L encountered by any scheduled task. With these two bounds finally a schedulability test for multi-level fixed priority preemptive systems will be constructed.

3.5.1 Combining tasks

Example 3.4.3 (Fig. 11) already implies, that the delivery curve β, and thus the processing capacity available to a task τ, depends both on the raw processing

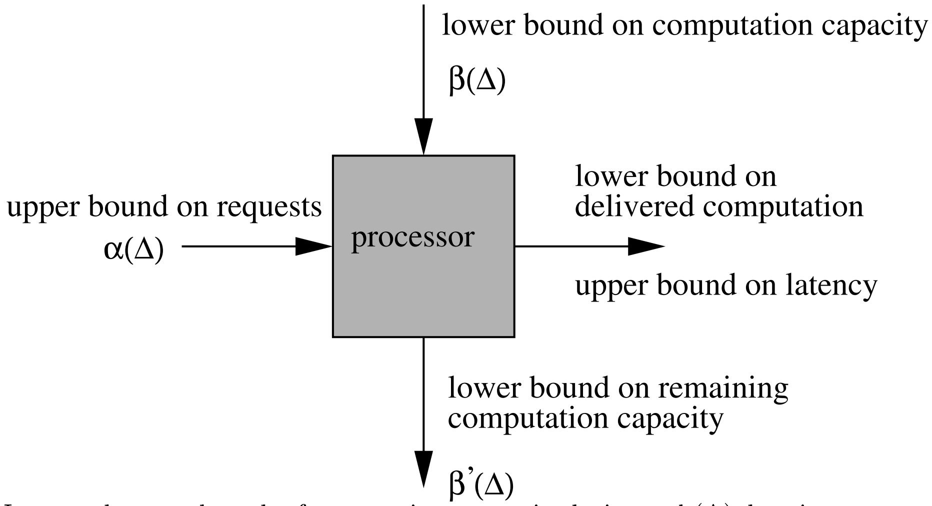

Fig. 13: Input and output bounds of a processing system in the interval (Δ) domain.

capacity and the capacity that is used by tasks of higher priority than τ. The following theorem formally states how the delivery curve β' guaranteed to a task τ is calculated from the original delivery curve β and the capacity requested by higher priority tasks, α_{prior}, by concatenating multiple nodes of Fig. 13 as shown in Fig. 14:

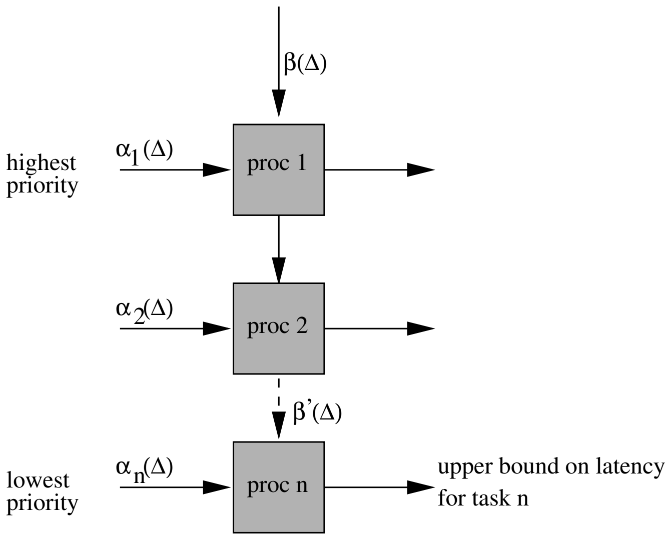

Fig. 14: Concatenating multiple Δ-domain processing nodes to calculate the lower bound on computation capacity guaranteed to the lowest priority task.

Theo. 3.5.1: **(Remaining delivery curve β')** *Given is a task set Γ of n tasks $\tau_1 \ldots \tau_n$, each with an accumulated request function R_i constrained by a subadditive request curve α_i. Assume that $\tau_1 \ldots \tau_{n-1}$ have higher priority than τ_n. The service offered to Γ be characterized by a superadditive delivery curve β. Then the service offered to τ_n is characterized by*

$$\beta'(\Delta) = \max_{0 \le u \le \Delta} \{\beta(u) - \alpha_{prior}(u)\} \qquad (3.9)$$

$$\textit{with } \alpha_{prior} = \sum_{i=1}^{n-1} \alpha_i$$

Proof: From Corollary 3.2.2 the actual computation capacity left for task T_n is known to be $C'(t) = C(t) - R'(t)$ with $R'(t) = \sum_{i=1}^{n-1} R'_i(t)$. It is now necessary to prove that β' fulfills Definition 3.4.1 with respect to C'.

$$C'(t) - C'(s) = C(t) - C(s) -$$

$$-\left(\min_{0\le u\le t}\{R(u) + (C(t) - C(u))\} - \min_{0\le u\le s}\{R(u) + (C(s) - C(u))\}\right),\ t \ge s$$

C(t) and C(s) are independent of the minimization, thus

$$C'(t) - C'(s) = C(t) - C(s) - C(t) + C(s) -$$

$$-\left(\min_{0\le u\le t}\{R(u) - C(u)\} - \min_{0\le u\le s}\{R(u) - C(u)\}\right),\ t \ge s$$

$$= \left(\min_{0\le u\le s}\{R(u) - C(u)\} - \min_{0\le u\le t}\{R(u) - C(u)\}\right),\ t \ge s$$

$$\ge \left(\min_{0\le u\le s}\{R(u) - C(u)\} - (R(v) - C(v))\right),\ t \ge s, \forall v \le t$$

$$= \min_{0\le u\le s}\{R(u) - C(u) - (R(v) - C(v))\},\ t \ge s, \forall v \le t$$

$$= \min_{0\le u\le s}\{(C(v) - C(u)) - (R(v) - R(u))\},\ t \ge s, \forall v \le t$$

$$\ge \min_{0\le u\le s}\{\beta(v-u) - \alpha(v-u)\},\ t \ge s, \forall v \le t$$

as this is true for all v, it is also true for the v that causes a maximum:

$$\ge \min_{u\le s}\left\{\max_{v\le t}\{\beta(v-u) - \alpha(v-u)\}\right\},\ t \ge s,$$

set d=v-u

$$= \min_{u\le s}\left\{\max_{d\le t-u}\{\beta(d) - \alpha(d)\}\right\},\ t \ge s,$$

The minimum over all maxima in a shrinking interval is equal to the maximum in the final (smallest) interval which is obtained by setting u=s.

$$= \max_{d\le t-s}\{\beta(d) - \alpha(d)\} = \beta'(t-s),\ t \ge s.$$

■

3.5.2 Latency

In order to derive schedulability tests, it is necessary to explain how a worst-case bound for the latency of a task can be calculated within the mathematical framework presented above. For this purpose latency is defined as follows:

Def. 3.5.2: **(Latency)** *The latency $L(t)$ is the smallest time delay such that all tasks released at time t have been completed by $t + L(t)$:*

$$L(t) = \min\{T : T \ge 0 \ \wedge\ R(t) \le R'(t+T)\} \tag{3.10}$$

The worst-case latency bound for a task can then be shown to be equal to the maximum horizontal distance between α and β (see Fig. 15 and 17):

Theo. 3.5.3: **(Latency bound)** *For a task constrained by a request curve α, with consecutive releases being served using a FCFS policy, and a processor which guarantees a delivery curve β, the latency $L(t)$ for all t satisfies*

$$L(t) \leq \max_{s\geq 0}(\min\{T : T \geq 0 \wedge \alpha(s) \leq \beta(s+T)\}). \tag{3.11}$$

Proof: Definition 3.5.2

$$L(t) = \min\{T : T \geq 0 \ \wedge \ R(t) \leq R'(t+T)\}$$

can also be expressed as

$$L(t) = \max\{T : T \geq 0 \ \wedge \ R(t) > R'(t+T)\}$$

$$\leq \max\{T : T \geq 0 \ \wedge \ R(t) > (R \underline{\oplus} \beta)(t+T)\}$$

$$= \max\left\{T : T \geq 0 \ \wedge \ R(t) > \min_{0\leq u\leq t+T}\{R(u) + \beta(t+T-u)\}\right\}$$

be u_s the u for which the minimum occurs:

$$= \max\{T : T \geq 0 \ \wedge \ R(t) > R(u_s) + \beta(t+T-u_s)\}.$$

For the case of $u_s \leq t$ use $R(t) - R(s) \leq \alpha(t-s)$

$$L(t) \leq \max\{T : T \geq 0 \ \wedge \ \alpha(t-u_s) + R(u_s) > R(u_s) + \beta(t+T-u_s)\}$$

$$= \max\{T : T \geq 0 \ \wedge \ \alpha(t-u_s) > \beta(t+T-u_s)\}$$

set $v = t - u_s$

$$= \max\{T : T \geq 0 \ \wedge \ \alpha(v) > \beta(v+T)\}$$

As the latency bound is to be valid for all t, it has also to be valid for all $v = t-u_s$, and thus in particular for the v resulting in a maximum T.

$$L(t) \leq \max_{v\geq 0}\{\max\{T : T \geq 0 \ \wedge \ \alpha(v) > \beta(v+T)\}\}$$

$$= \max_{v\geq 0}\{\min\{T : T \geq 0 \ \wedge \ \alpha(v) \leq \beta(v+T)\}\} = h(\alpha, \beta).$$

The case of $u_s > t$ requires to find the maximum T such that $R(t) > \min_{t\leq u\leq t+T}\{R(u) + \beta(t+T-u)\}$. At the same time, any minimum M(T) of $R(u) + \beta(t+T-u)$ for the interval $t \leq u \leq T+t$ must be greater or equal to the sum of the minima of the two constituent terms which are, due to the monotonicity of R and β, known to be $R(t)$ and $\beta(t+T-(t+T)) = \beta(0) = 0$. Thus

$$M(t) \geq R(t) + 0$$

which is in contradiction to the requirement of finding a T such that

$$M(t) < R(t)$$

This means that the case of $u_s > t$ can never occur.

■

This bound on L is tight whenever α represents a greedy task ($R(t) = \alpha(t)$) and the worst-case situation with respect to processor availability described by β occurs ($C(t) = \beta(t)$). This includes the conventional case of periodic task arrival and constant processor service and in this case leads to the classical result that the maximum task latency occurs at the critical instant (t=0) [105] (see Fig. 15).

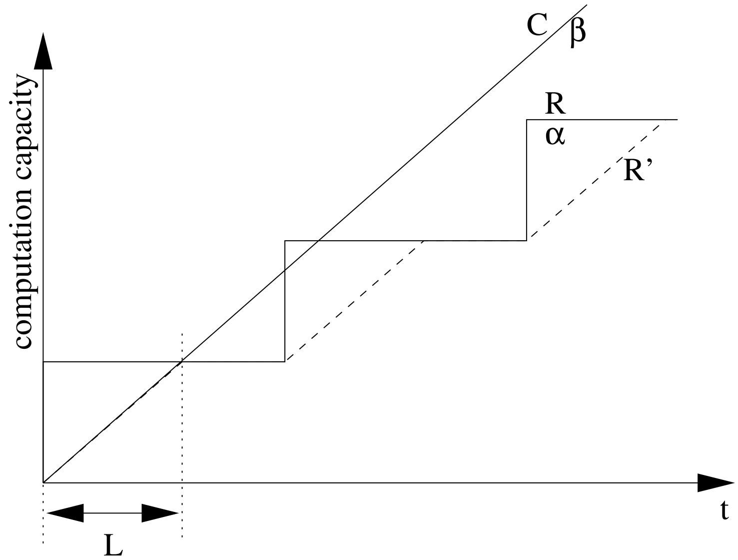

Fig. 15: Example of a situation where the latency bound of Theorem 3.5.3 is tight.

3.5.3 Schedulability test

The results of the previous sections are now combined to derive a schedulability test for a task set Γ (see Fig. 16):

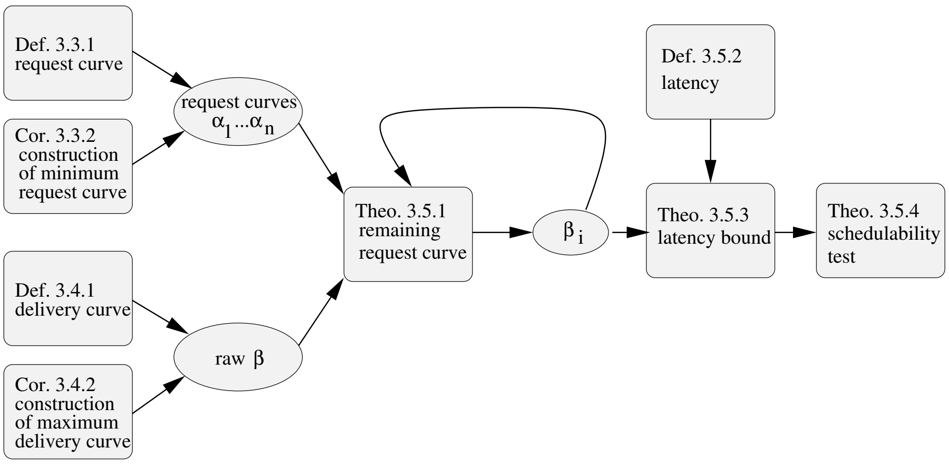

Fig. 16: Combination of the results of Chapter 3 to obtain a schedulability test for multi-level fixed-priority preemptive scheduling.

Definition 3.3.1 and Corollary 3.3.2 are used to construct the request curve of each task in the task set. Definition 3.4.1 and Corollary 3.4.2 define the delivery curve that characterizes the available processing capacity. If the tasks in Γ have different fixed priorities and preemptive scheduling is used, Theorem 3.5.1 can be used to calculate the delivery curve β_i which is guaranteed to a certain task $\tau_i \in \Gamma$. For the definition of task latency given in Definition 3.5.2, a worst-case bound on latency is given in Theorem 3.5.3. From the worst-case bound on latency, the schedulability test follows:

Theo. 3.5.4: (Schedulability Test) *Assume a processor with delivery curve $\beta(\Delta)$. Then a set of n tasks $\tau_1 ... \tau_n$ of decreasing priority which are constrained by request curves $\alpha_i(\Delta)$ and which have relative deadlines D_i is schedulable iff*

$$\max_{s \geq 0} \{\min\{T : T \geq 0 \wedge \alpha_i(s) \leq \beta_i(s+T)\}\} \leq D_i \qquad \forall t \geq 0, i : 1 \leq i \leq n \tag{3.12}$$

where β_i is the service offered to task τ_i calculated using Theorem 3.5.1.

The proof is obvious, as being schedulable means that even in the worst case constellation (critical instant [105]) the latency for each task is shorter than its relative deadline. See Fig. 17 for an example. Clearly the calculations necessary to use this theorem are computationally expensive for arbitrary functions α and β. Nevertheless, the theorem provides a general theory to describe schedulability in the Δ-domain, and for practical cases in which α and β can expressed, either approximated or exactly, as piecewise linear functions, only a finite number of computations are necessary.

In [120] it is shown that for a processor with constant delivery rate s(t)=1 ($\beta(\Delta) = \Delta$) and periodic tasks with computation demand e_i ($\alpha_i = e_i \cdot \left\lceil \frac{\Delta}{P_i} \right\rceil$, $e_i \geq 0$) and deadline D_i equal to the respective task period P_i, Theorem 3.5.4 is equivalent to the test of [74] which is in general equivalent to the test given by [99]. Through suitable modifications of α and β also blocking and release jitter can be represented, which would then allow to derive the iterative general deadline test given in [161].

The schedulability test in the general form of Theorem 3.5.4 does not make assumptions about the relation between deadlines and task periods. It should however be noted, that for the case of general deadlines it is shown in [97] that neither deadline monotonic priority assignment (DMS) nor rate monotonic priority assignment (RMS) is optimal. See [161] for an algorithm to find an optimal priority assignment.

Ex. 3.5.5: *Consider again example 3.2.3 with the request curve of Fig. 10 and the delivery curve of Fig. 11 and assume that in addition to the message handler τ_1 a periodic task τ_2 ($\alpha_{\tau_2} = R_{\tau_2} = e_{\tau_2} \cdot \left\lceil \frac{\Delta}{P_{\tau_2}} \right\rceil$) of lower priority with $D_{\tau_2} = P_{\tau_2}$ needs to be scheduled. Fig. 17 shows the previously constructed request curve α for the message handler task, the delivery curve β_1 of the processor, and the resulting β_2 left for τ_2, computed according to Theorem 3.5.1. As can be seen, the maximum latency L_{τ_2} is smaller than the deadline D_{τ_2} and the task system is thus schedulable. The figure illustrates that the delivery curve can be interpreted as a lower bound on deadlines that can be met.*

In this chapter a general task release model was formalized and some important properties were shown. This task model is a unified model that incorporates previous models and is, in addition, well suited to describe bursty tasks, tasks with variable computation demands and sets of alternative tasks, as well as general deadlines. Also, a new model for the computation delivered by a processor was introduced. This processor model can express dead-time and variable processor speed. It was shown how worst-case latency bounds for a task can be calculated using those models. A general schedulability test was derived from this latency bound.

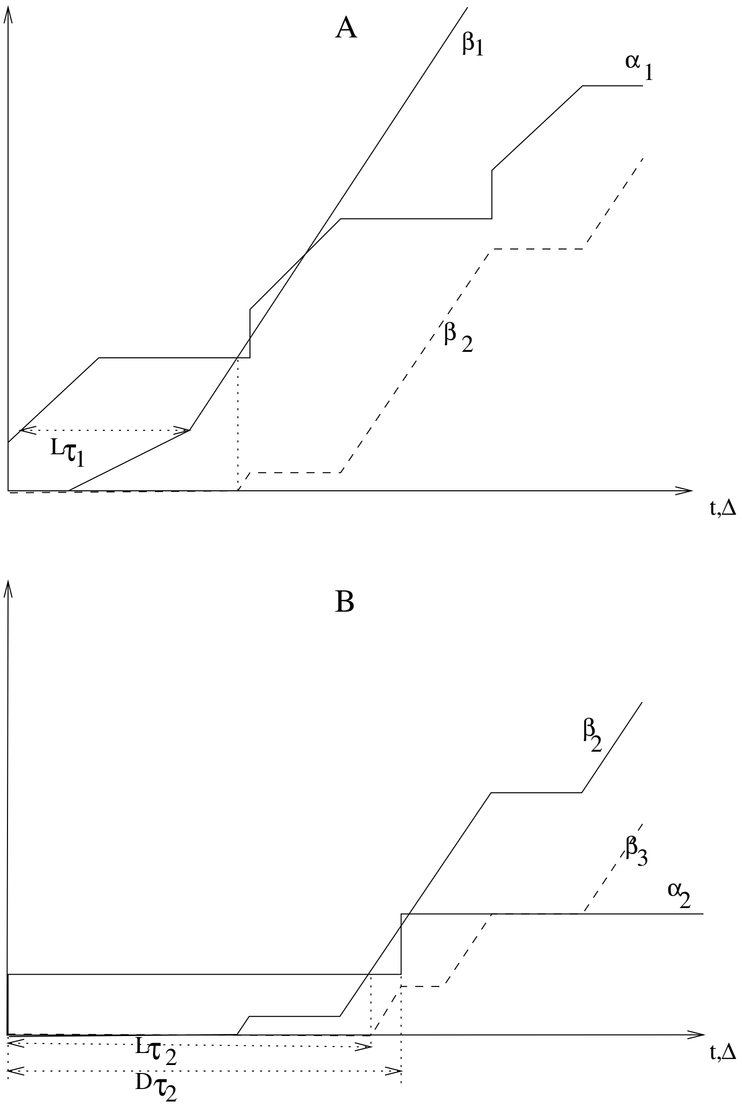

Fig. 17: Derivation of the remaining β for a high priority (A) and low priority (B) task.

4

Petri nets

The expert report [83] on future directions in real-time research which has already been cited in Chapter 1 further states:

> **"There are four different widely used classes of real-time formalisms: logics, automata/state machines, Petri nets, and process algebras. Each of these can probably be integrated with scheduling, testing and run-time checking/monitoring techniques to provide a common framework."** [83]

Among the real-time formalisms named there Petri nets are the closest match to the requirements derived in Section 1.3: They have an intuitive graphical representation [57] appealing to representatives of groups with different (technical) background [51], e.g. systems, hardware, software, mechanical engineers, and physicists. They also possess clearly defined semantics which allows execution [177] and formal analysis [115] and expresses concurrency and synchronization in a natural way [23]. Petri nets are - in contrast to state machines and related models - able to model data and control flow [51], and - in contrast to algebraic and logical specifications - able to represent the implementation structure of the system [26]. With Petri nets, hardware, software, environment, and even user interface of a system can be presented in form of an integrated model [51]. Formalism extensions have been introduced for a notion of time, for data processing, and for hierarchical structuring so that so-called high-level Petri nets are expressive enough to model complex systems. Petri nets are used for systems modeling and analysis in many domains [87][1] including communications protocol verification [171, 42], flexible manufacturing systems [48, 181, 84], fault-tolerant [38], safety critical [102], and real-time systems (see Section 2.4.4).

This chapter introduces and defines some concepts and notations of the Petri net modeling formalism. This is not a comprehensive treatment of Petri nets - see e.g. [115, 126, 135] for that purpose -, but is intended to provide the information necessary to understand the models presented later in this monograph.

[1] A comprehensive bibliography for Petri nets including many application papers can be found at http://www.daimi.au.dk/PetriNets/bibl

4.1 Basic Petri nets

Petri nets are a family of graphical and mathematical formalisms for modeling, simulation, and formal analysis of discrete event systems [181]. Petri nets allow the representation of both control and data flow within one formalism. Other frequently used modeling formalisms like finite state machines (FSM), Activity Charts of the Unified Modeling Language (UML) [21], marked graphs, and Synchronous Data Flow (SDF) can be regarded as specializations of Petri nets.

There exist many different Petri net variants and definitions, which are extensions of so-called *condition/event nets*:

Def. 4.1.1: **(Condition/event net)** *A condition/event net is defined as a 4-tuple* $CEN = (P, T, F, m_0)$, *where*

- P *is a finite set of places,* T *is a finite set of transitions,*
- $P \cup T \neq \emptyset \bigwedge P \cap T = \emptyset$
- $F \subseteq (P \times T) \cup (T \times P)$ *is a set of directed arcs.*
- $m_0 : P \rightarrow \{0, 1\}$ *is the initial marking. It defines the initial number of undistinguishable tokens on each place* $p \in P$.

A Petri net is a bipartite graph, which means that $p \in P$ and $t \in T$ alternate along each path in the graph. In the usual graphical representation of Petri net graphs, places are depicted as circles and transitions as rectangles (see Fig. 18). Places represent system state and transitions represent state changes. The current state is visualized by marking certain places with tokens (black dots). Transition firings change the token distribution and thus the state of the system. They may reflect the occurrence of events or the execution of an operation.

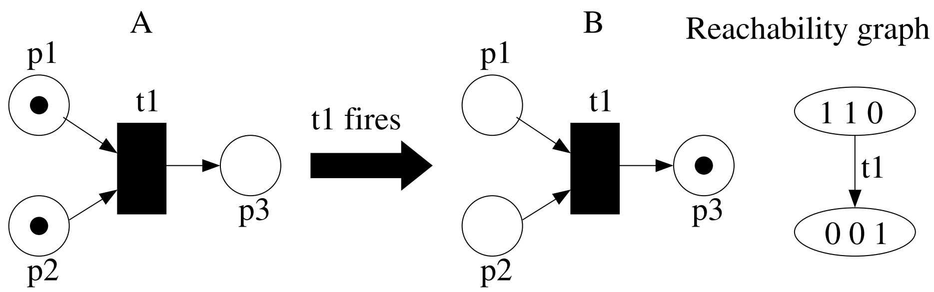

Fig. 18: Simple Petri net system and its reachability graph.

State changes in Petri nets are local. The following two definitions serve to characterize the neighborhood relevant for token flow.

Def. 4.1.2: **(Preset)** *The preset* $^\bullet x$ *of a Petri net element* x, $x \in P \cup T$, *is the set of all* $y \in P \cup T$ *such that* $(y, x) \in F$.

Ex. 4.1.3: *In the Petri net of Fig. 18,* $^\bullet t_1 = \{p_1, p_2\}$ *and* $^\bullet p_3 = \{t_1\}$.

Def. 4.1.4: **(Postset)** *The postset* $x^\bullet$ *of a Petri net element* x, $x \in P \cup T$, *is the set of all* $y \in P \cup T$ *such that* $(x, y) \in F$.

Ex. 4.1.5: *In the Petri net of Fig. 18,* $t_1^\bullet = \{p_3\}$ *and* $p_1^\bullet, p_2^\bullet = \{t_1\}$.

The flow of tokens is then determined by the enabling rule and the firing rule:

Def. 4.1.6: **(Enabling rule)** *A transition* $t_j \in T$ *is enabled if all places in the preset of* t_j *contain a token or the preset is empty, and all places only in the postset are empty or the postset is empty,* $m(p_i) = 1 \; \forall \; p_i \in {}^\bullet t_j \bigvee {}^\bullet t_j = \{\}$ *and* $m(p_k) = 0$ $\forall \; p_k \in t_j^\bullet \backslash {}^\bullet t_j \bigvee t_j^\bullet = \{\}$.

Def. 4.1.7: **(Firing rule)** *An enabled transition t may fire*[2]*. On firing it removes the tokens from all its input places and places one token in each of its output places. The firing of t changes the state of the net from a marking m to a marking m', denoted as $(m \xrightarrow{t} m')$.*

Ex. 4.1.8: *In the Petri net of Fig. 18 transition t_1 is enabled in situation A and its firing removes the tokens from p_1 and p_2 and places one token in p_3. In this situation (B), t_1 is no longer enabled.*

Remark: If several transitions are enabled concurrently the one to be fired next is selected non-deterministically. As these transitions may share places in preset or postset (*conflict*) the non-deterministic decision about the firing order may also have consequences with respect to which transitions fire at all.

Several consecutive firings can be manipulated jointly as a firing sequence:

Def. 4.1.9: **(Firing sequence)** *A firing sequence σ is a sequence of transitions $t_1 ... t_k$. If t_1 is enabled at marking m and each t_i results in a marking that enables t_{i+1}, this can be written as $m \xrightarrow{\sigma} m'$, with $\sigma = t_1 ... t_k$.*

Def. 4.1.10: **(Reachability graph)** *The reachability graph of a Petri net is a directed graph in which the root node, the node without predecessors, is labeled with the initial marking m_0, each edge is labeled with a transition that is enabled at the marking of the source node, and each node is labeled with the marking resulting from the firing of the transition associated with the edge(s) leading to this node.*

4.2 Petri net extensions

To make modeling with Petri nets easier and to keep the resulting net sizes small, various extensions have been suggested. The following paragraph explains the extensions that have been used for the models in this work[3], a detailed discussion can be found in [115, 181, 60].

- A capacity function $C : P \to \mathbf{N} \cup \{\infty\}$ determines the number of tokens that is permitted on a certain place. A transition is only enabled if, in modification to the enabling rule given above, the output places can accept the resulting tokens without exceeding their capacity. The marking functions are extended to $m_0 : P \to \mathbf{N}$ and $m : P \to \mathbf{N} \cup \{\infty\}$ respectively. By default the capacity of a place is infinite.
- In so-called high-level Petri nets [60] (e. g. Coloured Petri Nets (CPN) [87]), tokens are distinguishable and may carry data values of arbitrary types. Tokens without additional data content are then called 'black tokens'. With distinguishable tokens it is relevant which tokens are selected for firing from the places in the preset. For this purpose the concept of a binding is introduced [53]:

Def. 4.2.1: **(Binding)** *Let t be a transition. Let $p_i, i = 1..|^{\bullet}t|$ be the elements (places) in the preset of t. Let all tokens in any p_i be unique and distinguishable. Let M_i be the non-empty set of tokens in the place p_i. A binding $b(t) \in M_1 \times ... \times M_{|^{\bullet}t|}$ of transition t is a combination of tokens from the places in the preset of t, such that each place contributes one token.*

[2] Note that the semantics for untimed Petri nets does not prescribe that an enabled transition actually fires.

[3] As they are not needed for the models and concepts in this monograph, other extensions like edge weights and inhibitor arcs are left out, which simplifies the explanation of the semantics. Detailed definitions taking these aspects into account can for example be found in [53]

Every time a token is added to one of the preset places, p_j, this token is combined with all combinations of tokens from other places and thus $\prod_{k=1..|^{\bullet}t_v|} \begin{cases} |M_k| & \wedge \quad k \neq j \\ 1 & \wedge \quad k = j \end{cases}$ new bindings are created.

For each binding for a transition an individual firing decision is made. On firing, the elements (tokens) of b are removed from their respective places in the preset of t, and all other bindings that contain one or more of these removed tokens are destroyed and will not lead to a further firing. Instead of writing 'transition t fires using binding b' one may also write 'binding b fires'.

The incoming edges of a transition may be labeled with variable names which are used to address the respective elements of a binding in any mathematical expression associated with the transition. Similarly, also the outgoing edges may be labeled with variable names.

The existence of a binding fulfills by definition the enabling rule 4.1.6. However, for high-level Petri nets a further condition is added to the enabling rule: A transition is only enabled by a binding if the *guard* predicate of the transition evaluates to true for this binding. The guard predicate of transition t, $g(t) : b(t) \longmapsto \{true, false\}$, is evaluated for each binding using the values of the elements of the binding as values of the respective variables of the guard expression.

The values of the output tokens of a firing are determined by evaluating *action* expressions with the values of the firing binding bound to the variables of the action expressions. The labels on the outgoing edges associate the tokens produced by the firing with the results of these function evaluations.

High-level Petri nets on the one hand allow to reduce model size as compared to condition/event nets because more state information can be encoded within a single token. On the other hand the possibility to use arbitrary expressions as guards and actions increases the expressiveness of the formalism and, in the general case, prevents formal analysis.

- Transitions are annotated with firing durations. Temporal Petri nets can be divided into timed Petri nets [132], where a certain time passes between removal of tokens from input places (start of firing) and addition of tokens on output places (end of firing) of a temporal transition, and time Petri nets [20], where a transition must be continuously enabled by a binding for a certain time before the binding fires atomically. In conflict situations where a token enables several transitions concurrently a transition may thus be disenabled during this waiting time. Time Petri nets are more powerful in the sense that the behavior of timed Petri nets can be emulated using time Petri net transitions, but not the other way around. In addition to the use of per transition constants for the firing duration parameters, some authors employ a firing duration that depends on the value carried by a high-level Petri net token (e.g. [54]). This concept is also used in this work, with the following semantics:

Def. 4.2.2: **(Firing semantics for token dependent delay)** *Be t_v a transition with token value dependent firing delay. Let b be a newly created binding of t_v. For each newly created binding for which the guard expression evaluates to true, a timer is started and a timer end value, the firing delay, is calculated by evaluating the delay expression $d(t_v)$ which maps the values of the $|b|$ tokens of the enabling binding to time intervals starting at the instant of the evaluation. When the timer of a binding b reaches its end value, the transition fires[4]. The selection among several transitions with timers expired at the same instant is non-deterministic.*

Remarks: All bindings that do not contain any element of b are unaffected by the firing and their timers keep running. If one or more tokens of a binding are removed from their places in the preset of t_v while the timer of the binding is still

[4]This is a major difference to the untimed semantics given above. It is similar to the 'strong' timing of [62].

running, this binding is also destroyed. In a net with firing delays all transitions not explicitly annotated with a firing delay expression are implicitly assumed to have a zero firing delay.

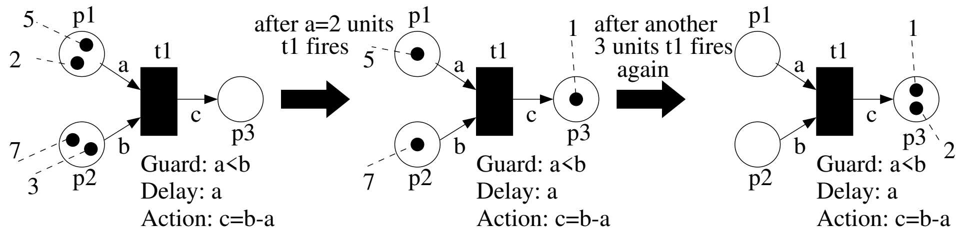

Fig. 19: High-level Petri net system.

Ex. 4.2.3: *Fig. 19 shows a simple high-level Petri net system. The tokens in the input places of the transition carry integer numbers as values. With these four tokens there are four possible bindings, b_1=(2,3), b_2=(2,7), b_3=(5,3) and b_4=(5,7), of which only b_1, b_2, and b_4 enable the transition due to the guard condition specified, which requires the value bound to variable (input edge) a to be smaller than the one bound to b. The timer end values for these bindings specified by the delay expression with the variable a which refers to the token form place p_1 in each binding, are $T_{b_1} = 2$, $T_{b_2} = 2$, and $T_{b_4} = 5$ respectively. After a delay of 2 units both b_1 and b_2 are ready to fire. Here we assume that b_1 is chosen to fire first. The two tokens of the enabling binding will be removed from their places and a new token will be produced in p_3. This new token will carry a value of '1', as this is the result of applying the action expression to the binding. As a further consequence the firing induced by binding b_1 destroys b_2 and thus b_2 never fires. After an additional 3 time units, 5 units from the original creation of all four bindings, the timer of b_4 expires and t_1 fires again.*

In this chapter the static and dynamic behavior of the basic Petri net variant, condition/event nets, has been defined. High-level temporal Petri nets have been introduced as extensions of condition/event nets. Also, the dynamic behavior (firing semantics) of time Petri nets with token dependent firing delay has been defined.

The next chapter will present a novel model structuring principle that makes use of the respective strengths of both the basic Petri net formalism (analyzability) and high-level temporal Petri nets (expressiveness) within one model.

5

Structure of a common model

In this chapter a novel structuring principle is proposed which partitions a Petri net model of a real-time system into a submodel that represents the application functionality and a submodel that governs timing progress.

This separation of concerns not only facilitates evaluation in the respective subdomains, e.g. formal verification of system functionality, but also helps to overcome the difficulties encountered in modeling systems with state-of-the-art preemptive scheduling strategies [32] with Petri nets. These difficulties are responsible for the fact that previous work (see Section 2.4.3) either neglects scheduling at all, uses non-preemptive scheduling where the schedule is prescribed and fixed by the net structure, or preemptive scheduling that is in one way or other incompatible with real-time system characteristics or evaluation methods. This previously unsolved problem of representing general preemptive scheduling strategies has seriously limited the applicability of Petri nets with respect to the real-time domain.

5.1 The general preemption modeling problem for Petri nets

The construction of Petri net models of systems with general preemptive scheduling is difficult [91] for two major reasons:

Execution interruption: Preemptive scheduling makes only sense if a certain non-zero duration for actions/tasks is assumed. This is usually realized by using a Petri net formalism with a notion of delay associated with transition firing. Almost all approaches for real-time system model construction use timed Petri nets [132] for this purpose. The semantics of timed Petri nets is that a firing transition removes the tokens of its enabling binding from its input places, waits for the specified delay and then produces tokens on its output places. This semantics does not allow the waiting to be interrupted, which is a necessary precondition for the realization of preemption.

Previous work that concedes the necessity of preemption implements this either by dividing the task into several non-preemptive sections with preemption points in between [122, 24] or by computing the preemption point based on some random distribution already before the transition is fired and than only delaying the transition firing up to this precalculated time [110, 180]. While the former approach may be an acceptable solution to the problem of interrupting a task if it models the behavior of the actual system closely enough, the latter is problematic with respect to real-time systems for two reasons: First, the non-deterministic lengths of the slices between preemptions lead to a non-deterministic total execution time of the task, which is not compatible with specified worst case execution times in hard real-time systems. Second, this concept does not allow external stimuli (e.g. from submodels representing the environment) to trigger preemption.

Another approach is taken in [104]. Here a temporal transition `tr` fires immediately but produces a token `to` with a creation timestamp in the future ("future token"). If any transition representing a higher priority task fires in between the firing of `tr` and the time specified by the timestamp of `to` then the execution delay of this higher priority task is added to the value of the timestamp of `to`. This approach requires that a specific scheduler is integrated in the execution engine of the simulator and thus mixes the scheduling strategy of the modeled system and Petri net firing semantics. This is particulalary critical if the modeled system uses different scheduling strategies at the same time.

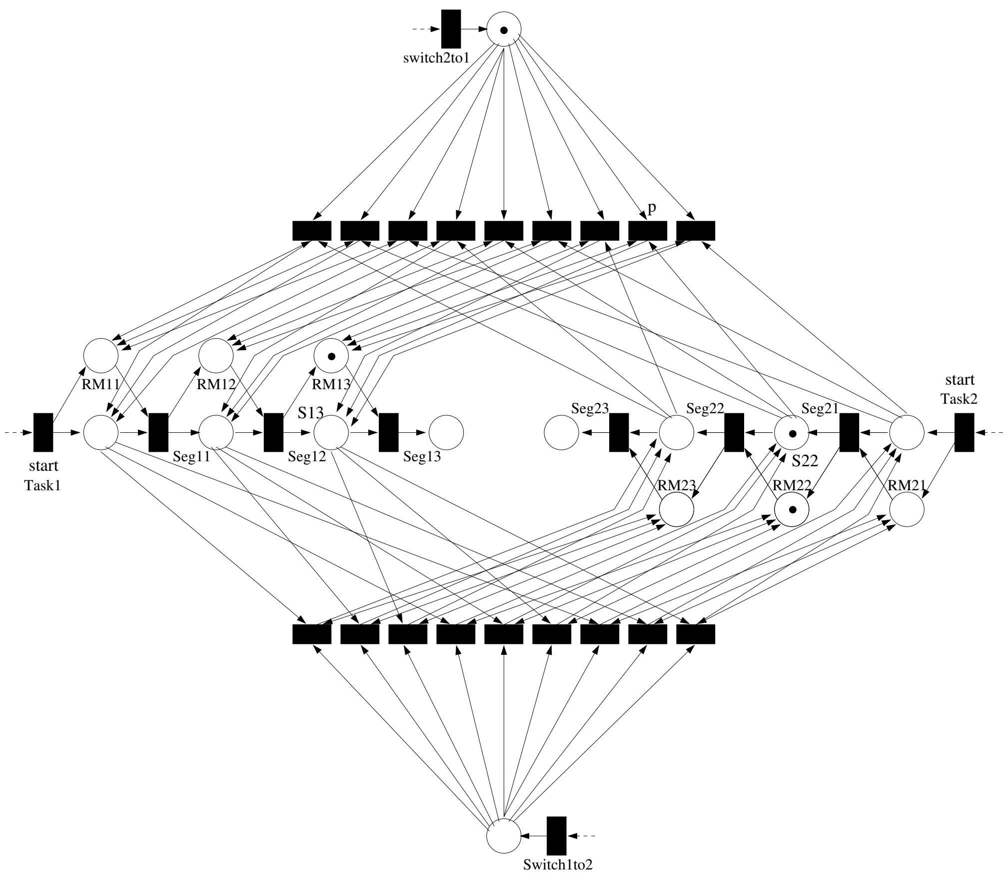

Fig. 20: Structural overhead for task switching between two tasks with three segments each. The `RM` places serve as markers of where to resume execution.

Transfer of control flow: An even bigger problem is constituted by the fact that for a general preemption scheme not only the execution of the current task must be interrupted (see above) but that the control flow must also be transferred from each interruption point of one task to possibly any preemption point of another task, which e.g. with dynamic priority scheduling may be any of the tasks in the system.

In very restricted circumstances (e.g. three tasks, fixed priority structure, abort semantics in [93]), this approach may be directly reflected in the net structure, but in the general case of preemption with resume semantics and dynamic priorities it results in an unacceptable explosion of the structural complexity of the net:

Ex. 5.1.1: *In the example of Fig. 20 the two horizontal structures in the middle are two tasks with three non-preemptable sections and three preemption points each. With the current marking, task 1 was preempted after completing its second segment, task 2 is currently executing, and a preemption command for task 2 is pending. Assume that transition* `Seg21` *just fired, placing the tokens in* `S22` *and* `RM22`*. Now transition p fires, removes the token from* `S22` *and places it on* `S13`*, thus preempting task 2 and resuming task 1.*

While this example shows only one of many possible net structures for preemption and switching between sections of multiple tasks, it does point out the general problem of structural complexity arising from the decentralized approach to the transfer of control flow. In the case of n tasks the transfer structures shown in the figure would need to be provided between any two of the n tasks (n^2 structural complexity).

In [91] a Petri net extension for preemption has been suggested that allows to clear all tokens out of a model subnet. However, this approach supports only preemption with abort semantics. As a precondition it also assumes untimed transitions so that the state of a subnet is at all times represented by the tokens on places in the subnet.

In summary, a modeling approach is considered to solve the *general preemption modeling problem* if it fulfills the following requirements:

1. Tasks can have *non-zero task execution times.*
2. The switching between tasks may happen at *arbitrary preemption points.*
3. Whether and when a task will be preempted can be decided while the task is already executing (*reactive preemption*). It is in general not appropriate to model reactive systems, in which task executions are caused by stimuli from the environment, with clairvoyant schedulers.
4. The task switch can happen between any two tasks. The priority structure must thus not be hard-coded in the net structure (*dynamic priorities*).
5. Execution of a previously preempted tasks may be continued at the point where it was preempted (*resume semantics* versus abort semantics).
6. The approach is *scalable* in the sense that the structural net overhead (places, transitions, edges) required to realize preemption increases at most linearly with the number of tasks.

None of the previous approaches mentioned above fulfills all the stated requirements. The difficulties described above are caused by the fact that in a conventional model the net elements of the application functionality and those for scheduling and preemption are tightly interwoven. The scalability problem indicates that the all-to-all communication should be replaced by a central controller that manages the transfer of control flow between tasks. The issue of arbitrary execution interruption and resumption suggests the use of numerical variables, e.g. in the form of high-level Petri net token values, for the representation of execution progress instead of representing it by discrete net structures. These ideas were used to derive the model structure presented in the next section.

5.2 Delegated execution

A criterion for the quality of a modeling approach is how close a match between the model and the modeled system can be achieved. With respect to real-time system models it should thus be possible to express

- *structural properties* such as task structure and functionality, task precedence chains, interactions between tasks like communication or synchronization, input/output typical for reactive and embedded computer systems, and separation into functional components (application tasks, CPU with operating system and scheduler) by structural elements,
- *data properties* such as scalar timing parameters by numerical or textual attributes, and
- *independently variable and explorable behavior*, such as scheduling strategies, by independent submodels.

Traditionally, two different modeling principles have been used for Petri net system models:

Task-centered modeling as in [63] produces a model that corresponds closely to the task and dependence structure of the application where tokens serve mostly as program pointers. The decision about execution progress and the functionality of the scheduler are tightly integrated into this model of the task structure, which leads to the explosion in structural complexity of a model with preemptive behavior demonstrated above and requires considerable changes in the task structure model to accommodate different scheduling algorithms even though these should be two orthogonal issues.

Resource-centered modeling as in [180] concentrates on representing the system hardware structurally while the tasks to be executed are modeled as data items carried in tokens flowing through the hardware model. On the one hand, this modeling principle accommodates task switching decisions with dynamic successor relations enforced by an instance outside the task. On the other hand, there is no straightforward way to express structured tasks, task functionality, intertask communications, and precedence relations in this approach, which restricts its applicability to systems with independent and unstructured tasks the actual functionality of which is of no importance for the aspects evaluated with the model.

To combine the advantages of both these approaches and to incorporate the ideas formulated at the end of the previous section we propose the *delegated execution* modeling principle: The model consists of a subnet for the tasks which models the usual structural and functional task properties (functional net), and a subnet for the scheduler where all decisions about time progress are localized and the consumed time is allocated to individual tasks (scheduler net). The functional net thus *delegates* the decision about its execution progress to the scheduler net.

A related concept of localization of all time progress in a subnet has also been used in the approach of [49], but there in the context of a multi-level task-centered model, that is, the program segments of the application task levels are untimed but their execution time is determined by timed transitions on the lowest level which represent primitive commands. In that work the problems of (preemptive) scheduling - interruption and transfer of control flow - were neither addressed nor solved.

Ex. 5.2.1: *Fig. 21 shows a realization of the two tasks of Fig. 20 using the delegated execution modeling style and a scheduler subnet. In this case it is not necessary to subdivide the tasks into segments to enable preemption (see Sec. 6.5.3). Tasks and scheduler*

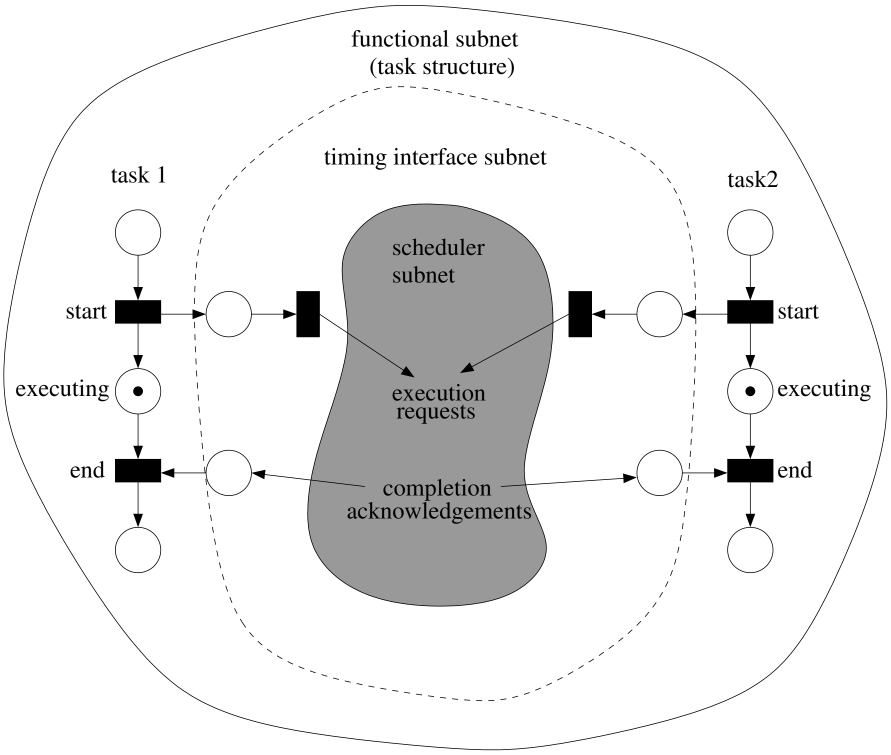

Fig. 21: Two preemptable tasks and a scheduler in a model structured according to delegated execution.

are connected by a timing interface subnet responsible for the storage of timing parameters and communication routing. The two tokens in the `executing` *places indicate that both tasks have requested execution but the jobs have not yet been completed. The scheduler subnet is specific for the scheduling algorithm used. Some exemplary scheduler models are presented in Section 5.4.2.*

5.3 Modeling systems with delegated execution

5.3.1 Submodels and communication

The functional/task structure and the timing interface subnets of Fig. 21 can be regarded as the application specific part of the model which is connected to a standard subnet that implements scheduling and resource access policies, both for single-processor and multiprocessor cases. This view results in a model structure as shown in Fig. 22.

Task and scheduler subnets communicate by message tokens [171]. The messages sent from the application to the scheduler represent execution requests and contain information about the corresponding task, its worst case execution time, deadlines, required access to shared resources etc. The messages sent from the scheduler to the tasks of the application notify these about granted resource access and task completion.

By using a broadcast communication between scheduler and tasks the n^2 structural complexity of the communication structures is reduced to n, as for scheduling decisions the tasks do no longer need to communicate with each other but only with the central scheduler subnet.

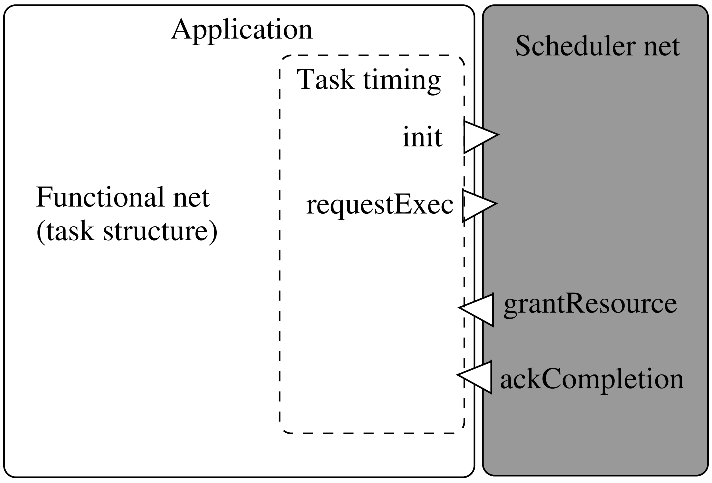

Fig. 22: Model structure with delegated execution.

Besides reducing the communication overhead the second benefit of centralizing the decision about task progress inside a scheduler net is that information about the state of partially executed tasks can be kept as data within the scheduler net and does not have to be transformed to a structural representation as this was necessary in Fig. 20 (places `RMx`). This also allows the implementation of dynamic task priorities and thus resource access protocols [32].

The communication mechanisms are implemented in the task timing interface submodel that connects the task structure and scheduler submodels. In this part of the model the timing parameters for each task are stored and it mediates between the control flow progress in the task structure and the time progress in the scheduler submodel. Using the timing parameter information stored in the interface subnet it converts 'black token' requests for execution from the functional subnet into request tokens with all the necessary data for the scheduler. As the timing information and time dependent behavior are encapsulated within the task timing submodel and the scheduler, it is possible to model the task functionality in a way so that after cutting the connections between the task structure and timing interface subnets, formal Petri net verification methods can be applied to the structural submodel to analyze it with respect to functional correctness (see Chapter 8).

The event-based interface between the timing adapter net of the application submodel and the scheduler net has two token streams in each direction: `init` for task dependent initializations of the scheduler, `requestExec` for execution requests, `grantResource` for notifications that the scheduler has allocated a certain resource, and `ackCompletion` to signal that a certain task segment has been completely executed. The resource grant and completion acknowledgement tokens from the scheduler contain task id values and possibly sequence numbers, so that a structural conflict between the task transitions waiting for these tokens can be resolved deterministically.

5.3.2 Modeling application tasks

Fig. 23 shows part of a Petri net model of the application subnet: a task or task segment consisting of the task structure part and the timing interface part. The task consists of three consecutive stages: request stage, resource access stage, and completion/synchronization stage. The latter two stages may not always be necessary. More information about different variants of the task structure model can be found in Chapter 6.

The task is activated by placing a token into `taskActivated`. This may hap-

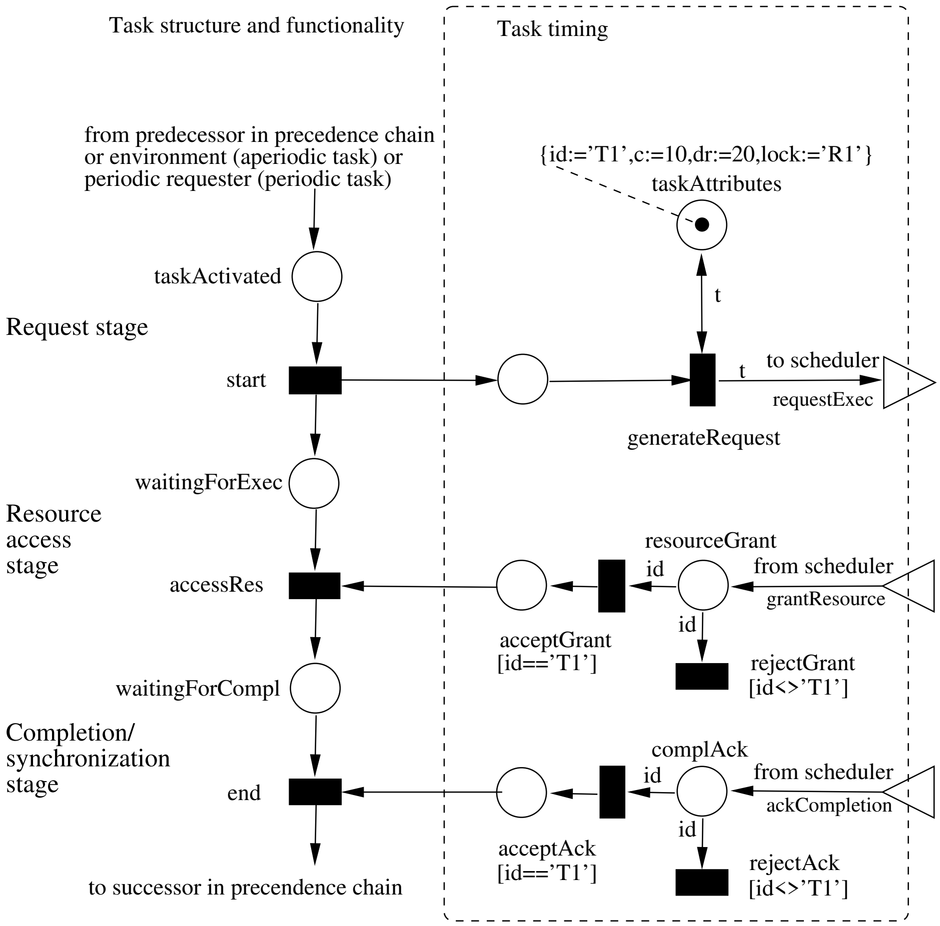

Fig. 23: Task body with resource access.

pen either by a submodel representing the environment (for aperiodic tasks), by a submodel representing a task request mechanism (see Chapter 6), or by the completion signal from a predecessor task in the precedence queue. The token in `taskActivated` causes transition `start` to fire, after which `generateRequest` fires and sends a colored token containing task timing and resource utilization information to the scheduler subnet. The information about the task parameters is taken from the token in `taskAttributes` - in the example a task with name 'T1', worst case execution time of 10 units, relative deadline of 20 units and requesting resource 'R1'. Once the scheduler has reserved the resource for the task segment and the processor starts working on this segment, a colored token with the resource grant notification will be placed into `resourceGrant`. A resource grant is broadcast to all task segments in the model. When the resource for this segment has thus been granted transition `accessRes` fires and the control flow proceeds to `waitingForCompl`. The scheduler signals the completed execution of this segment by placing a colored acknowledgement token into `complAck`. The completion signal causes `end` to fire and to activate any successor tasks.

The `accessRes` transition should be used for any communication with the submodel of a shared resource, because the firing of `accessRes` is guaranteed to happen only after the resource access has been granted. Also, any transition action code should be localized with this transition for the same reason. If the task structure net is extended to represent a more complicated control flow or data processing within the task, the corresponding net should be localized between the `accessRes` and `end` transitions.

5.3.3 Schedulers

While the application submodel (task structure and timing information) is, obviously, largely application dependent, the schedulers are not. Due to the clearly defined interface between the scheduler and the application subnet, scheduler submodels can be created as library components. This allows on the one hand to reuse the scheduler models, on the other hand different schedulers can easily be plugged into the same model to compare the results (see Chapter 9). Petri net models of different scheduler will be discussed in the next section.

The event-based interface between the timing information subnet and the scheduler described above, together with the fact that transition firing and the placing of tokens into a place can also be interpreted as sending and receiving of events, even permits to use scheduler components that are not implemented with Petri nets but with other formalisms or programming languages, as has been done in [117]. In contrast to [13], this approach maintains Petri net execution semantics for the task structure submodel.

5.4 Petri net models of schedulers

In this section three examples for Petri net models of scheduling algorithms are given and it is shown how preemption at arbitrary points is made possible.

By using high-level time Petri nets it is possible to model all kinds of schedulers. The examples presented in the following subsections are non-preemptive first come first served (FCFS), preemptive fixed-priority (FP), and preemptive fixed-priority with priority ceiling protocol (PCP). A catalog with Petri net models of more complex schedulers (integrated interrupt-driven FP, non-integrated interrupt-driven FP, timer driven FP, earliest deadline first (EDF), shortest job first (SJF), resource access protocols (FP with priority inheritance protocol (PIP), FP with stack resource policy (SRP)), and aperiodic servers (polling server, deferrable server, priority exchange server) [32, 90] can be found in [116]. Of course, the approach is not restricted to single-processor models: Like the application submodel the scheduler submodel may be hierarchically structured, e.g. to represent distributed architectures.

5.4.1 Non-preemptive FCFS scheduler

The first scheduler model to be explained is that of a non-preemptive first-come-first-served (FCFS) scheduler (see Fig. 24). Though non-preemptive, it is presented here because it is the most simple scheduler and serves to demonstrate the main elements of a scheduler model and the notation.

A message with the task descriptor of a new request is placed into `newRequests`. The task descriptor token is an object of type `Task` which is a record of task attributes as shown in Fig. 23. The transition `insert` adds the request to the initially empty list of waiting requests on place `requestQueue`. The initial token on `requestQueue` is an empty list of elements of type `Task`. The edge labels denote variables to which are assigned the values of the respective tokens involved in the enabling binding. Transitions are optionally annotated with a guard condition, a firing delay, and an action which transforms token values. The respective instructions are assumed to be executed sequentially in the order in which they are written. The 'dot notation' for object oriented languages like Java and C^{++} is used for the syntax of method calls like adding an element to the end of the list L

(`L.addLast(t)`) and access to an attribute of the token object such as reading the execution time of the task object token in the delay expression of `execute` (`t.c`). This transition is enabled by the task descriptor token of the currently executing task residing on place `execTask`. After the delay period equal to the execution time of the currently executing task has passed (see Def. 4.2.2 for the definition of the firing semantics used here), the task descriptor token is sent back to the task model subnet as a completion notice. The triangular elements denote interface ports to other parts of the system model. They have been explained above. The firing of `execute` places a token in `procIdle` which then causes `select` to fire if the queue is not empty and to place the first waiting task on `execTask`. The number of tokens in place `procIdle` represents the number of available processors in a system with global scheduler, while a distributed scheduling scheme would need to be modeled using multiple scheduler submodels.

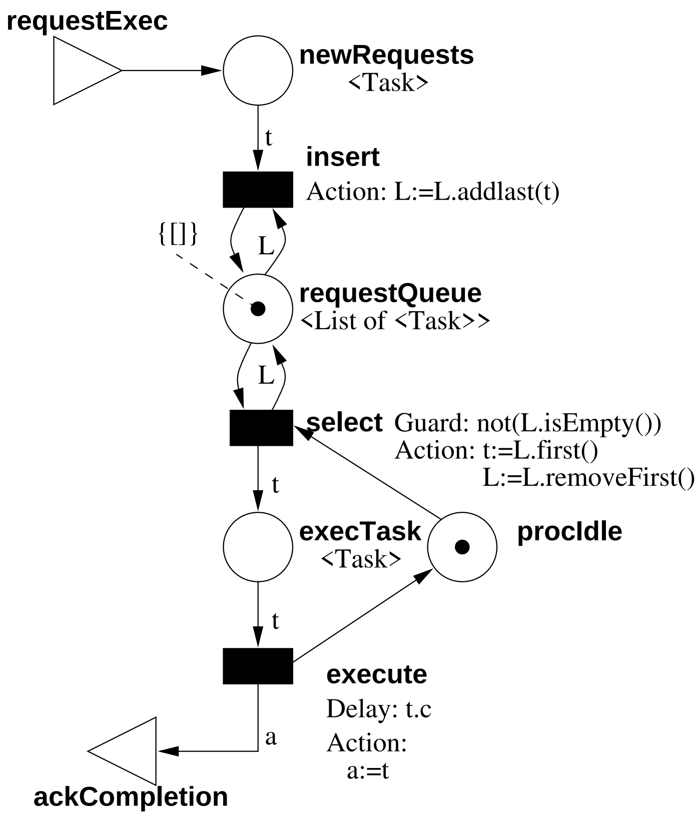

Fig. 24: Non-preemptive FCFS scheduler.

5.4.2 Arbitrary preemption points

The straightforward solution to the problem of permitting arbitrary preemption points is to use the semantics of time Petri nets [111, 20] for transitions that enforce time progress. The important point is that with time Petri net semantics the tokens of the enabling binding remain on the input places during the waiting period until the transition actually fires. It is thus possible to remove (one of) these enabling tokens and thus prevent the transition from firing for the binding concerned.

There are several possibilities to implement resume semantics. In the stochastic Petri net approach [110, 180] the memoryless property of exponentially distributed execution times is exploited to avoid the storage of remaining execution times. For real-time systems with a bounded worst-case execution time this approach is not feasible. Here it will be necessary to store information about the exact enabling and firing or disabling time of a transition so that partial task executions can be accounted for. This requires that the transition action inscription language has access to the current execution time, a functionality offered by modern Petri net execution environments such as [123, 53, 6].

5.4.3 Basic fixed-priority schedule

Fig. 25 shows the model of a preemptive fixed-priority scheduler with neither resource allocation, nor any provisions for non-ideal behavior, such as context switching or scheduler computation overhead.

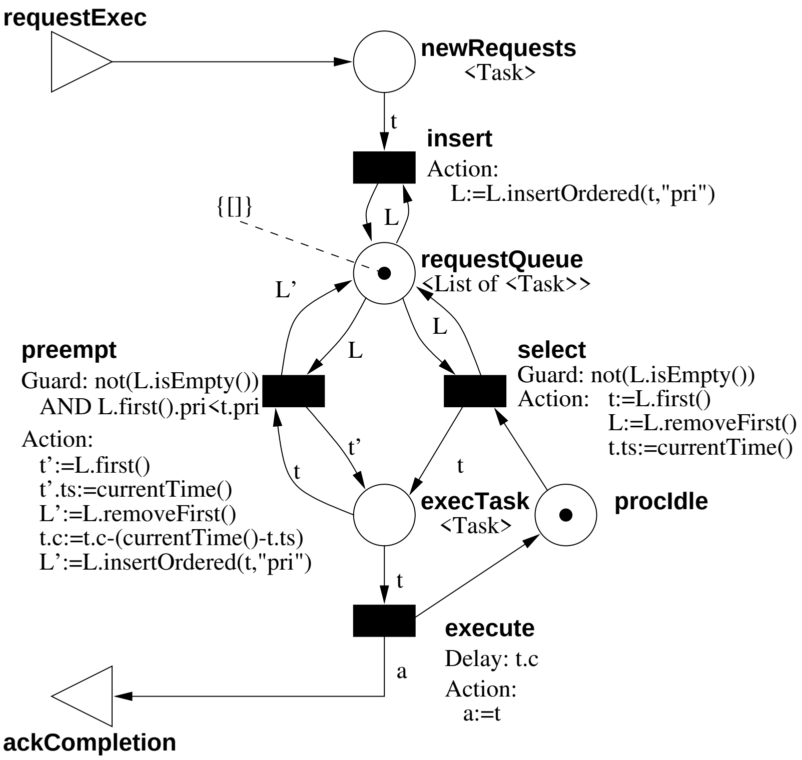

Fig. 25: Basic preemptive fixed-priority scheduler.

To use the preemptive fixed-priority scheduler, the task descriptor token needs to contain more information than the remaining execution time (`c`) only. The attribute `pri` designates the task priority which has been assigned using, for example, the rate-monotonic [105, 142] or deadline-monotonic [10, 101] scheme. In `ts` the start time of the current execution slice of the task is stored. The net of the preemptive fixed-priority scheduler is very similar to the one of the FCFS scheduler, except that a new task is now inserted in the queue according to its priority (`L.insertOrdered(t,"pri")`) and there is an additional transition `preempt` which realizes preemption: If the task at the head of the queue has a higher priority (as common in the scheduling literature this is indicated by a lower numerical value of `pri`) than the one currently residing in `execTask`, they are swapped, that is, the head of the queue is placed on `execTask` and the descriptor token of the previously executing task is inserted into the queue. Every time a token is placed on `execTask` the current time (accessible via function `currentTime()`) is stored into the `ts` attribute of the descriptor token. The action code of the preemption transition then uses this value to calculate how long the task was actually executed in the previous slice and subtracts this from its remaining time. By storing the execution progress in the token and using the token value to specify the transition delay, instead of storing the remaining delay in the transition itself [110], it is possible to express preemption of arbitrary tasks and to do so without any need for replication of the 'CPU transition' `execute`.

The guard condition and action code for the `preempt` transition may easily be extended to give priority to aperiodic tasks, or to discriminate between preemptable, non-preemptable, and non-resumable behavior of different tasks. Similarly, it is straightforward to add a delay to the `preempt` and `select` transitions to incorporate context switching overhead and to take a processor speed factor into account when calculating the delay of `execute` from the remaining execution time of the task in `execTask`.

5.4.4 Preemptive fixed-priority scheduler with priority ceiling protocol

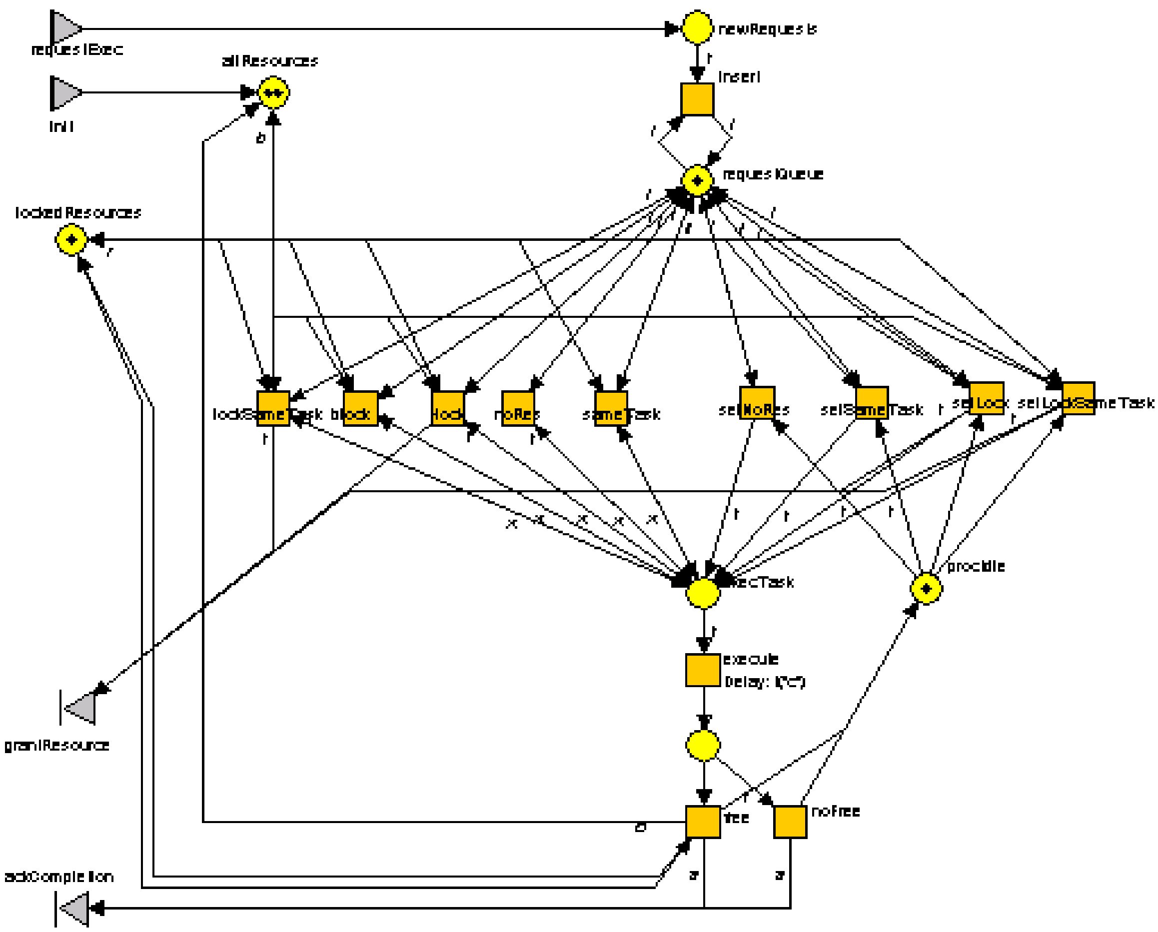

Fig. 26: Preemptive fixed-priority scheduler with priority ceiling protocol. The double headed arrows are read/write arcs. The screenshot shows a model developed with the Moses tool set.

The detailed presentations of the previous two schedulers served to explain the use of time Petri nets in dedicated scheduler subnets to model preemptive scheduling. In Fig. 26 the net of a preemptive fixed-priority scheduler with priority ceiling protocol (PCP) for controlling access to shared resources [129] demonstrates that more complex schedulers can be modeled with this approach as well. The model is similar to the scheduler of Fig. 25 with additional interface ports for initialization of the resource pool `allResources` (`init`) and sending of resource grants (`grantResource` (see Section 5.3)), but there are multiple selection and preemption transitions to take the different preemption and selection rules of the resource access protocol into account (no resource access, locking of a resource, blocking on a resource, resumption of a task which has previously locked a resource).

In this chapter the delegated execution modeling principle was presented and explained. According to delegated execution the system is divided into an untimed functional/structural task submodel, a timing information submodel, and a scheduler model which decides about preemption and time progress of the individual tasks. This separation of concerns on the one hand facilitates independent evaluation of submodels and on the other hand helps to solve the two problems that have up to now prevented the use of Petri nets for models of real-time systems with general preemptive scheduling strategies. Also in this chapter Petri net scheduler models for several scheduling algorithms were presented which make use of the semantics of time Petri nets in order to realize task preemption and resumption.

In the next chapter Petri net patterns will be introduced as abstraction mechanism for complex visual models. The example patterns from the real-time system application domain shown there implement the delegated execution principle.

Based on Petri net design patterns and the delegated execution model structuring principle a systematic approach to real-time system model construction will be developed in the Chapter 7.

The use of a common model consisting of submodels that are independently evaluated for different aspects of the behavior raises some questions regarding the validity of the obtained evaluation results. This issue will be addressed in Chapter 8.

6

Systematic model construction with patterns

This chapter motivates the use of design patterns for construction and comprehension of dataflow visual language models. After the introduction of the concept and a discussion of previous work on patterns in other contexts, a number of generally applicable Petri net design patterns are identified and an investigation into their use and relevance in published Petri net models is reported. A descriptional template for Petri net patterns is proposed and using it five Petri net design patterns, three of them real-time domain specific, are explained in detail.

6.1 Conquering model complexity

A visual system model can be very intuitive and is often helpful to communicate certain behavioral aspects to people even from diverse engineering backgrounds [84, 51]. However, complex visual models representing non-trivial systems are often difficult to construct and to understand [60, 48]. One reason for this is that model level behavior has to be expressed in terms of low-level primitives and the amount of low-level primitives in the model prevents from recognizing the interactions on a higher level. This is especially true for dataflow visual languages [77] such as Petri nets with a very small number of primitives several of which need to interact to implement even the most elementary behavior, such as alternatives or loops in the control flow.

There are several strategies to deal with the problem of model complexity:

- The most basic strategy is to have the structure, the *graphical layout* of the model reflect the stronger and weaker cohesion and interaction of certain model parts, progress of execution, or data flow (e.g. [48]). For non-visual models such as program code this corresponds to the goal of avoiding the `goto` statement.
- Model layout alone is usually not sufficient to master complex models, so *hierarchical structuring* was introduced. Hierarchical structuring allows to encapsulate certain parts of the model and to hide the internal structural complexity. State machines had their industrial breakthrough with the hierarchical State-Chart [73] and ROOM [139] formalisms, and also for Petri nets the need for hierarchy was recognized and several hierarchy and component concepts were

proposed [81, 53, 22, 85, 60, 48]. For program code this refers to the principles of information hiding and functional and object-oriented abstraction.

- Hierarchical structuring helps to cope with size, but the concepts of hierarchy and components are still only concerned with "which elements belong together". Hierarchy alone is not the complete solution [50]. An additional, orthogonal strategy is needed which is concerned with the question "what behavior does a collection of (possibly not collocated) elements represent", or, put as a synthesis problem, "I know what behavior I want to achieve. Which model represents this behavior?". This paper proposes the use of visual language *design patterns*, corresponding to program design and coding patterns, to capture and convey mechanisms and the purpose of certain submodels. Patterns can represent best practices to create models for certain application domains and help to avoid that a modeler new to a visual language needs to rediscover all mechanism realizations on his own.

6.2 Design patterns

6.2.1 What are design patterns?

Attempts to capture the constituents, underlying constraints, and architectural considerations of a design as so-called "patterns" have their origin in the work of the architect Christopher Alexander [2], and have recently become more and more popular in the software engineering domain, especially in the context of object-oriented techniques (e. g. [58], [31], [44], [170]).

A pattern in this sense is the description of a *recurring problem* or problem type and a *generalized solution* for this problem. Gamma et al. [58] state that a pattern consists of four essential elements: a name, a problem, a solution, and a description of consequences. A suitable and concise pattern name transforms the complex array of elements that make up the solution to an entity in its own right that can be manipulated and, more importantly, be communicated and discussed as a whole. The problem part of a pattern describes the situation that calls for the application of this particular pattern as well as conditions that need to be met to allow its use. The solution section presents a collection of elements and relationships that are necessary to solve the problem. The designer should regard the "solution" as a starting point and template which he will in most cases need to modify and extend to suit the context of his particular problem set [66]. Finally a pattern may contain a section discussing the consequences of the application of this pattern, the trade-offs and possible alternatives, to allow an informed decision between patterns that solve similar problems. This section will become more and more important after the pattern catalog for a particular design domain has reached a certain size.

Originally, the software pattern movement was strongly defending the view that the solution to a certain well defined problem may only be called a pattern if it was previously used in several (at least three) independent occasions. In an alternative view one might consider a solution to a problem presented in a certain form (pattern) and its quality - the fact that it actually constitutes a "best practice" (e.g. documented by known uses) - as two independent factors. This view has recently been gaining support in the software pattern community. All example 'patterns' in section 6.5 will thus be called patterns, even though there are some for which not yet any known uses outside our own work exist. Yet it has to be pointed out that those patterns were not artificially designed but resulted naturally from implementing the concepts of Chapter 5 in multiple models.

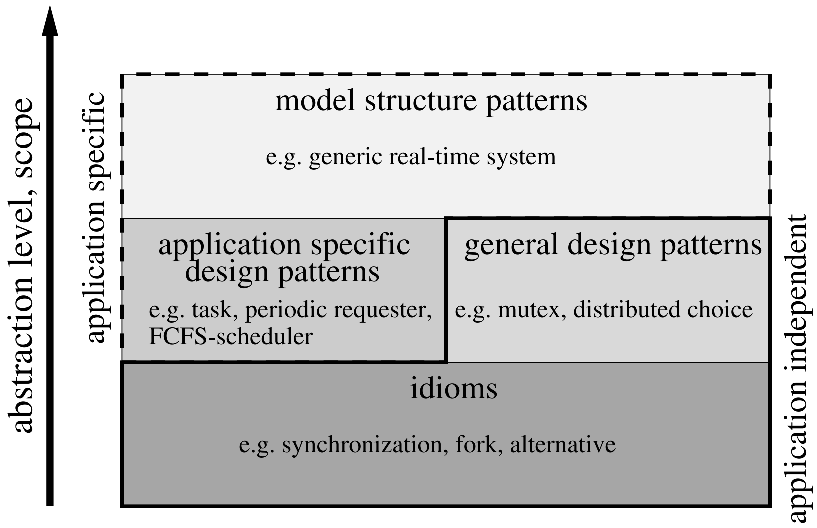

Fig. 27: Classification of Petri net patterns.

6.2.2 Levels of abstraction

Alexander introduced not only patterns, he also suggested to arrange patterns in the form of a so-called "pattern language" [2] to systematically guide the designer from the system level specification to a detailed design of a certain quality. A pattern language thus contains patterns at different levels of abstraction (see Fig. 27). The level of a pattern is determined by its internal complexity (structure and/or inscription), the number of other patterns it contains, and its scope with respect to the model in which it is used:

Idioms express basic mechanisms. They do not contain other patterns.

General design patterns combine idioms and other patterns to express higher-level mechanisms which are independent of any particular application or model domain.

Application specific design patterns are on the same level of abstraction as general design patterns, but their problem/solution pairs target a specific problem domain, e.g. real-time systems or flexible manufacturing systems.

Model structure patterns are concerned with the question of what the essential parts of a model are for a certain application domain. They do not necessarily contain any specific visual language elements on their own but mainly refer to other patterns. There are not many instances of these patterns within one model, and often only one.

The pattern examples in this paper are taken from all of these levels. They may be considered part of a pattern language for the modeling of real-time systems with Petri nets.

Ex. 6.2.1: *Fig. 28 demonstrates the visual aspect of Petri net patterns. It shows on the left hand side part of a model of a manufacturing system with two assembly lines which are served by a robot that is shared between both lines. The shaded areas highlight the patterns this model is composed of: Two instances of the* synchronization *and* fork *idioms and one instance of the* mutex *general design pattern which uses these idioms. The net structures of these pattern are also presented individually on the right hand side of the figure.*

In further investigation of several manufacturing system models it may turn out that the shown model is actually a domain specific pattern -flexible manufacturing

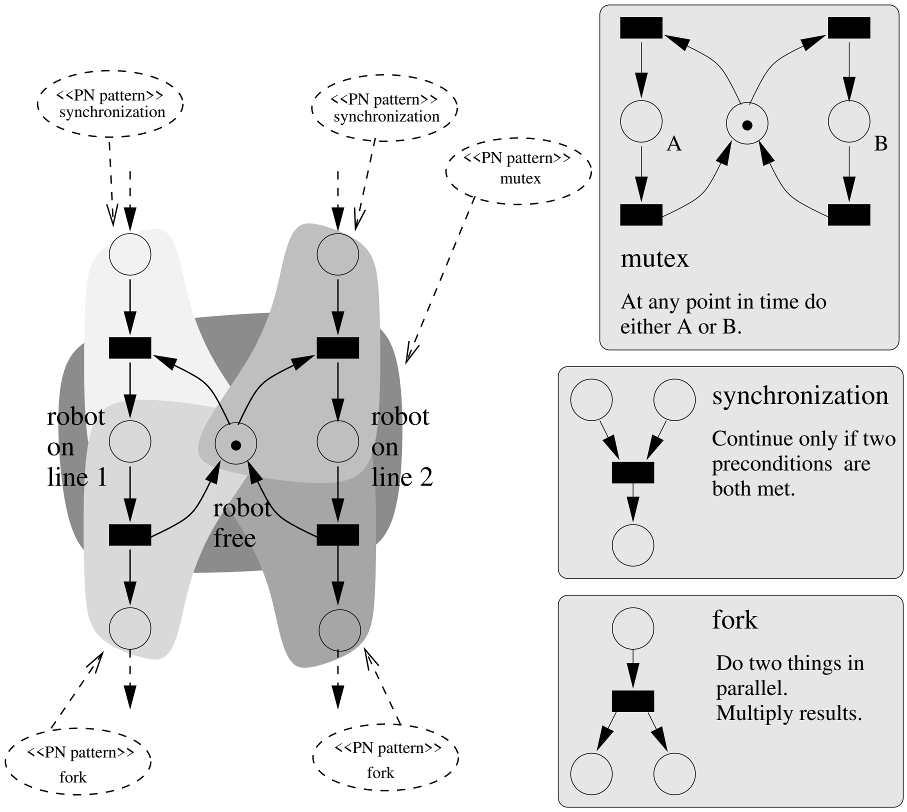

Fig. 28: Overlapping Petri net patterns in the model of a manufacturing system.

cell-, and an important part of a generic manufacturing system model structure pattern.

6.2.3 Patterns versus components

It might appear to be a logical next step to cast patterns into, possibly parameterized, building-block-like components [85]. However, this is usually not possible because there are fundamental differences between patterns and components:

- Patterns represent an abstract interaction schema and ordering structure between elements. The actual transformations/actions executed within the pattern participants will most likely be different for each use of the pattern.
- Components, following the principle of information hiding, have a narrow interface to restrict access to their contents. Patterns, on the other hand, are intended to be modified, even structurally, to solve a concrete problem. It is in the general case hard to predict which modifications will be necessary for a certain pattern in a certain situation.
- A pattern implements a certain behavior through interaction of its participants. As many model elements participate in several different mechanisms, both on equal (cooperating patterns) and on different (patterns use other patterns) levels of abstraction, Patterns usually overlap (see Ex. 6.2.1).

6.2.4 Previous work on design patterns for dataflow visual languages

To our knowledge there has been no previous description of patterns in the above sense for Petri nets or other dataflow visual languages.

In the introductory and theoretical literature to Petri nets (e.g [126, 76, 135, 136, 60]) a rather small body of reoccurring examples (e.g. producer/consumer, reader/writer, deadlock, dining philosophers) is frequently used to illustrate certain behaviors of a net or to describe structural properties for formal analysis. However, these examples are not used in the context of larger models and not intended to be used constructively. Some works suggest a building block approach to model building in different domains [18, 17, 35, 38, 175]. In [38] a combination of special purpose building blocks and net transformations is used to add fault-tolerant behavior to an existing system model. [175] constructs a model of a hardware architecture from a set of primitives.

The only other works relating design patterns and visual languages are the StateCharts patterns suggested in [50] in the context of the Unified Modeling Language (UML) [21] and the so-called Visual Design Patterns (VDP) of [165, 146]. The StateCharts patterns of [50] have the same intention as our work, the pattern structure, representation, and compositionality issues however are different as the StateCharts formalism is not a dataflow visual language. The Visual Design Patterns suggested in [165, 146] are, despite their name, generic, parameterizable templates for components. They can not partially overlap. Also, in both papers only one actual pattern example, a higher-order function for distributed computations, is given, and the important aspect of problem-oriented documentation of patterns (see Sec. 6.2.1) is not treated at all.

As long as patterns are used only as an aid to construct a model, there is no need to explicitly show them in the resulting model. This is different, however, if the patterns are also used as an aid to understand a model. Of the several options which are available to visually highlight patterns in a model, in this monograph, the notation of the Unified Modeling Language (UML) [21] for software patterns will be used: An oval carries the name of the pattern and dashed lines inscribed with the role names connect it with the most important participating elements.

6.3 General Petri net idioms and design patterns

From the literature and our own modeling work 33 non-application-specific Petri net idioms and patterns have been extracted (see Fig. 29), which use one or more of the 5 Petri net root idioms (*synchronization*, *fork*, *nondeterministic choice*, *OR*, and *sequence*). Root idioms are the primitives of Petri net model construction. In consequence, they are so general that highlighting them usually will not contribute to better model comprehensibility. Most of the patterns of Fig. 29 are, though not in pattern form, described in standard Petri net references (e.g. [126, 115]), some of the other pattern descriptions can be found in [119, 71] and Section 6.5 of this paper.

To support the claim that design patterns can be of substantial help for visual language model construction and comprehension, it is necessary to examine how prevalent patterns are in models, that is, to what degree models can be covered by patterns. Tab. 2 lists the patterns found and the model coverage - the percentage of Petri net places and transitions that participate in one or more of the identified patterns - achieved for a number of representative Petri net models from the literature. The following points had to be taken into account for the selection of the models to be examined:

- As patterns express behavior and not only structure, pattern discovery can not be done from the figure of the net alone. A detailed description of the model is necessary in order to be able identify patterns and to distinguish between patterns with similar structure but different intent and behavior.
- In calculating the coverage figures in Tab. 2 the five root idioms are not counted, because otherwise a 100% coverage is always achievable.

Pattern	simple LAN [110], p. 14	CPU [110], p. 154	FMS with continuous transportation system [110], p. 183	FMS with AGV [110], p.186	memory consistency protocol for multiprocessor system [103], p. 232	multiprocessor architecture [103], p.217	distributed self-diagnosis algorithm [171]	pipelined processor [140]	patient monitoring system [168]	die bonder [84]	multiprocessor node [180], p. 6	multiprocessor system [93]
acknowledgement								X		X		
alternative							X			X		
clock								X	X			
concurrency	X					X			X	X		
controlled action		X			X		X		X			X
loop	X	X										
mutex			X	X	X	X		X			X	X
nondeterminism		X	X	X					X		X	X
quasi-cont. movem.										X		
procuder/consumer			X	X						X		
resource		X	X	X					X		X	X
terminated loop									X	X	X	
token occ. switch	X			X	X					X		
variable										X	X	
number of net elements	30	27	61	51	33	66	17	30	61	192	32	37
element coverage (%)	100	62	75	84	39	60	82	73	77	42	60	70

Tab. 2: Petri net design patterns and model coverage in selected papers.

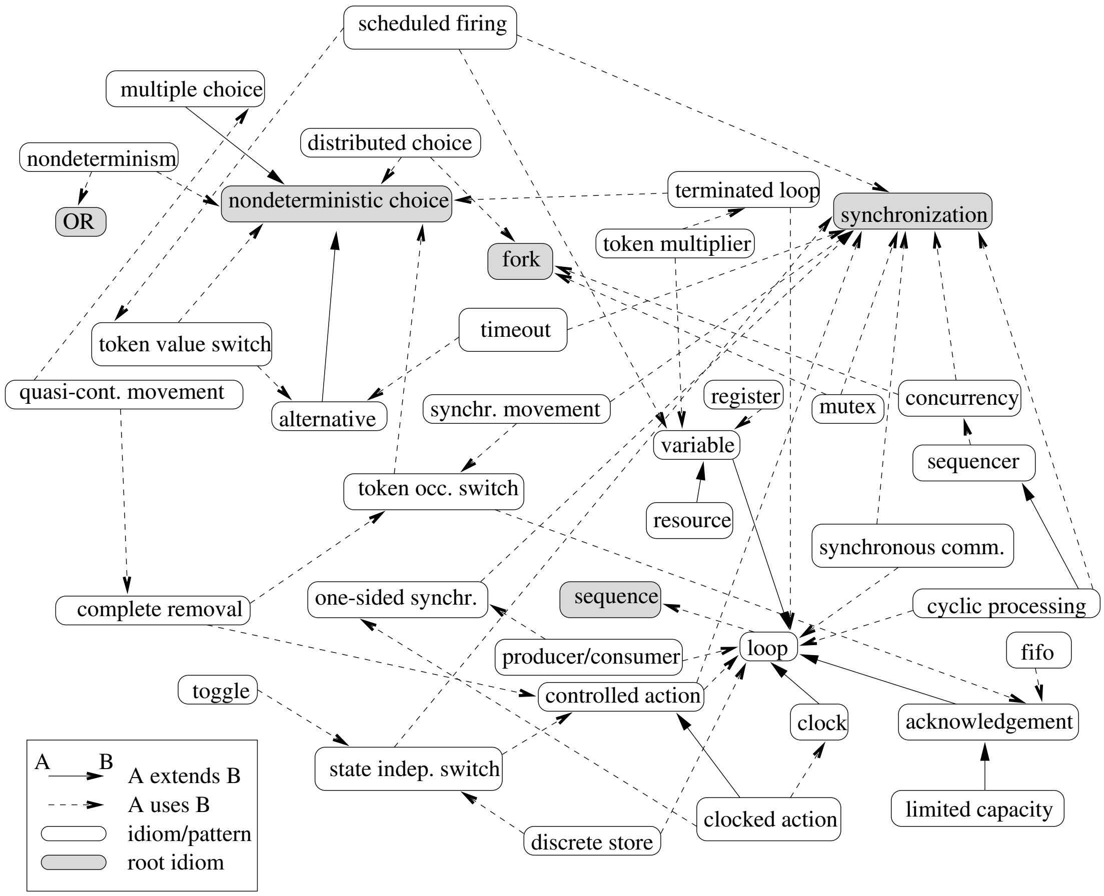

Fig. 29: Basic Petri net design patterns and their dependence relationships.

The table shows that for most models a rather high coverage of 60% to 100% is achieved. The reason that the model of the manufacturing machine [84] reached, despite the relatively high number of patterns found, only a coverage of 42% is that in this model linear sequences of processing steps are a distinctive feature, but the *sequence* idiom was, as explained before, not included in the coverage calculation. An interesting result is that the model with the lowest coverage of 39% is one of the two models that represent an algorithm and not a physical system. Further research will be necessary to investigate whether a different set of yet unidentified patterns is necessary for models of algorithms, or whether the uniqueness of larger parts of the model is a characteristic feature of this kind of modeling problem.

6.4 Petri net pattern template

An important issue when discussing patterns as a design aid is how they are documented. The requirements for a pattern description and thus for the description template are:

- A description must convey enough information about problem and solution, therefore it must be *detailed*.
- One should be able to quickly survey a number of pattern descriptions, therefore they must be *concise* and *structured*.
- The descriptions should facilitate comparisons between patterns, therefore they must use a *uniform* template.

This paper suggests a description template which follows the four principal elements of a pattern that were mentioned in the previous section. Several quite similar pattern description templates exist in the literature. The one used here puts more emphasis on the description of the solution than [138] and less on object-orientation and implementation issues than [58, 31]. Some sections of the template are optional and may not apply to certain patterns.

Name block:

Name: A name to identify the pattern and distinguish it from others. The name should be such that it clearly conveys the main idea of the pattern.

Problem block:

Problem: The problem and problem context which the pattern addresses.

Forces: Factors which may influence whether a solution is considered more or less appropriate in a certain context.

Example: A concrete example of the problem within some application domain.

Required net formalism: Most of the patterns in this paper rely on the availability of certain extensions of the basic Petri net formalism, such as high-level Petri nets with token values, guards, and actions.

Solution block

Solution: The basic idea of the pattern (if non-trivial)

Sample structure: A graphical representation of a Petri net that implements/uses the pattern. In the case that the sample structure is a neutral skeleton, the net elements shown are to be understood as representatives of concrete pattern participants in certain roles. In the case the sample structure is taken from a concrete example, the description needs to make clear which elements have roles as pattern participants and which elements belong to the more complex behavior of the example only.

Description: A detailed description of the function of the Petri net building block, also discussing design considerations, variations and options. As far as possible, the explanation should make use of other patterns contained within the pattern under consideration.

Known uses: References to models that use this pattern.

Interacts with: References to other patterns with which this pattern commonly collaborates.

Variants: Modifications of this pattern which solve different problems. Depending on the scope of the modification this section may explain the whole variant or be only a reference to its full pattern description.

Consequences block:

Uses: References to other patterns that are contained within the described pattern.

Similar to: References to and comparisons with other patterns that are similar in some aspect like net structure or targeted problem.

Further sections may be used to highlight special aspects or trade-offs of using a certain pattern.

6.5 Real-time system patterns

This section gives an example catalog of five patterns that will be used later for the systematic construction of real time system models. This catalog contains patterns from all four abstraction levels introduced above.

6.5.1 Idiom: Alternative

Name: *Alternative*, also known as deterministic choice, structural conflict

Problem: Token flow shall be directed into different paths depending on the value the token carries.

Example:

- branch in control flow where token represents current program pointer,
- routing of material flow in a manufacturing system or messages in a communication systems,
- any kind of decision.

Required net formalism: High-level Petri nets with token values and guards

Solution: Use two different transitions, each with a different guard.

Sample structure: See Fig. 30.

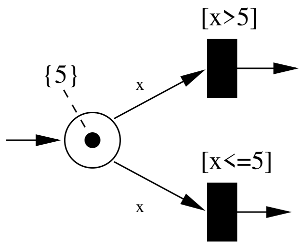

Fig. 30: Alternative pattern

Description: The transition for which the guard condition is true will fire.

Known uses: Almost all practical high-level Petri net models use this idiom directly or indirectly.

Variants: More than two transitions may be used to realize more than two distinct choices. If the guard conditions of two or more transitions overlap so that both may fire according to Petri net semantics (a so-called conflict), the choice will be made non-deterministically. Non-deterministic decision is also the standard case for basic Petri nets without token values and guards.

Similar to: See *distributed choice.*

6.5.2 General design pattern: Distributed choice

Name: *Distributed choice*

Problem: Token flow shall be directed into different paths depending on the value the token carries.

Forces:

- For some reason (e.g. model of a distributed system), the alternative transitions can not be in direct structural conflict, that is, they do not have the same input place.

- The decision how to handle a token shall be made locally in a submodel.
- One or more receivers may accept the token.

Example: A communication system is modeled. The token representing the message carries the a value denoting the address of the destination submodel. Only the receiver that matches the specified address shall process the message token.

Required net formalism: High-level Petri nets with token values and guards.

Solution: After broadcasting the token to all possible receivers, each receiver determines whether it has to process the token.

Sample structure: See Fig. 31.

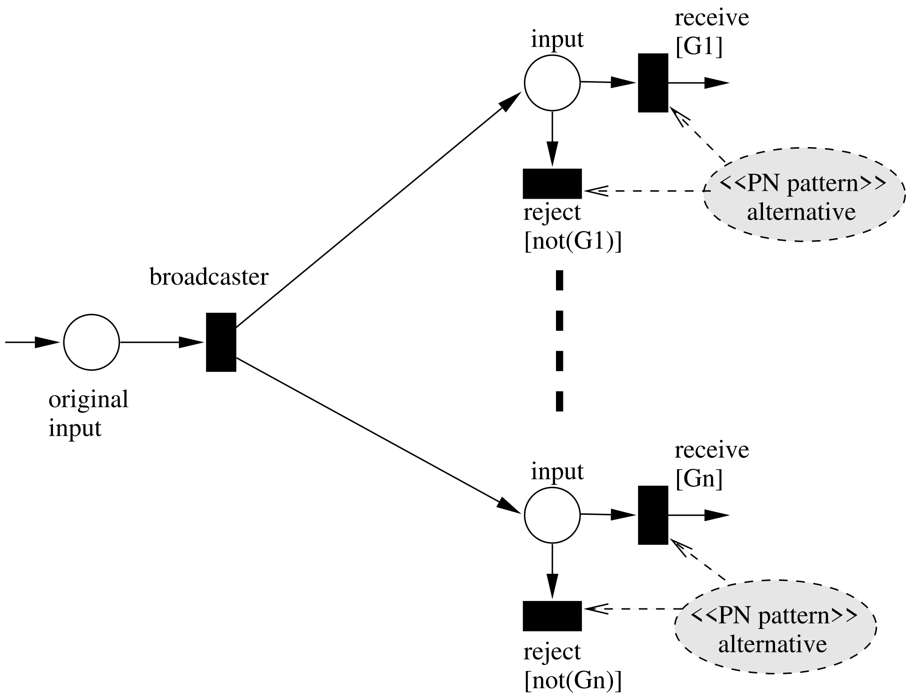

Fig. 31: Distributed choice pattern

Description: One transition broadcasts the original token to the input places of all receivers. The `receive` transition of receiver i fires if the guard condition G_i is true, otherwise the `reject` transition with guard not(G_i) fires and clears the receiver input place.

Known uses: The memory controller model of [70] uses this pattern to model RAM cell addressing [71]. The delegated execution modeling style (see Sec. 5.2) uses this pattern to send resource grant notifications and completion acknowledgements.

Variants: The fact that the token is broadcast to all receivers allows multicast semantics, that is, several receivers with overlapping guard conditions G_i may process the token.

Uses: An *alternative* pattern with two alternatives in each receiver.

Similar to: While this pattern is functionally similar to the *alternative* pattern, it is applicable in situations where the alternative transitions have no direct access on the original input place. Also, overlapping guards do not lead to a conflict and nondeterministic selection of one receiver, but to the deterministic firing of several receiver transitions. This pattern can also be used to implement an alternative where the decision about acceptance is encapsulated locally in a submodel, which is particularly relevant for hierarchical and component-based modeling formalisms (e.g [53]).

6.5.3 Application specific design pattern: Task segment

Name: *Task segment*

Problem: A real-time task shall be represented as a Petri net.

Forces:

- Support of preemptive scheduling without excessive structural overhead.
- Generic task structure for periodic and aperiodic tasks.
- Support access on shared resources while making sure that the resource is not accessed before the permission for access has been granted by the scheduler.
- Any actions represented by the task body model shall be executed only after task execution has actually been started for the first time, not already when the request has been sent to the scheduler.
- Task body model shall be as small as possible.
- Separate verification of functional correctness using formal Petri net verification methods shall be possible.

Example: This pattern is applied several times in the mine drainage system case-study in Sec. 9.

Required net formalism: High-level Petri nets with token values and guards.

Solution:

- Separation of control flow (functional aspect) and time progress management (scheduler) compatible with the delegated execution modeling style of Sec. 5.2 and the *generic real-time system* pattern presented below.
- The functional net can use black tokens.
- The pattern consists of a main part (functional net) and an optional part (task timing interface) that connects the task structure to a scheduler for timed evaluation of the system. The task timing submodel itself uses untimed high-level Petri nets.

Sample structure: See Fig. 32.

Description: Due to the variety of possible task configurations (task chains, resources) the *task segment* pattern has many variants. The main distinctions between these variants (see **variants** below) relates to whether the task segment accesses a resource and whether it is the pattern of the functional or the complete model of the task.

Fig. 32 shows the complete model of a task segment, with task structure submodel and timing interface submodel, which requests and accesses a resource within the same segment. This pattern has been explained in Sec. 5.3.2 (see also Fig. 23).

The mechanism to filter out all those broadcast message tokens (resource grants and completion acknowledgements) that are adressed to a particular task is realized using the *distributed choice* pattern.

Variants: Fig. 33 divides the task segment model of Fig. 32 into six functional areas. Tab. 6.5.3 associates these functional areas with different task configurations and modeling situations.

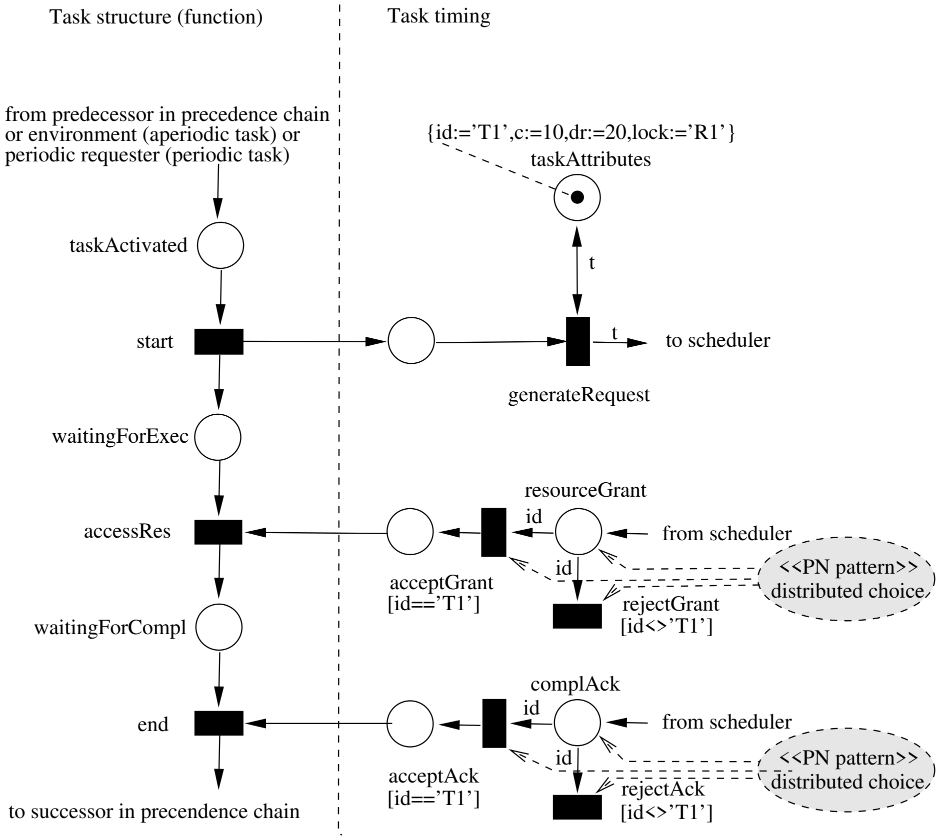

Fig. 32: Task body with resource

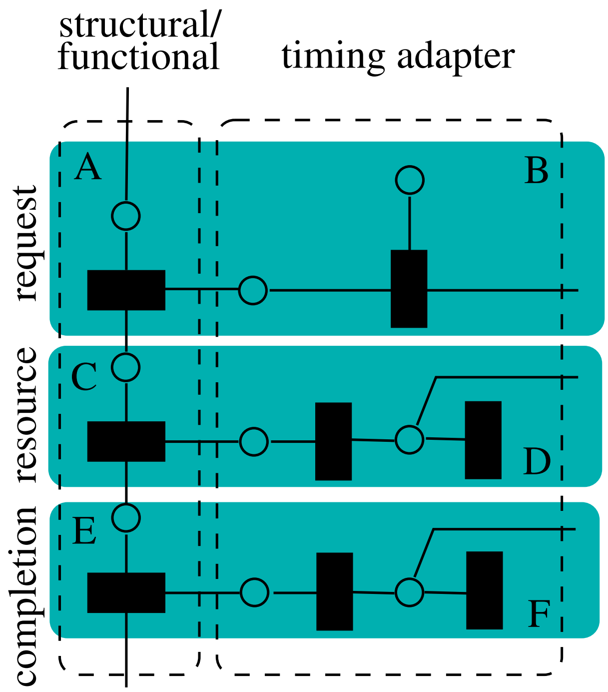

Fig. 33: Functional areas of the *task segment* pattern

The task segment consists of three stages, request stage (A,B), resource access stage (C,D), and task completion stage (E,F). The B,D, and F parts of the model are the timing adapters of each stage. They are dropped for a purely functional model. The request stage is necessary for every task

task configuration			modeling situation	
			functional net	complete net
no resource access	final segment/ independent task	action	A,E	A,B,E,F
		no action	A	A,B
	middle segment of task chain	action	A,E	A,B,E,F
		no action	A,E	A,B,E,F
resource access	final segment/ independent task	action	A,C,E	A,B,C,D,E,F
		no action	A,C	A,B,C,D
	middle segment of task chain	action	A,C,E	A,B,C,D,E,F
		no action	A,C,E	A,B,C,D,E,F

Tab. 6.5.3: Correspondence between task configurations and functional areas of the *task segment* pattern.

configuration. The resource access stage is left out for segments that do not access a shared resource. Any modeled task action would in this case be associated with transition `end` of the task completion stage. The task completion stage is not necessary for a task that is at the end of a task chain and for which no task action is modeled.

The `start` and `end` transitions of consecutive task segments in a chain can be fused.

Uses: *Distributed choice* for the communication of resource grants and completion acknowledgements between all tasks in the model. *Task segment* is an extension of the *sequencer* pattern: A *sequence* is used to make sure that concurrent activities in a submodel are executed in the right order.

Interacts with: In the context of a *real-time system* the task body of a periodic task may rely on a *periodic requester* to be activated periodically.

6.5.4 Application specific design pattern: Periodic requester

Name: *Periodic requester*

Problem: A signal (token) needs to be emitted periodically.

Forces:

- The emission must only start after an initial delay.

Example: Periodic activation of tasks.

Required net formalism: High-level temporal Petri net, both timed and time Petri nets may be used.

Solution: Self-loop with a timed transition, and an initial transition which is fired only once.

Sample structure: See Fig. 34.

Description: A token carrying the timing information - here an initial delay (phase) of 10 units and a period of 50 units - is initially placed in `initial`. After the amount of time corresponding to the initial phase has passed, a black token is put into the `output` place which is part of the model that is to be driven by the periodic requester, and the colored token with timing information is placed into `waiting`. From now on, `repeat` will fire every time the specified period has passed, producing an output and starting an new waiting period.

Known uses: *Periodic requester* is part of the model used in [117] for the evaluation of the performance of a fault-tolerant multi-processor scheduler.

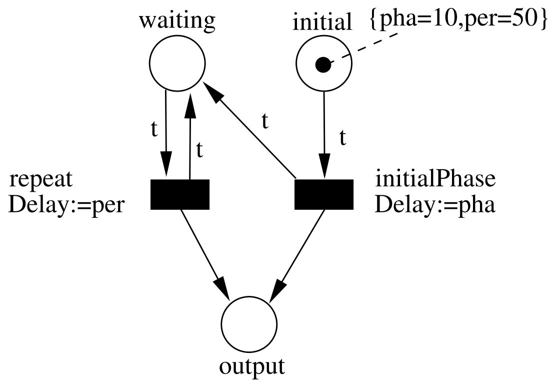

Fig. 34: Periodic requester pattern

Variants: If the temporal Petri net formalism used allows overlapping waiting times, several periodic requesters may be folded so that only one net is needed to produce tokens with different phases and periods. The output tokens would then need to carry some identification so that the following model elements can distinguish the different requests.

Uses: *Periodic requester* is obtained by extending the *clock* pattern with a mechanism to allows for an inital phase.

6.5.5 Model structure pattern: Generic real-time system

Name: *Generic real-time system*

Problem: How to model a real-time system with tasks, environment, and realistic schedulers with Petri nets?

Forces:

- Model should be able to integrate hardware, software, and system environment.
- Preemptive scheduling and shared resources are to be modeled without excessive structural overhead.
- Combination of one task model with different schedulers implementing different scheduling policies.
- Easily understandable model structure.

Example: See section 9 for the description of an example system and its model.

Required net formalism: The structure pattern does not contain any Petri net elements itself. The generic model may even be used to combine several visual languages within one model.

Solution: Each model consists of parts for task invocation, task structure, task timing information, schedulers, and the system environment with I/O components. A clearly defined interface exists between the task model and the scheduler model.

Sample structure: See Fig. 35.

Description: The model structure of Fig. 35, an extension of Fig. 22, reflects the fact that an embedded real-time system consists of an environment, an application task structure, and a processor/scheduler component. The generic model integrates and builds on the conventional task graph model commonly used for schedulability analysis and extends it with a description of task and environment functionality.

In order to be able to model preemptive scheduling in the general case it is necessary to separate the submodel representing the task functionality from

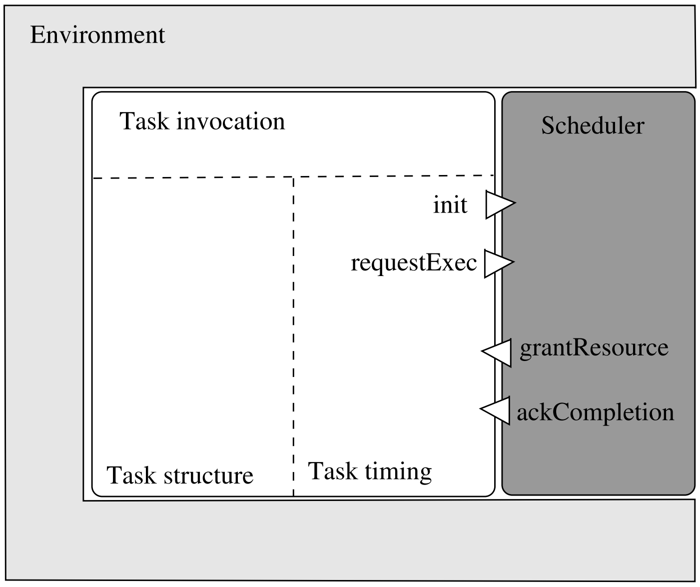

Fig. 35: Generic structure for graphical model of an embedded real-time system

the part that controls execution progress (see Sec. 5.1). The task structure, task timing, and scheduler submodels and their interactions to implement this separation in the form of the *delegated execution* modeling principle have been described in Sec. 5.2.

The task invocation submodel determines the actual points in time when tasks arrive. It can be part of functional net, in the case of free-running tasks, part of the environment in the case of aperiodic tasks driven by external stimuli, or an independent subsystem that implements other invocation strategies.

Known uses: The models of Section 9 and [117] were constructed using this pattern.

Variants: Not all parts of the generic model may be necessary for an actual model. For example, if all tasks are aperiodically triggered by the environment, the task invocation submodel will be empty.

Concrete submodels implementing certain parts of the generic model need not be contiguous. For complex systems the organization of the application task subnet will follow problem specific structures, e.g. the physical layout of the resources, and a hierarchical model will be used with components, that implement variants *task segment* pattern and thus have their own task structure and task timing parts.

Uses: Possibly uses any of the patterns described above. This pattern is the starting point for the application of a pattern language on real-system modeling with visual, Petri-net-like languages.

In this chapter visual dataflow language/Petri net design patterns have been introduced as new model structuring principle orthogonal to spatial and hierarchical structuring. The design pattern abstraction helps to structure complex models both during model construction and for later model comprehension. Several general and real-time system specific Petri net patterns have been identified. The relevance of the former for Petri net models from the literature has been investigated and the latter have been documented in detail, as the modeling and evaluation methodology presented in the next chapter is based on them.

7

Modeling methodology

This chapter describes a process for the construction and evaluation of models of real-time systems. The patterns introduced in Section 6.5 are used to assemble and structure the underlying model.

7.1 Steps of the model building and evaluation process

The proposed modeling and evaluation process (see Fig. 36) consists of four major steps

- model construction,
- evaluation of the untimed model (functional verification),
- addition of timing, scheduling, and resource access information, and
- evaluation of the timing behavior,

the first of which is further divided into three minor steps. The individual steps of the methodology will be explained in the following sections. During a system development cycle the designer will most likely need to cycle through these four steps more than once, taking more detailed specifications, system extensions, and modifications into account.

An example for the application of this methodology to a system is given in Chapter 9.

7.1.1 Construction of the untimed system model

Most system evaluation approaches using Petri nets or other formal methods neglect the problem of how to construct the actual model to be analyzed. This section describes a systematic three step approach to model construction.

In the first step, a *task diagram* is constructed from the system requirements and specifications in the problem domain.

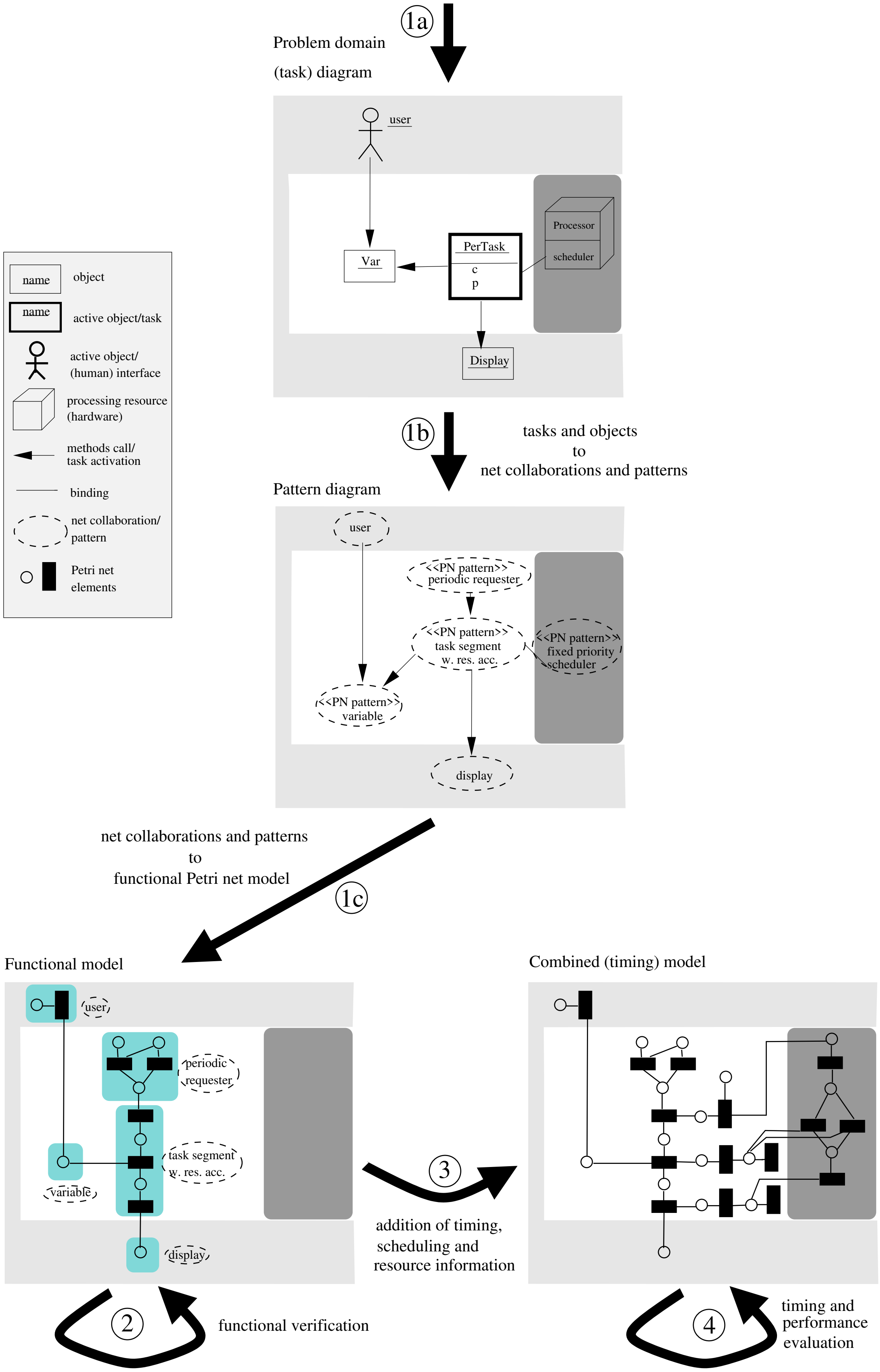

Fig. 36: Steps for model construction and system evaluation.

The proposed task diagram (see top of Fig. 36) extends the conventional task graph model with elements denoting objects operated upon such as variables, sensors and actors in the environment, and processing resources. The task diagram uses the notation of the Unified Modeling Language (UML) [21]: Thick-lined boxes are active objects, which represent tasks with their attributes and behavior, thin-lined ones are other, passive objects both in system and environment (e.g. memory locations or I/O ports). Stick figures represent an active influence on the system from the environment. Three-dimensional boxes represent the underlying, non-application-specific aspects of the system, such as the combination of execution resources (CPUs) and operating systems with schedulers. Edges with arrows are used to express task activations and method invocations on objects, edges without arrows associate each task with an instance of the underlying execution platform. The diagram layout follows the *generic real-time system* pattern introduced in Section 6.5.5.

To convert the task model into a Petri net the *pattern diagram* is created in an intermediate step.

The pattern diagram consists of UML collaborations and patterns, graphically represented as dashed ovals with inscriptions. A collaboration is a collection of visual language (Petri net) elements that interact to implement a certain behavior or functionality. A pattern is a collaboration that is documented, for example in the way described in Chapter 6. A pattern is a collaboration that may occur in several models, while the net representations of all functions specific to the system under evaluation are non-pattern collaborations. In the diagram patterns can be distinguished from non-pattern collaborations by the additional inscription (UML stereotype) '<<PN pattern>>'.

Due to the fact that a pattern is a kind of collaboration, this diagram would more suitably be called 'collaboration diagram'. However, this term is in the UML already reserved for a different type of diagram that contains no collaboration elements but describes the sequence of interactions between objects.

The pattern diagram is transformed into the *functional Petri net model* by replacing each collaboration with the corresponding net structure, using the documented net structures of patterns where applicable and manually modeling application-specific functionality for the remaining collaborations. This part of the modeling process corresponds to the implementation of the system - though on a higher level of abstraction - and can not be automatized. Labeling the resulting net elements with their generating collaborations improves readability and comprehensibility of the net model.

7.1.2 Evaluation of functional behavior

The functional model is used to evaluate functional behavior. The means for evaluation range from animated execution which visualizes token flow and mainly supports the debugging process via unattended simulation of which only the output and condensed information collected during the simulation run are of interest to formal verification of functional system properties using the full range of Petri net analysis techniques [87, 171, 75].

In the pure functional net the execution order of parallel Petri net branches is not constrained by any schedule or limited execution capacity. Thus the system is represented in a form that considers all possible schedules. Also, due to transition firings without delay it includes the (unrealistic) behavior of instantaneous reactions on stimuli from the environment. This reflects the fact that in an early phase of the system development process many functional requirements will already be known, but without an actual implementation most timing and scheduling parameters, such as allocation of processing resources, worst-case execution times, priorities, etc will not.

Most system models are too complex to allow full formal analysis by the means available today [169]. One solution is to restrict the formal verification to meaningful submodels. Also, net reduction techniques (e.g. [148, 19]) help to reduce the complexity for analysis, as the functional net will usually have high structural redundancy.

The kinds of analytical results obtained from the functional analysis are formally characterized in Chapter 8. In general, there are two types of useful analytical results:

- The system does not meet a certain functional requirement no matter which schedule is used. This requires a redesign of the system and another iteration in the modeling and evaluation process.
- The system may sometimes not meet a certain functional requirement, e.g. due to concurrent resource access. From this result requirements and parameter settings for the timing, scheduling, and resource access attributes of the system can be derived to eliminate the non-determinism and enforce the desired behavior (see Chapter 9 for an example).

7.1.3 Augmenting the model with timing and scheduling information

After the evaluation of the functional behavior of the model and when the implementation dependent timing and scheduling parameters are known and validated using scheduling analysis methods such as the approach presented in Chapter 3, timing information is added to the system model by replacing the functional nets of all task segments with their extended, timed counterparts and connecting these to a model of the chosen scheduling strategy. Both the functional and the complete Petri net representations of the task segment pattern have been presented in Section 6.5. With known task timing and resource parameters the transformations between these two pattern net structure versions are completely automatable in both directions.

Ex. 7.1.1: *Fig. 37 shows an example for these transformations: In the combined net the* `start` *transition triggers the* `generateRequest` *transition to send a request token to the scheduler net. This request token carries data about the task parameters relevant for scheduling and resource access. In this example, the task has the id 'T1', a worst-case execution time of 10 units, a relative deadline of 20 units, a priority of '3', and it requests exclusive access to a resource'R1'. After the task requests a resource, the successful locking is acknowledged by the resource access protocol in the scheduler net with a resource grant token. It is broadcast to all task segments and rejected by all except for the task that sent the request, where the arrival of the grant message leads to the enabling of transition* `accessRes`*. Once the processor has completely executed a task, it broadcasts the completion acknowledgement token, which enables transition* `end` *of the corresponding task structure in the functional net.*

In addition, the model may be extended by places and timed transitions to mark the violation of timing constraints or the occurrence of other undesirable system states. Different kinds of timing constraints for real-time systems are listed in [47].

7.1.4 Evaluation of timing behavior

Formal verification methods break down for models that involve timing/scheduling, non-determinism, and operations on data. Simulation - despite its limitations -

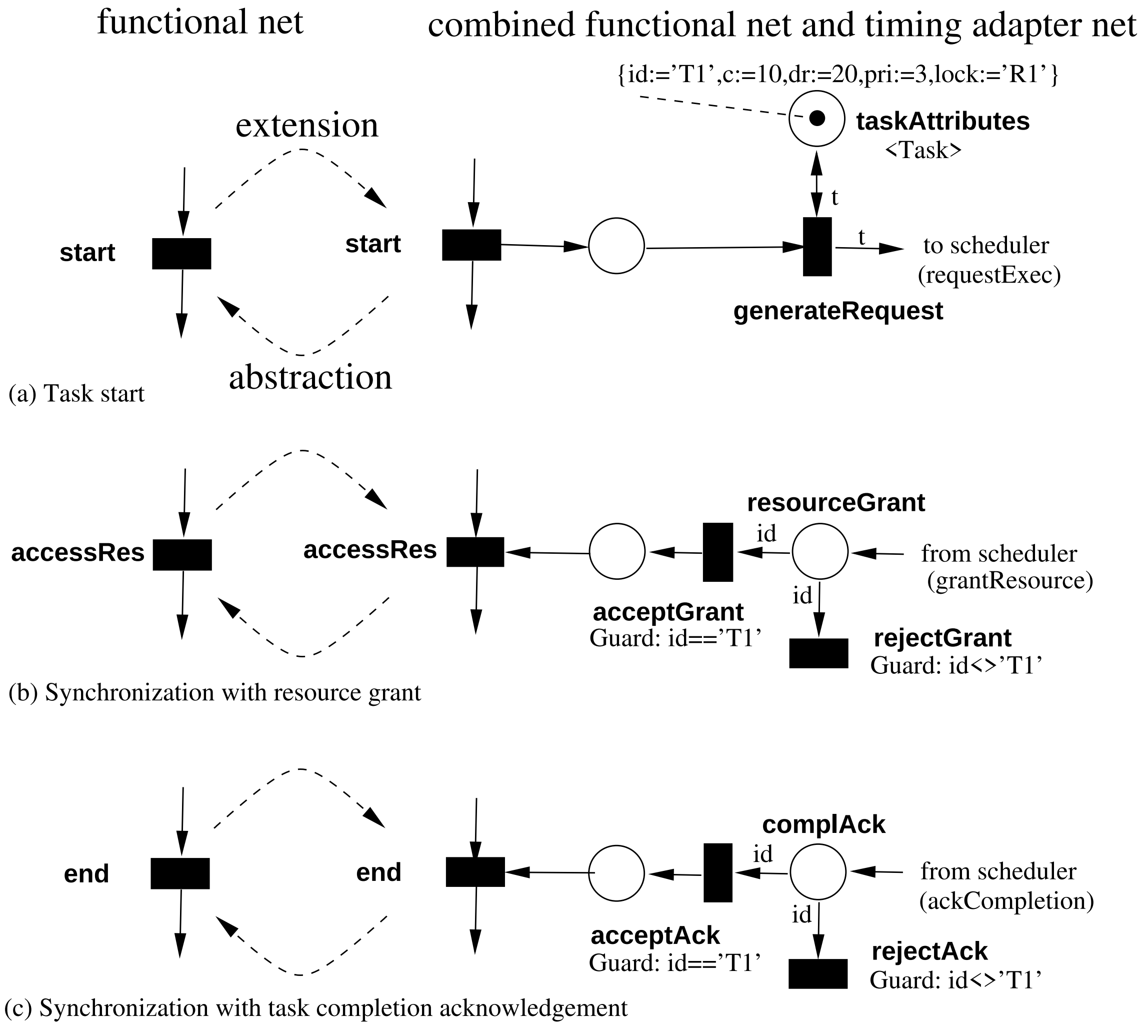

Fig. 37: Transformations between the functional net and the functional net with timing adapter net. The initial token shows the attribute values of the token object of type <`Task`>.

thus remains as the only feasible method to gain some confidence in the correctness of the systems timing behavior [114]. The simulation of a timed model is also a suitable method to evaluate system performance and throughput.

The temporal Petri net model of the application constructed in the previous step together with a scheduler model (see Sec. 5.3.3) can be executed to simulate the system under evaluation. In Chapter 9 it is shown how execution traces produced by simulation are used to identify a scheduling strategy that leads to a deadline miss.

Exhaustive simulation is usually not possible, therefore it is necessary to plan in advance what the interesting situations to be simulated are. The results of the functional verification step can be used as starting points.

Both in the functional and the combined model appropriate structures can be added to model [171] or inject [80] system faults and evaluate the effects on system behavior.

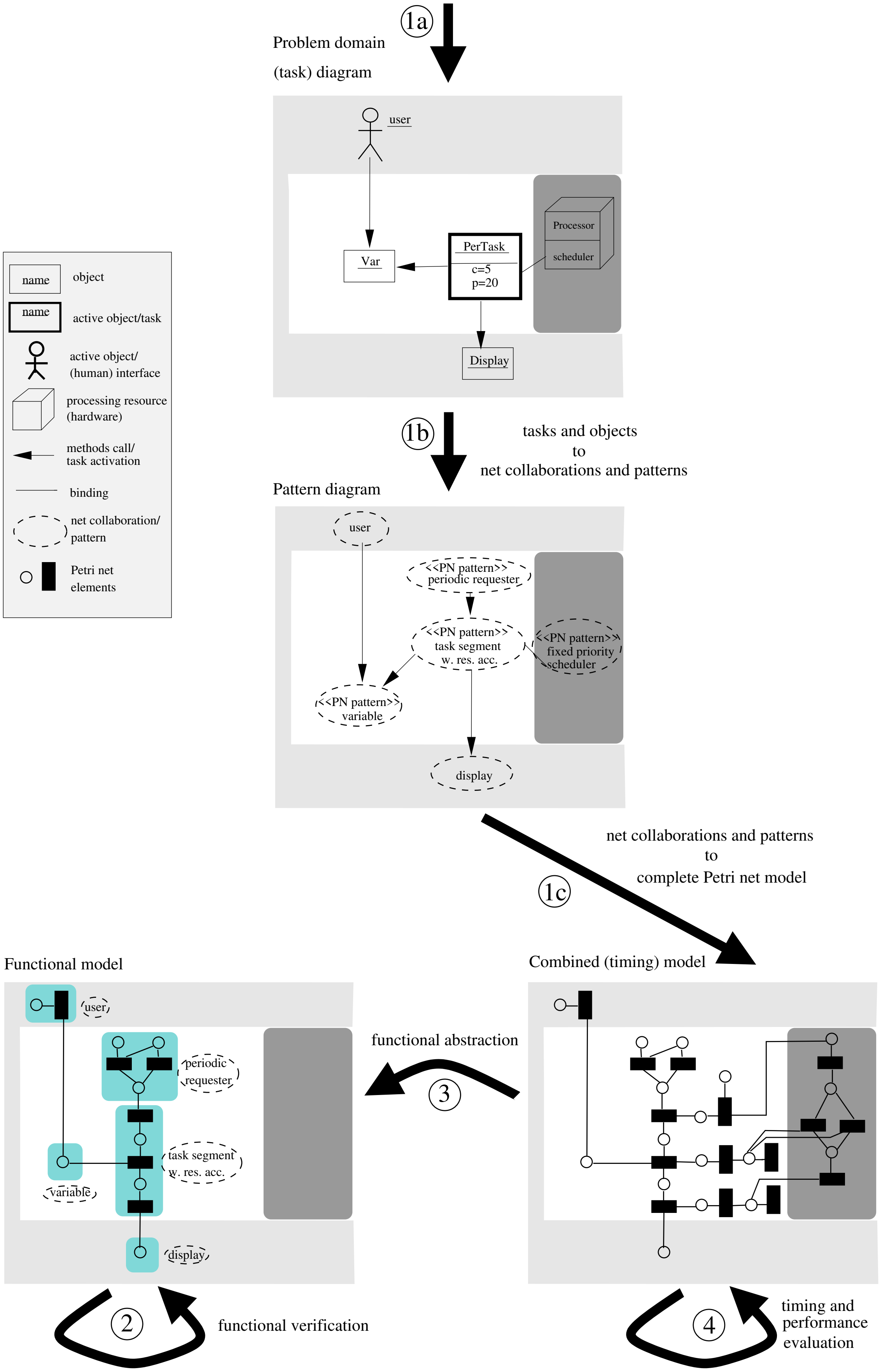

Fig. 38: Alternative modeling workflow: Direct construction of the complete model and evaluation of an untimed abstraction.

7.2 Direct construction of the combined model

Usually, untimed functional verification will be conducted before timing evaluation, following the process described above, as timing information is typically available only later in the design process, and some requirements for the timed and scheduled system, e.g. with respect to mutual exclusion, are derived from the results of the functional analysis. However, pattern-based construction also allows to directly construct the complete Petri net model from the pattern diagram (see Fig. 38). Then, in case functional properties are to be verified, the transformations of Fig. 37 can be applied to the combined model in the inverse direction, thus generating an untimed abstraction of the complete model to apply formal verification.

In this chapter the four steps of the methodology for the modeling and evaluation of real-time systems proposed in this monograph have been summarized. It has been explained how patterns are used for systematic model construction and how various evaluation methods interact with the model.

8

Validity of formal functional verification results

In the methodology proposed in Chapter 7 formal verification methods for system functionality are applied to a model which carries only partial information about the system, therefore the question arises whether these analytical results are also valid for the combined model with timing and resource constraints, and thus for the system under evaluation. This chapter investigates that question. It defines and proves the types of valid analytical statements about the combined and timed model which can be made from a functional analysis of the untimed abstraction.

8.1 TPPT-partitioning of a Petri net

In order to discuss the relationship between and the properties of the functional and the combined model, the notion of a TPPT-partitioning is introduced:

Def. 8.1.1: **(TPPT-partitioning)** *Let $N_C = \{T_C, P_C, F_C\}$ be a Petri net system with transitions T_C, places P_C, and flow relation F_C that is partitioned into two subnets $N_F = \{T_F, P_F, F_F\}$ and $N_S = \{T_S, P_S, F_S\}$ such that $T_F \bigcap T_S = \emptyset \bigwedge P_F \bigcap P_S = \emptyset$. This partitioning is called a TPPT-partitioning with* T-partition *N_F and* P-partition *N_S, if for all edges connecting N_F and N_S the following two conditions hold:*

$$F_C \bigcap (T_S \times P_F) = \emptyset$$

$$F_C \bigcap (P_F \times T_S) = \emptyset$$

Fig. 39 illustrates this definition. All connecting arcs between the T-partition N_F and the P-partition N_S start at a transition in the T-partition and end at a place in the P-partition ($a_1 \in T_F \times P_S$), or they start at a place in the P-partition and end at a transition in the T-partition ($a_2 \in P_S \times T_F$).

With the net elements produced by the transformations of Fig. 37 to connect functional model and timing/scheduler subnet the functional model is obviously a T-partition of the complete model.

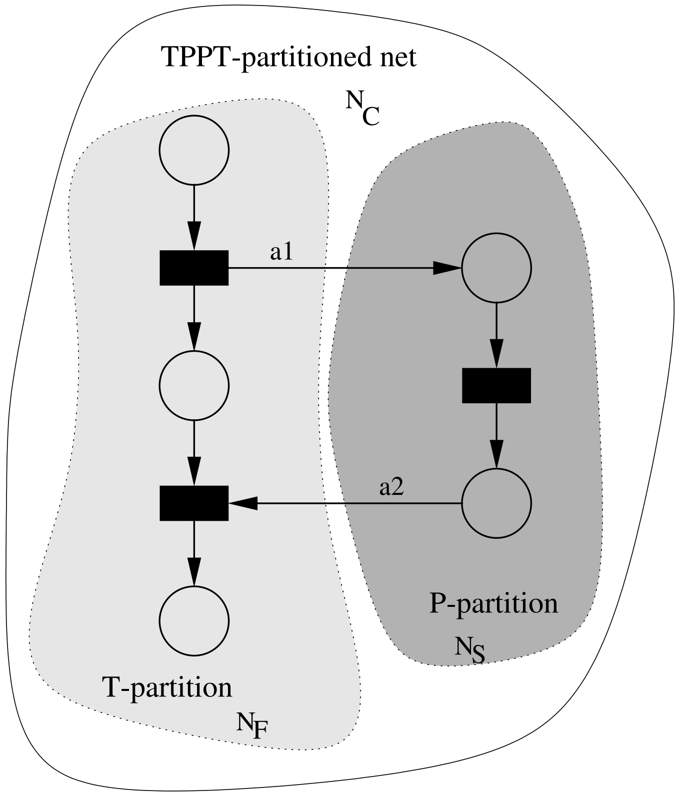

Fig. 39: Example of a TPPT-partitioned net.

8.2 Central property of the partitioned model

The central property of the transformation rules of Fig. 37 is that the behavior which is possible for the functional net is restricted by the scheduler net in the combined model. This property is formally expressed in the following Theorem 8.2.1, which is formulated using the previously defined TPPT-partitioning:

Theo. 8.2.1: *Let N_C be a Petri net system that is TPPT-partitioned into a T-partition N_F and a P-partition N_S. Let R_F and R_C be the sets of all reachable markings m_F and m_C of N_F and N_C respectively, starting with equal initial markings $\forall p \in P_F$. Let $R_{F,C}$ be the set of markings m' with $m' : P_F \longmapsto \mathbf{N}^1$ and $\forall m_C \in R_C, \forall p \in P_F : m'(p) = m_C(p)$. Then the following holds:*

$$R_{F,C} \subseteq R_F \tag{8.1}$$

The entities observed in this theorem are the functional net N_F, the scheduler net N_S, and the combined net N_C. If N_F is the T-partition and N_S the P-partition of a TPPT-partioning of N_C, then the set of possible markings (the behavior) of the places of N_F with N_F being part of N_C is a subset of the set of possible markings of the places of N_F with N_F being observed on its own. m' is the partial marking observed on the places that belong to both N_F and N_C. In the general case, several marking m_C of R_C, corresponding to different partial states of $N_C \backslash N_F$, will be mapped to the same marking m'.

Proof: There are two ways how additional markings for the reachability graph of $R_{F,C}$ as compared with the one of R_F could be generated: A conflict between a transition $t_f \in T_F$ and $t_s \in T_S$ would possibly cause a new branch, or a transition $t_s \in T_S$ would inject an additional token into N_F. Due to the structure of the TPPT-partitioning it is immediately clear that the arcs connecting N_F and N_S

[1]Note that F is not necessarily safe.

will never create a new conflict with a transition $t_f \in T_F$ and will also never deliver a token to any place $p_f \in P_F$. ■

The following section gives applications for this result.

8.3 Expressiveness of the functional model

From the previous theorem two corollaries directly follow which characterize the power and validity of the analysis of the untimed functional model with respect to the behavior of the combined model with scheduling and resource access.

The first corollary states that behavior can be changed by adding a scheduler net to the functional net:

Cor. 8.3.1: *Let N_F and N_S be two Petri nets which form the T-partition and the P-partition, respectively, of a TPPT-partitioned Petri net N_C. Then:*

$$\exists N_F, N_S : R_{F,C} \subset R_F \tag{8.2}$$

This means that there are combinations of T- and P-partitions N_F and N_S for which the set of reachable markings of N_F is actually restricted by the behavior of N_S. For the system specification problems this means that for a given scheduler net it may be possible to find a scheduler net which prevents the occurrence of certain undesirable behavior found during the analysis of the functional net. This corollary can easily be proven by giving an example where $R_{F,C} \subseteq R_F \bigwedge R_{F,C} \neq R_F$.

The second corollary characterizes system behavior that is invariant to the addition of a scheduler net to the functional net:

Cor. 8.3.2: *Let m be a partial marking for the places of the T-partition N_F of a TPPT-partitioning of N_C ($m : P_F \longmapsto \mathbf{N}$). Let $m_{0,C}$ and $m_{0,F}$ be the initial markings of N_C and its projection on N_F ($m_{o,F} : P_F \longmapsto \mathbf{N}$ and $\forall p \in P_F : m_{0,F}(p) = m_{0,C}(p)$), respectively. Then:*

$$\left(\nexists \sigma_F : (m_{0,F} \xrightarrow{\sigma_F} m)\right) \implies \left(\nexists \sigma_C : (m_{0,C} \xrightarrow{\sigma_C} m)\right), \forall m \tag{8.3}$$

This means that if N_F is observed on its own and a certain marking m is not reachable from the initial marking $m_{0,F}$, then no P-partition N_S can be constructed such that any of the resulting reachable markings of the combined net N_C (with the same initial marking for the places of N_F as before) contains m. With respect to verification problems this means that a behavior (system state) that does not occur in the functional net when analyzed independently can not occur in the combined net either. The proof follows directly from Theorem 8.2.1.

In this chapter the validity of analytical results with respect to the complete system has been investigated. For verification of functionality the central property of the untimed functional abstraction has been stated and two corollaries have been derived which characterize the types of generally valid analytical statements about the system that can be made from the untimed functional abstraction.

9

Application example: Mine drainage system

9.1 System specification

In this chapter a Petri net model for a part of a mine drainage system will be constructed and analyzed. This system is an example which is commonly used in the literature to illustrate real-time system modeling methods, e.g. in [108, 27, 122, 30, 88]. The example will be used to demonstrate both pattern based model construction and evaluation of functional and temporal properties using the methodology presented in Chapter 7, the schedulability test of Chapter 3, and statements about the validity of evaluation results presented in Chapter 8.

The purpose of the mine drainage system is to pump out the water that collects at the sole of a mine and to monitor several environmental parameters in the mine, such as water, CO2, and CH4 levels, as well air and water flow. The pump is activated as soon as the water level reaches a certain mark. In order to prevent an explosion, whenever there is a critical concentration of CH4 the pump motor must not be switched on or must be shut down within 1 second, no matter what the water level is. Detailed specifications for the mine example application can be found in [30].

In the chosen implementation, the application has seven tasks ('airflow monitor', 'CO2 monitor', 'waterflow monitor', 'CH4 monitor', 'interrupt handler', 'pump motor on', and 'pump motor off'). The system environment consists of several sensors, the pump, the model of the mine sole where the water accumulates, an operator console for manual switching of the pump, and a display/log to which the results of all checks as well as any alarms are written (see Fig. 40).

'CH4 monitor' and 'motor on' communicate via a shared memory location containing the the CH4Status flag (object `CH4status` in Fig. 40 and places `CH4_OK/CH4_critical` in Fig. 43, 44, 45, 51 and 52), and both motor tasks have to coordinate their access to the motor itself (object `Motor` in Fig. 40 and places `motorRunning/motorStopped` in Fig. 44, 45 and 52). Tab. 3 lists the task parameters related to timing and resource access. The specification dictates two end-to-end deadlines: First, the sequence 'interrupt handler' - 'motor on' must execute within 20 seconds, and second, the sequence 'CH4 monitor' - 'interrupt handler' - 'motor off' must execute within 1 second. The pump motor may not be switched on while the CH4 level is critical. The outputs of the CO2, airflow, and waterflow monitors must be displayed within 1, 2, and 3 seconds respectively.

Task	pri	c_{wc}	p	d_r	resources
Interrupt handler	2	0.11	100	*	
CH4 monitor seg. 1	3	0.15	5	*	
CH4 monitor seg. 2	3	0.01	5	*	
CH4 monitor seg. 3	3	0.09	5	*	CH4Status
Motor off seg. 1	4	0.1	-	*	
Motor off seg. 2	4	0.06	-	*	motor
CO2 monitor	6	0.15	60	1	
Airflow monitor	7	0.15	60	2	
Waterflow monitor	8	0.15	60	3	
Motor on seg. 1	9	0.1	-	*	CH4Status
Motor on seg. 2	9	0.16	-	*	motor

Tab. 3: Mine drainage system task parameters. The interrupt handler is a sporadic task, so the value given as period is the minimum inter-arrival time. For the deadline entries with a '*' refer to the text.

9.2 Constructing the model

The model is constructed in three steps. First a model of the task structure, task dependencies, and environment is built. This task diagram, which uses UML notation [21], is shown in Fig. 40. In the second step, this task diagram in the notation of a UML object diagram is transformed into a pattern diagram (see Sec. 7.1.1 and Fig. 41). Fig. 42 shows the resulting diagram for the 'CH4 monitor', 'interrupt handler', 'motor on' and 'motor off' tasks. The remaining monitoring tasks are very similar to the CH4 monitoring task. For simplicity they, as well as the two external interrupt sources, have been omitted in the explanation of the following steps. In the next step the patterns and environment collaborations are replaced by the actual corresponding Petri net models (see Fig. 43 and 44) using the untimed, functional variant for the Petri nets that refine the task segment patterns.

The task diagrams of Fig. 40, Fig. 42, Fig. 43, and Fig. 44 are structured and laid out following the *generic real-time system* pattern, the exchangeable scheduler subnet is represented by the dark gray bar at the right hand side of the figures. The part of the system presented in Fig. 42 interacts with two I/O devices which form the system environment: A CH4 sensor, an operator display unit which logs actions taken as well as any alarms, and the sump, the bottom of the mine where the water collects. The CH4 monitoring task, periodically activated by a *periodic requester* consists of three *task segments*: the first reads the sensor value, the second does some computation to process it and the third writes the result (CH4 level OK/not OK) into a variable that is shared with the 'motor on' task. The results are also written to the operator display. In case of a dangerously high CH4 level, the motor interrupt handler is executed. This task has the job of coordinating the different sources of motor switching requests of which only the CH4 sensor is shown here. The interrupt handler then activates the task which switches the motor off if it is running. This task consists of two segments, the first of which does some computation and the second of which accesses the output device for the motor and writes to the operator log. Access to the motor is also shared with the 'motor on' task. All tasks are connected to a scheduler submodel via the interface explained in Section 5.3.

Fig. 43 and 44 show the Petri net model obtained after replacing the collaborations of Fig. 42 with the corresponding nets, applying patterns where appropriate. As can be seen, the task segment pattern implementations overlap to avoid excess elements that do not contribute to the model function. Between the `accessRes` and `end` transitions of the task segment that modifies the shared CH4 status variable according to the measured value the net structure generated by the pattern

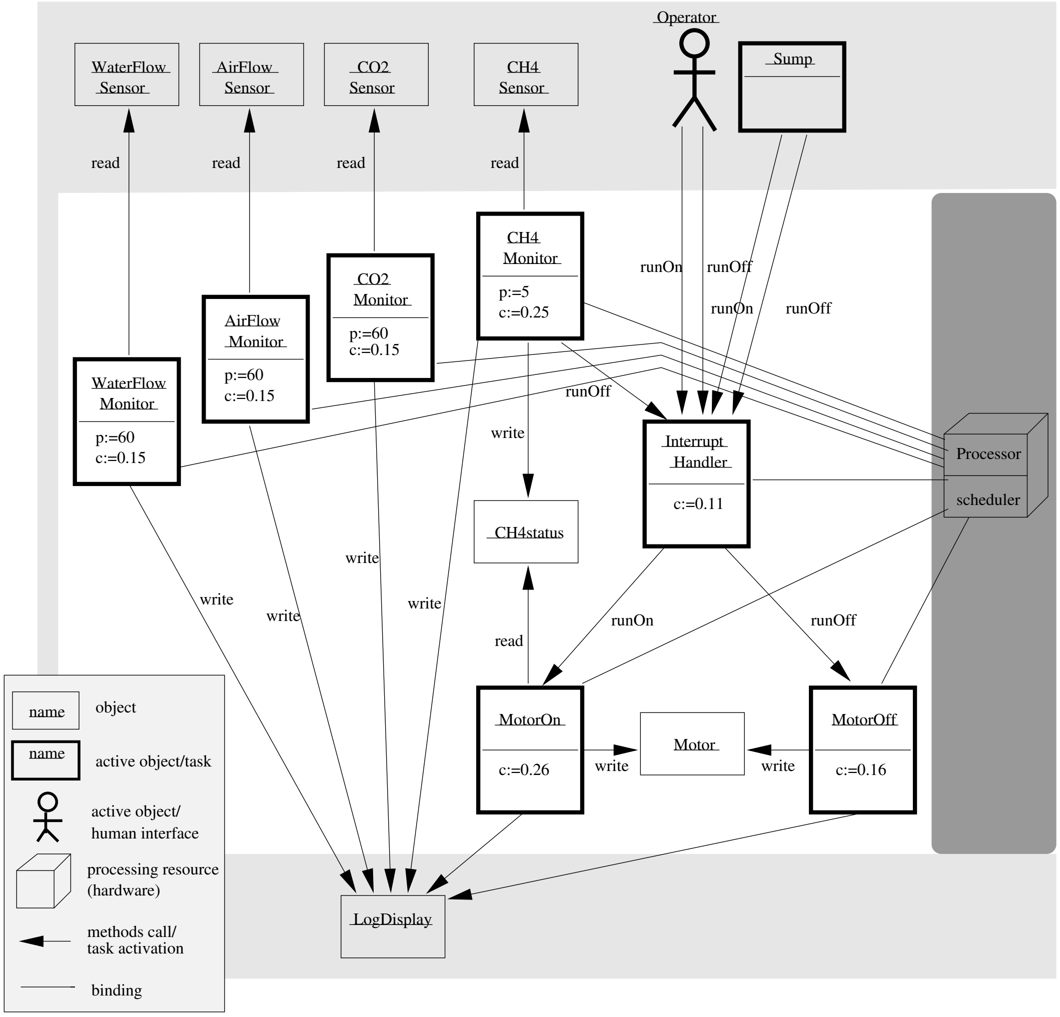

Fig. 40: Task structure of the mine drainage system.

has been extended with a detailed subnet which models the modification of the shared status variable and the functionality that the motor interrupt handler is only activated if the measured value is above the limit but was still below the limit at the previous measurement.

As an example for the use of additional patterns for the implementation of application-specific model parts two instances of the *alternative* idiom are indicated in Fig. 43.

Annotating the resulting net with the patterns from which it is composed also aids model comprehension by somebody who did not construct it, an aspect that is of major importance if the model is used as animated system specification by a larger design team. In this work the UML style of marking patterns (oval for pattern label connected with the participant elements - here only with the transitions, to avoid visual clutter) is used. In a computer-based Petri net tool highlighting of patterns using different colors would be even more effective.

9.3 Verification of functional behavior

The functional behavior to be investigated as an example for this step is, whether it can happen that the tasks execute in such a way that the motor is not switched

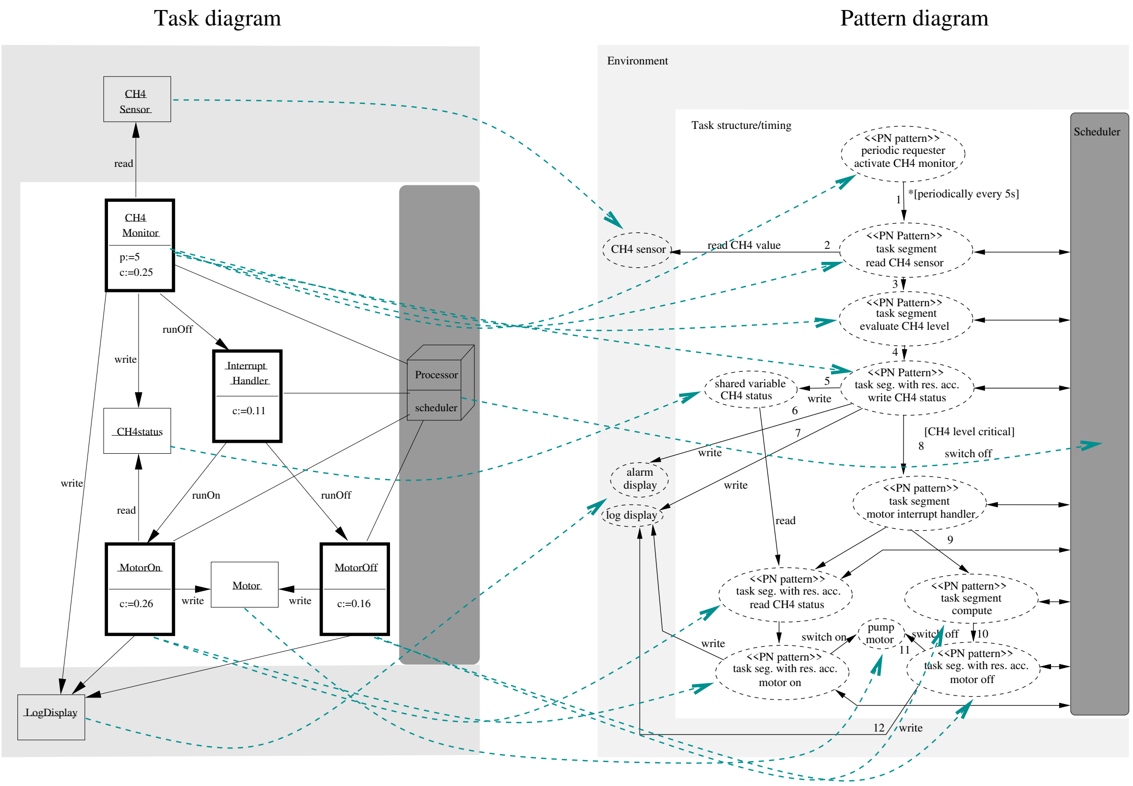

Fig. 41: Correspondence between task diagram and pattern diagram elements for the mine example.

off despite a critical CH4 level. This query can be formally expressed in CTL [43] as:

$$\neg EF \quad (CH4\_critical \ \wedge \ motorRunning \wedge \\ \wedge \ motorOffCompleted \ \wedge \ motorOnCompleted)$$

where the identifiers refer to tokens in the corresponding places in the reduced net of Fig. 45. The reduced net is generated from the net representing the 'motor on', 'motor off', and 'interrupt handler' tasks of Fig. 44, by applying the 'parallel redundant place' and 'serial fusion' reduction rules of [148] and by pruning subnets that can not influence (by token injection, token removal, or blocking) the behavior of the subnet we are interested in. The behavior of various interrupt sources and the environment subnet in Fig. 44 can for the analysis question considered here be reduced for the CH4 monitoring task to the combination `p11, t2` that changes the CH4 status flag and sends a 'motor off' command to the interrupt handler and an initial marking in `p1` representing all possible sources for a 'motor on' command. The faulty behavior may arise when a 'motor on' command from either the water level sensor or the operator console coincides with an emergency shutdown sequence. From Fig. 40 it can be seen that there is, except for the non-blocking writing to the log, no structural connection between the waterflow, airflow, and CO2 monitoring tasks and the tasks involved in the emergency shutdown. As there is no functional interaction, these tasks can not interfere with the emergency shutdown procedure in the untimed system and are thus disregarded.

The net of the example in this chapter is small enough to be reduced manually, but there are tools[1] available that perform automated net reduction, generate the state space, and answer CTL queries (model checking) (e.g. PEP [68], INA[2], and SMV[3]).

[1] A comprehensive survey of Petri net tools can be found at http://www.daimi.aau.dk/PetriNets/tools

[2] http://www.informatik.hu-berlin.de/lehrstuehle/automaten/ina/

[3] http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/

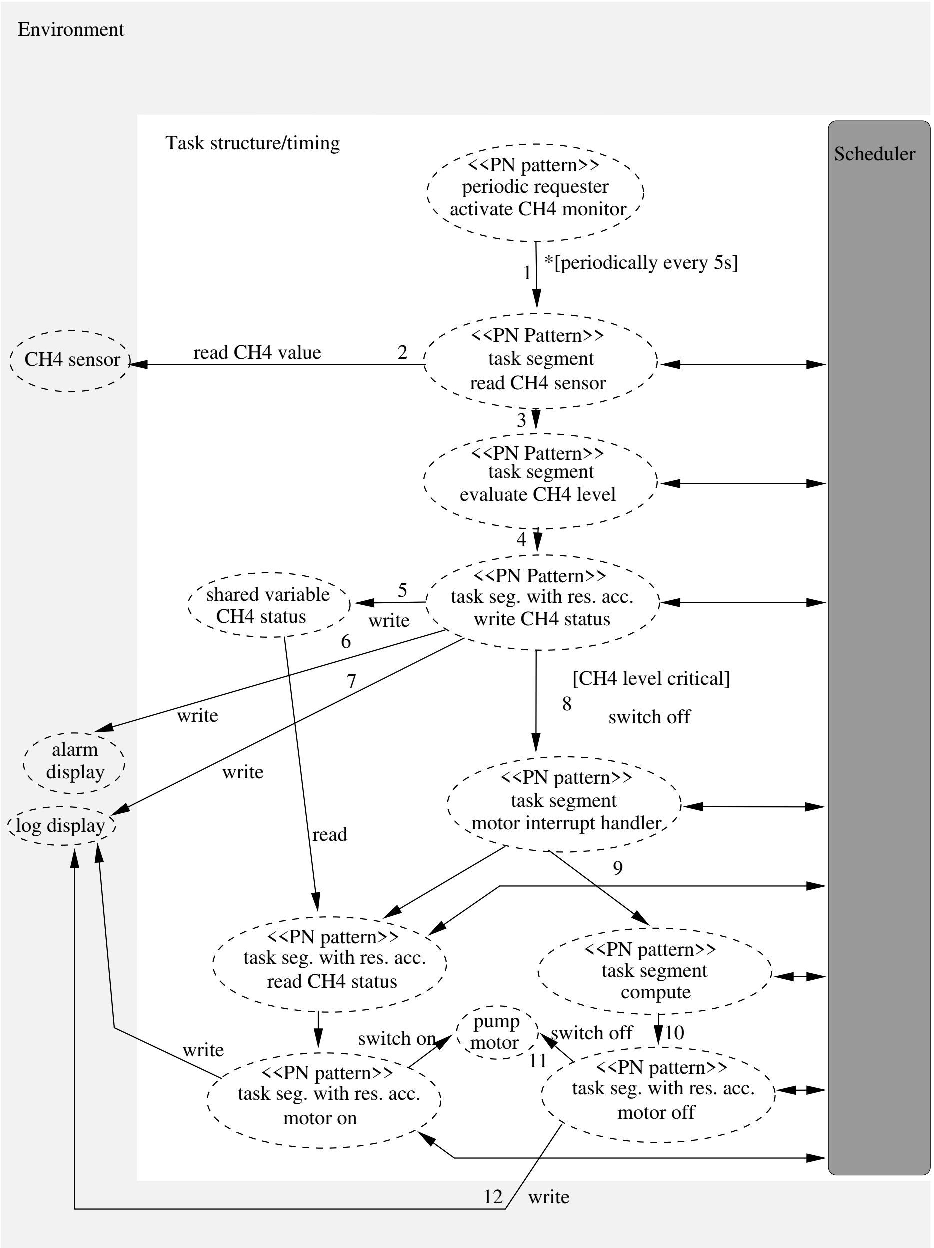

Fig. 42: Pattern diagram of the mine drainage system. The numbers denote the sequence of execution for the emergency shutdown discussed in Sec. 9.5.

The two possible outcomes of the analysis are that either the faulty behavior does not occur in the functional net, then no further action is necessary, because it can not occur in the timed model either due to Corollary 8.3.2, or that it does occur, then Corollary 8.3.1 states that there may exist a scheduler net which can prevent the undesired behavior.

Fig. 46 shows the reachability graph generated from the reduced model[4]. As can be seen, there exists a final state representing the faulty behavior. It occurs when between reading the old, non-critical CH4 status and actually switching the motor on the 'motor on' sequence is preempted by the CH4 monitoring task toggling the CH4 flag to critical and executing the shutdown sequence. The motor will then be shut down correctly, but after the shutdown sequence the 'motor on' task resumes and switches the motor back on.

[4]The state space has been generated with an internal tool written by Jörn W. Janneck.

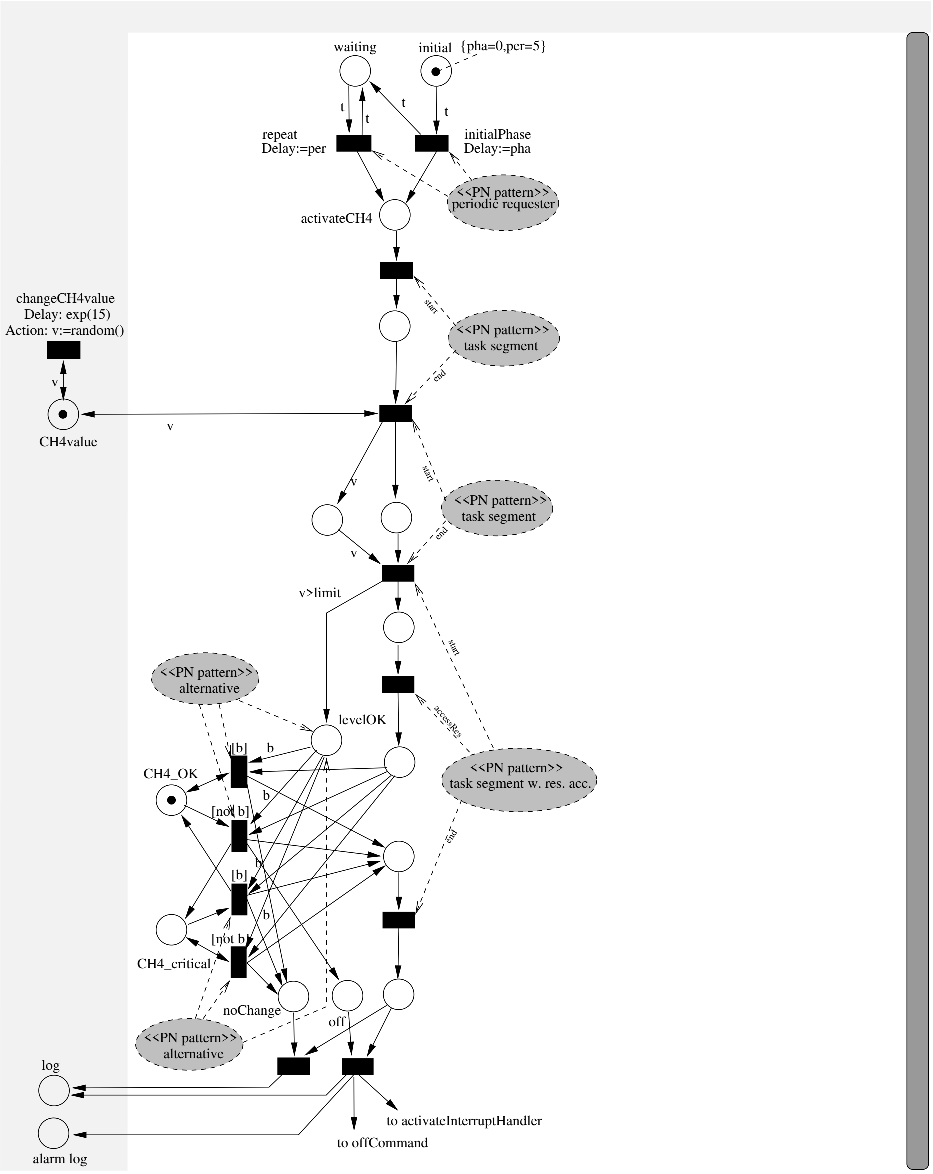

Fig. 43: Petri net model of the mine drainage system (functional model), part 1 with the 'CH4 monitor'

There are several design options for ensuring that this behavior can not occur in the system. The two solutions presented here use overlapping critical sections together with a resource access protocol. Clearly, it is necessary to prevent either the CH4 monitoring task or the 'motor off' task from executing between the flag reading and the motor switching in the 'motor on' task. This can be achieved by either (a) accessing the CH4 status in 'motor on' only while already holding a lock on the motor or by (b) accessing the motor in 'motor on' only while still holding the lock on the CH4 status. In the final timed model this will be the responsibility of an appropriate resource access protocol. To verify that these two approaches work, the functional net in Fig. 45 is augmented with the places `motorUnlocked` and `CH4statusUnlocked` for variants (a) and (b) respectively - they are P-partitions (see Section 8.1) and emulate the resource access protocol of the scheduler - and the reachability analysis is repeated. Figure 47 and 48 show the state spaces for solutions (a) and (b). The faulty behavior does not occur any more, and, assuming that the behavior of the resource access protocol of the actual scheduler net observable from the functional net is like that of the net with one or the other of these inserted places, Corollary 8.3.2 ensures that the fault will not occur in the system.

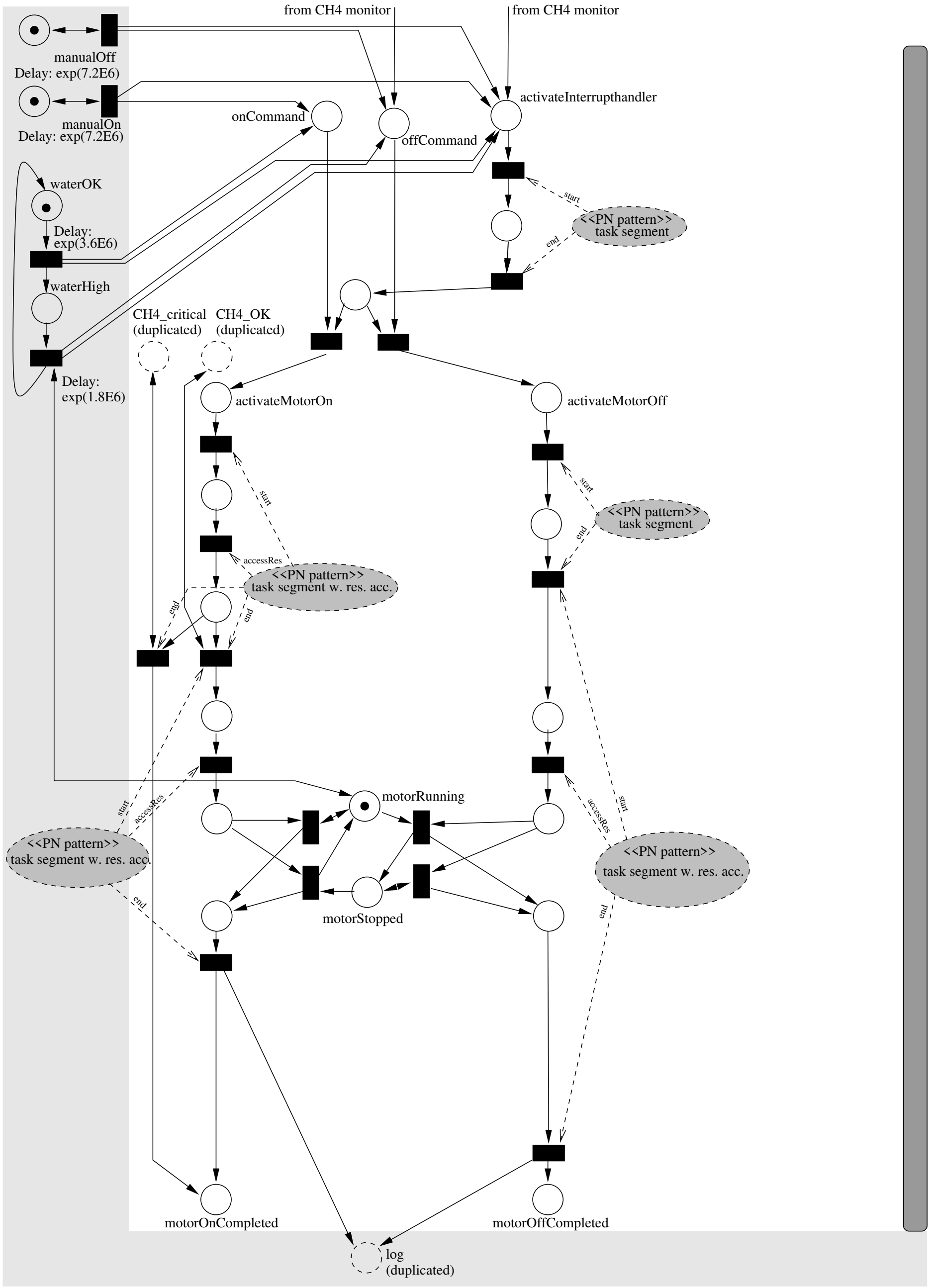

Fig. 44: Petri net model of the mine drainage system (functional model), part 2 with 'interrupt handler', 'motor on', 'motor off', and sump.

This section has shown how functional analysis together with Corollary 8.3.1 can be used to derive requirements for the timing/scheduling of the system. Another example of an inquiry using directly Corollary 8.3.2 would be: 'Will it be possible to switch the motor on again after an emergency shutdown?' In this case, if the analysis gave a negative result it would be clear that the functional specification is wrong and no scheduler/resource access protocol can correct it.

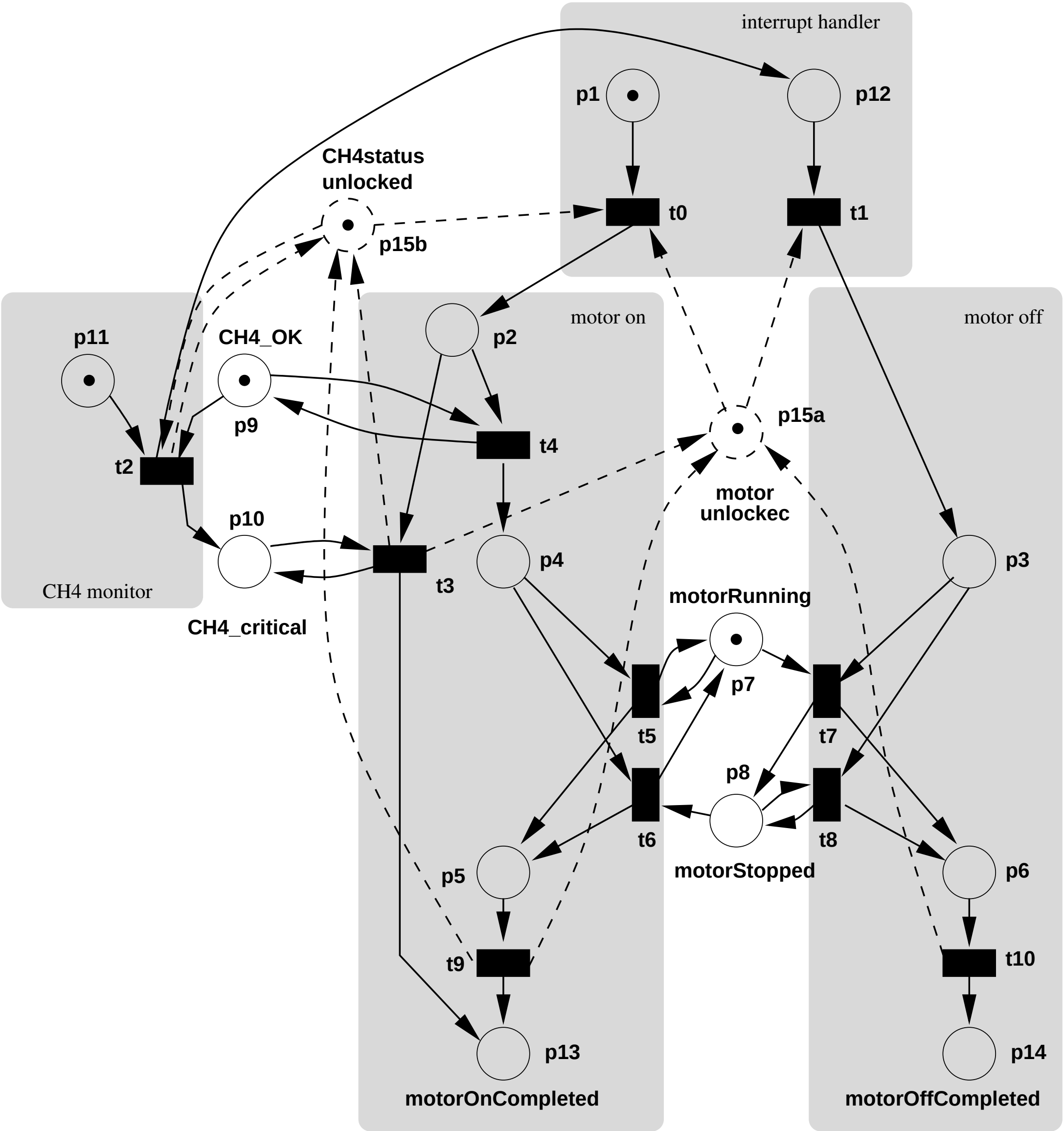

Fig. 45: Reduced net model of the tasks influencing emergency shutdown. The dashed elements are added to represent shared resource locking to avoid the discovered faulty behavior.

9.4 Timing information

The schedulability test of Chapter 3 has been applied to the tasks of the mine system with the timing parameters of Table 3. As examples, two results are reported here:

The emergency shutdown sequence tasks, 'CH4 monitor', 'interrupt handler', and 'motor off', have the highest priorities in the system. Fig. 49 shows that the execution of the shutdown sequence does not cause a deadline miss for the next lower priority task, 'CO2 monitor'.

For the evaluation shown in Fig. 50 it is assumed, contrary to Table 3, that all tasks not belonging to the shutdown sequence have a higher priority than the shutdown sequence. As can be seen, in this case the deadline for the shutdown sequence would be missed.

For the following we assume the timing parameter values as listed in Table 3. With these values the schedulability tests indicate a schedulable system for fixed-priority preemptive scheduling.

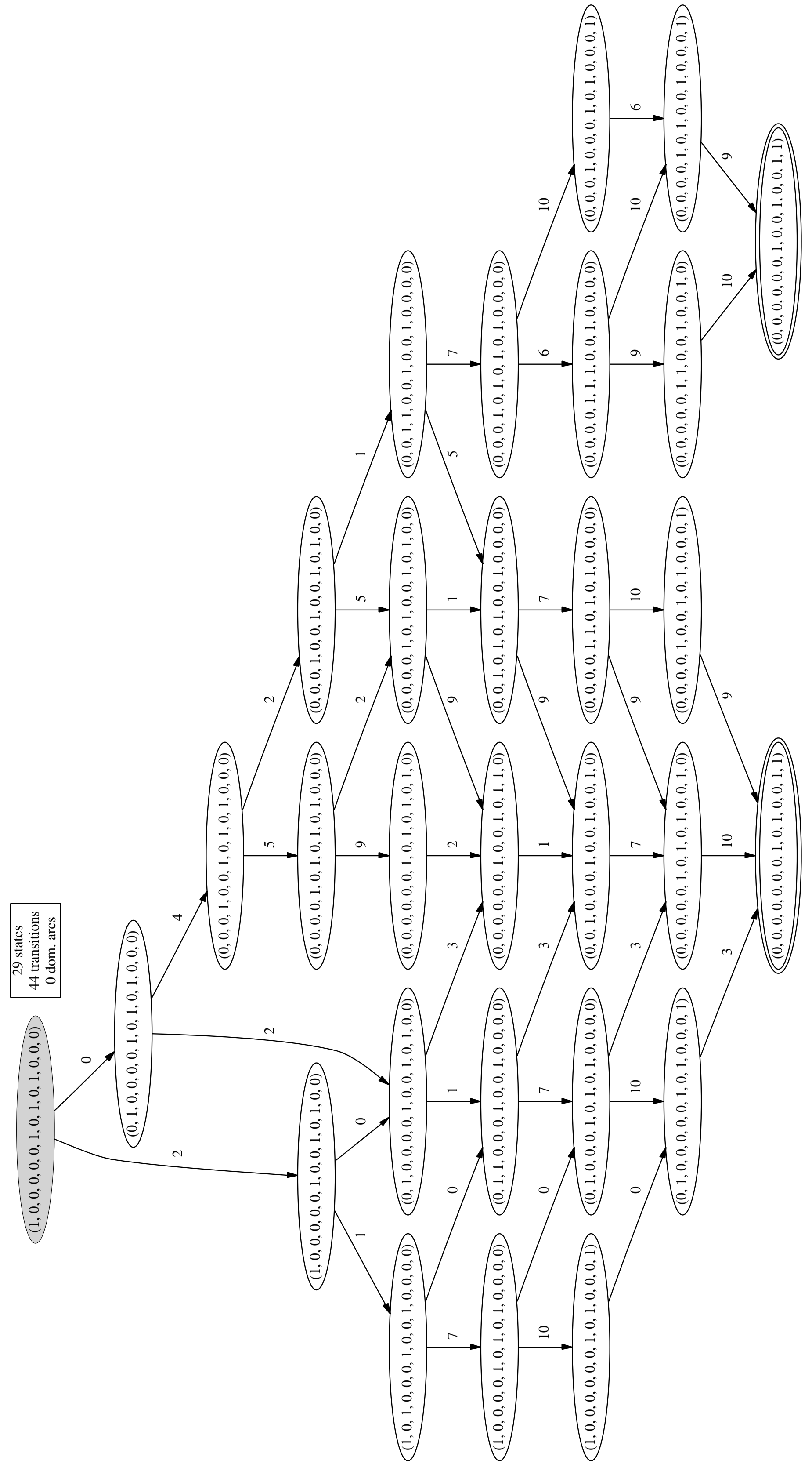

Fig. 46: Reachability graph of the net in Fig. 45 showing an undesirable final state with tokens in places p7 (`motorRunning`), p10 (`CH4_critical`) , p13 (`motorOnCompleted`), and p14 (`motorOffCompleted`) (00000010010011).

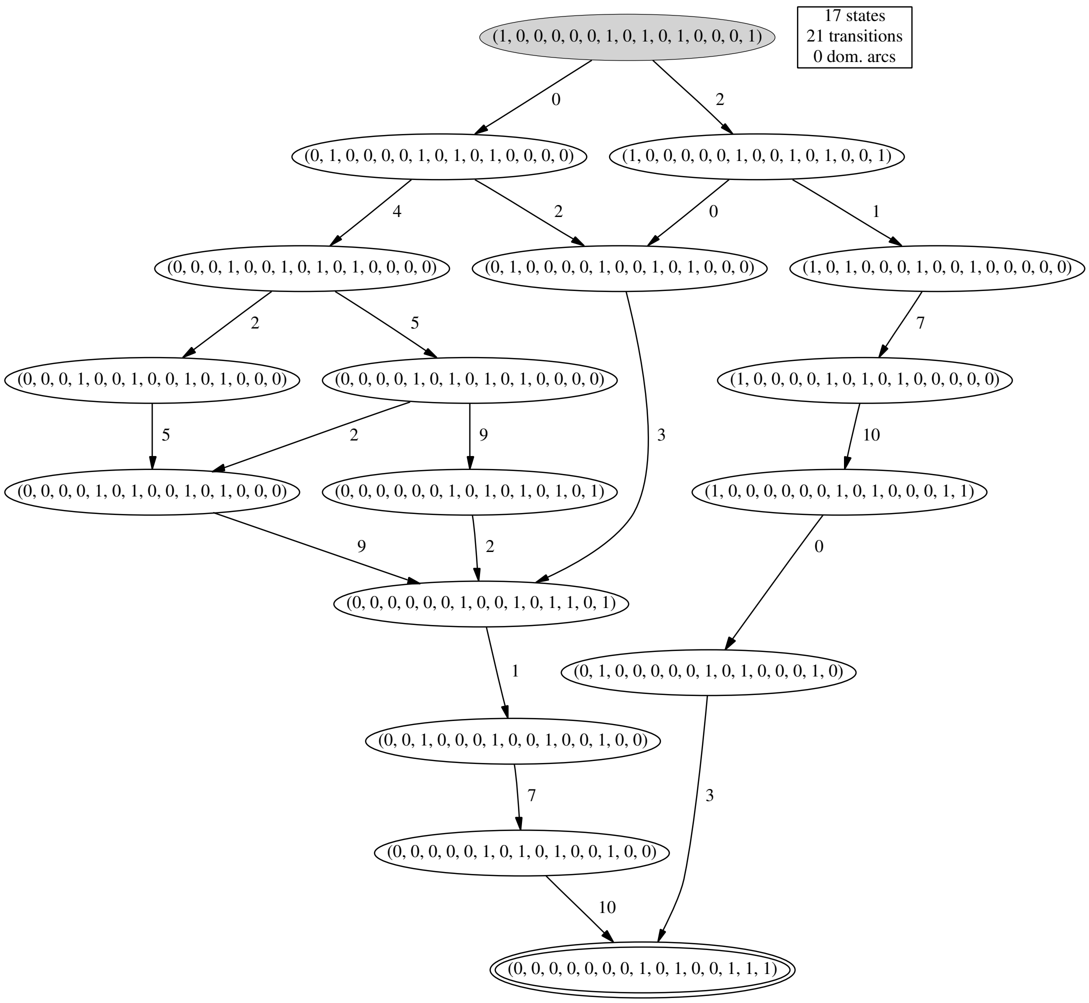

Fig. 47: State space of the net in Fig. 45 with the solution variant (a).

Figures 51 and 52 show a part of the final system model with timing information after applying the transformations of Section 7.1.3. The model uses locking variant (b) described above, thus a task segment has been added at the end of the 'motor on' task to free both locked resources, the CH4status variable and the motor. In order to be able to visually separate the functional net and the timing adapter net, the elements of the latter are drawn in gray. In the figure all inscriptions in the task timing subnet except for the task parameters have been dropped to decrease visual clutter.

9.5 Evaluation of timing behavior

As an example for a property to be analyzed in the scheduling simulation it will be investigated whether there are situations for which the 1 second deadline of the emergency shutdown deadline can not be met by a certain scheduling strategy.

The system was simulated using non-preemptive First Come First Served (FCFS), Earliest Deadline First (EDF) with non-preemptable sections, Fixed Priority (FP) with non-preemptable sections, and Fixed Priority with Priority Ceiling Protocol (PCP) [116]. Only the results for non-preemptive FCFS and FP with PCP will be presented here.

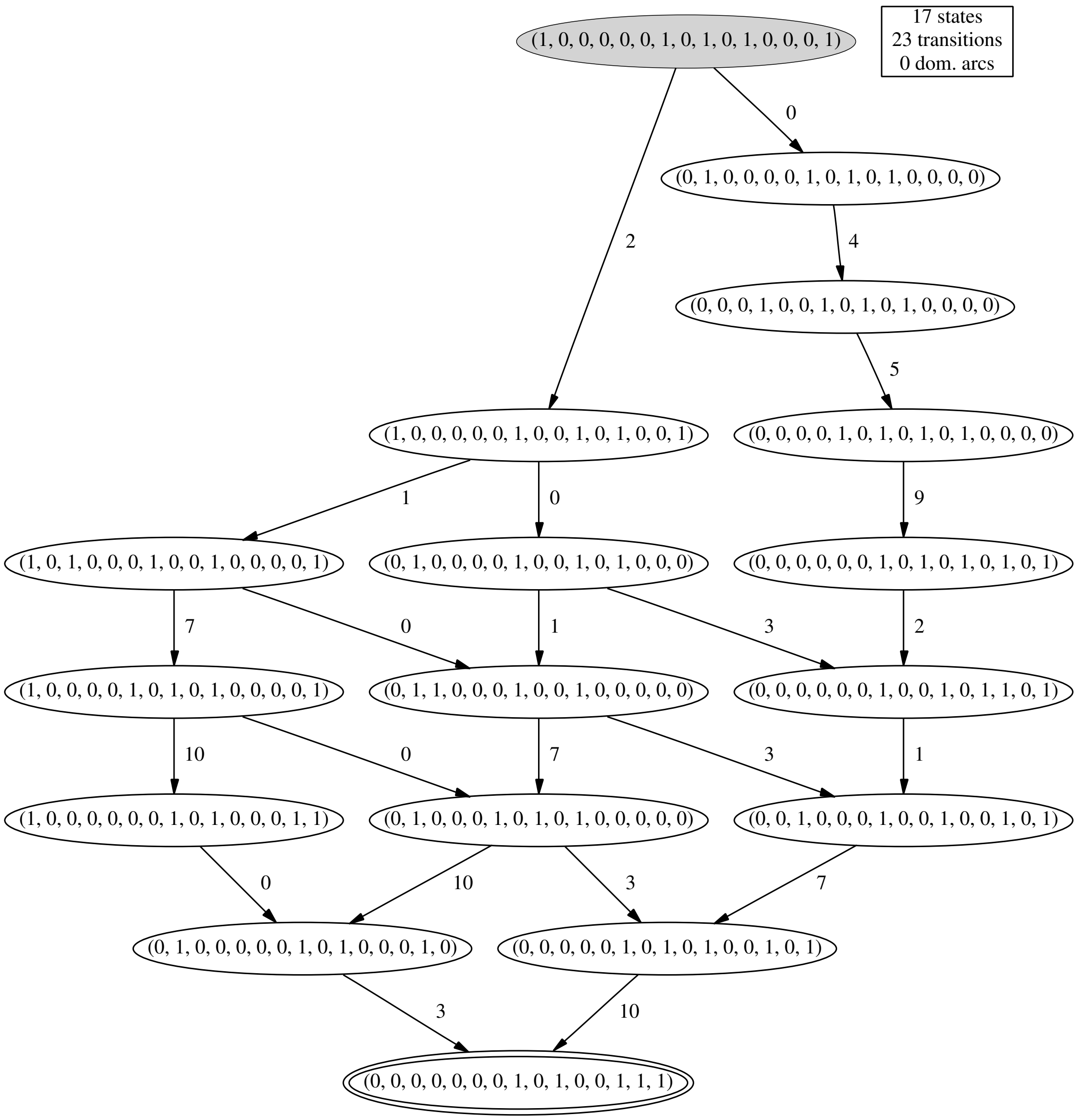

Fig. 48: Reachability graph of the net in Fig. 45 with the solution variant (b).

Figures 54 and 53 show the resulting Gantt charts for the critical scheduling situations for simulations with preemptive fixed-priority scheduling with priority inheritance and non-preemptive FCFS scheduling on a single processor.

The most critical situation arises whenever the 'CH4 monitor' executes and initiates an emergency shutdown of the motor while a 'motor on' sequence is also executing. For the fixed-priority PCP scheduler this worst case occurs when the 'CH4 monitor' task is activated shortly after 'motor on' has started execution: The 'motor on' task which has already locked the CH4 status variable is preempted by the 'CH4 monitor' task, the last segment of which blocks on the CH4 status until 'motor on' has finished and freed the resource. Then the 'CH4 monitor' completes and activates the interrupt handler which in turn starts 'motor off'. The 'CH4 monitor' task arrives at t=0.56 s and the shutdown sequence completes at t=1.34 s, thus the deadline at 1.56 s is met.

With non-preemptive FCFS scheduling, an even worse constellation with respect to the shutdown deadline can be constructed when the interrupt handler for the 'motor on' sequence is released slightly before the three simple monitor tasks, and the the 'CH4 monitor' just after the interrupt handler has requested the 'motor on' task.

As can be seen from the figures, the end-to-end deadline for the shutdown se-

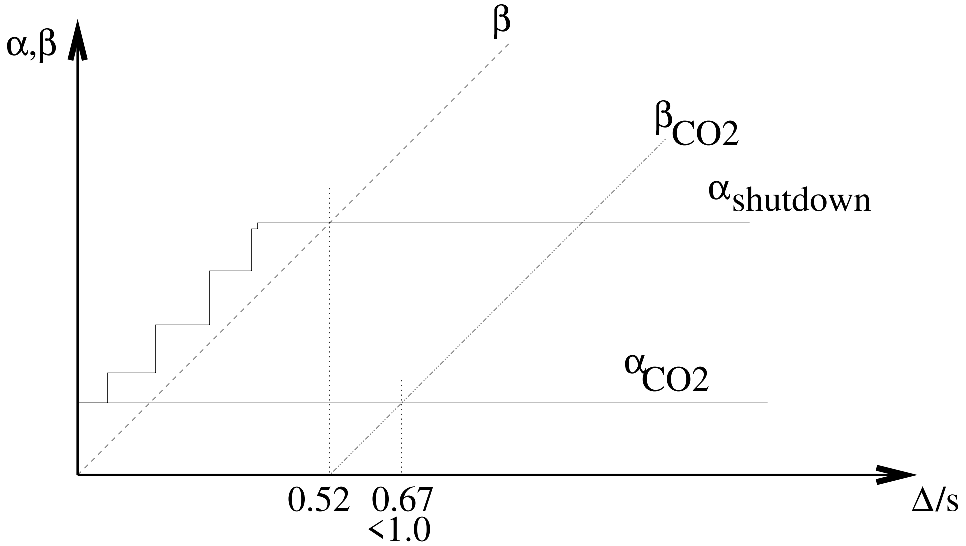

Fig. 49: Schedulability test for 'CO2 monitor'. β is the delivery curve of the processor, $\alpha_{shutdown}$ the request curve of the shutdown sequence, β_{CO2} the remaining delivery curve after scheduling the shutdown sequence, and α_{CO2} the request curve of the 'CO2 monitor' task which has a deadline of 1 second. The deadline will be met.

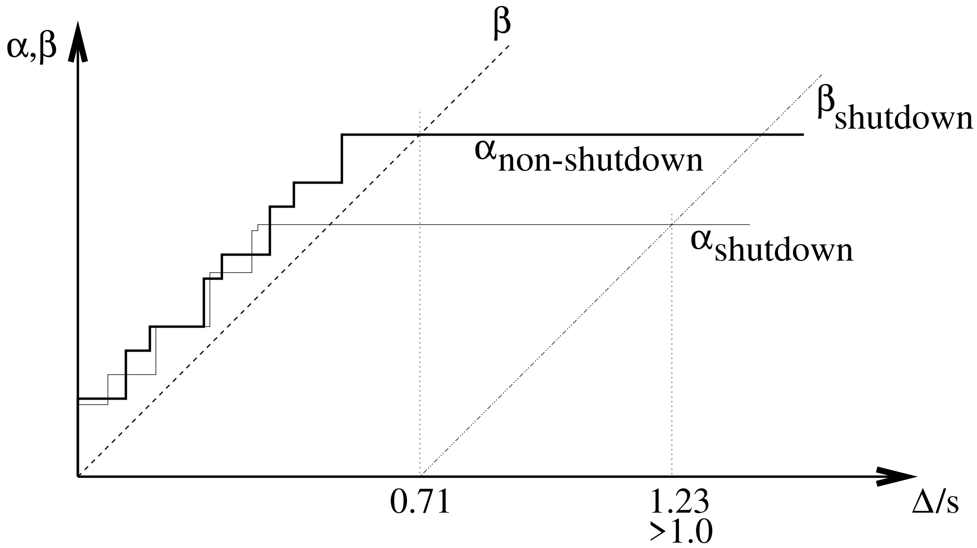

Fig. 50: Schedulability test with low priority for the shutdown sequence. With the remaining delivery curve $\beta_{shutdown}$ the shutdown sequence may finish in the worst-case situation at 1.23 seconds and thus miss the deadline of 1.0 seconds.

quence is met for the fixed-priority with PCP scheduler, because the maximum blocking time the shutdown sequence can encounter is the length of 'motor on'. However, for non-preemptive FCFS scheduling the three non-critical monitoring tasks are also executed on the critical path of the shutdown sequence and thus the time span from the release of the 'CH4 monitor' ($0.11 + \epsilon$ s) to the end of 'motor off' (1.34 s) is 1.23 seconds and the deadline at 1.11 s is missed. This corresponds to the second analysis result presented in the previous section, as the use of FCFS scheduling together with the task invocation sequence of this example is equivalent to all tasks not belonging to the shutdown sequence having higher priorities than the shutdown sequence. This example shows that the simplified assumption of non-preemptive and random scheduling often made in the Petri net modeling of real-time systems [122] may lead to wrong results. In [30] rate-monotonic schedulability analysis is conducted which declares the mine drainage system schedulable with fixed-priority preemptive scheduling, but this work did not take into account the effects of blocking on shared resources.

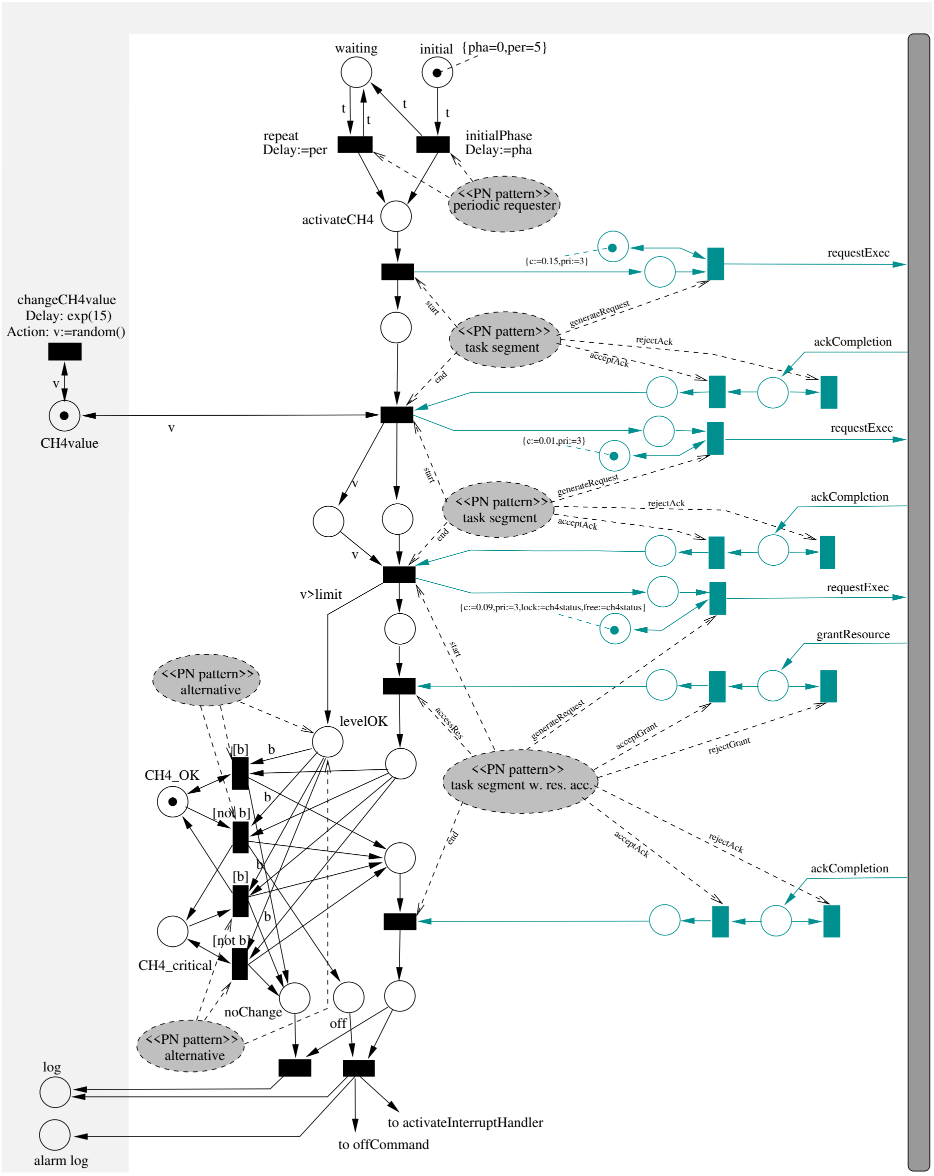

Fig. 51: Part 1 of the Petri net model for the mine drainage system with timing/scheduling subnet added: CH4 monitor task.

In this chapter the application of the methodology of Chapter 7 has been demonstrated using a well-known example system from the literature. It has been shown how to systematically derive the analyzable and simulatable Petri net model from task specification and how to obtain verification results from the model. Compared to the previous treatment, the integrated approach produced new insight into the behavior of the system.

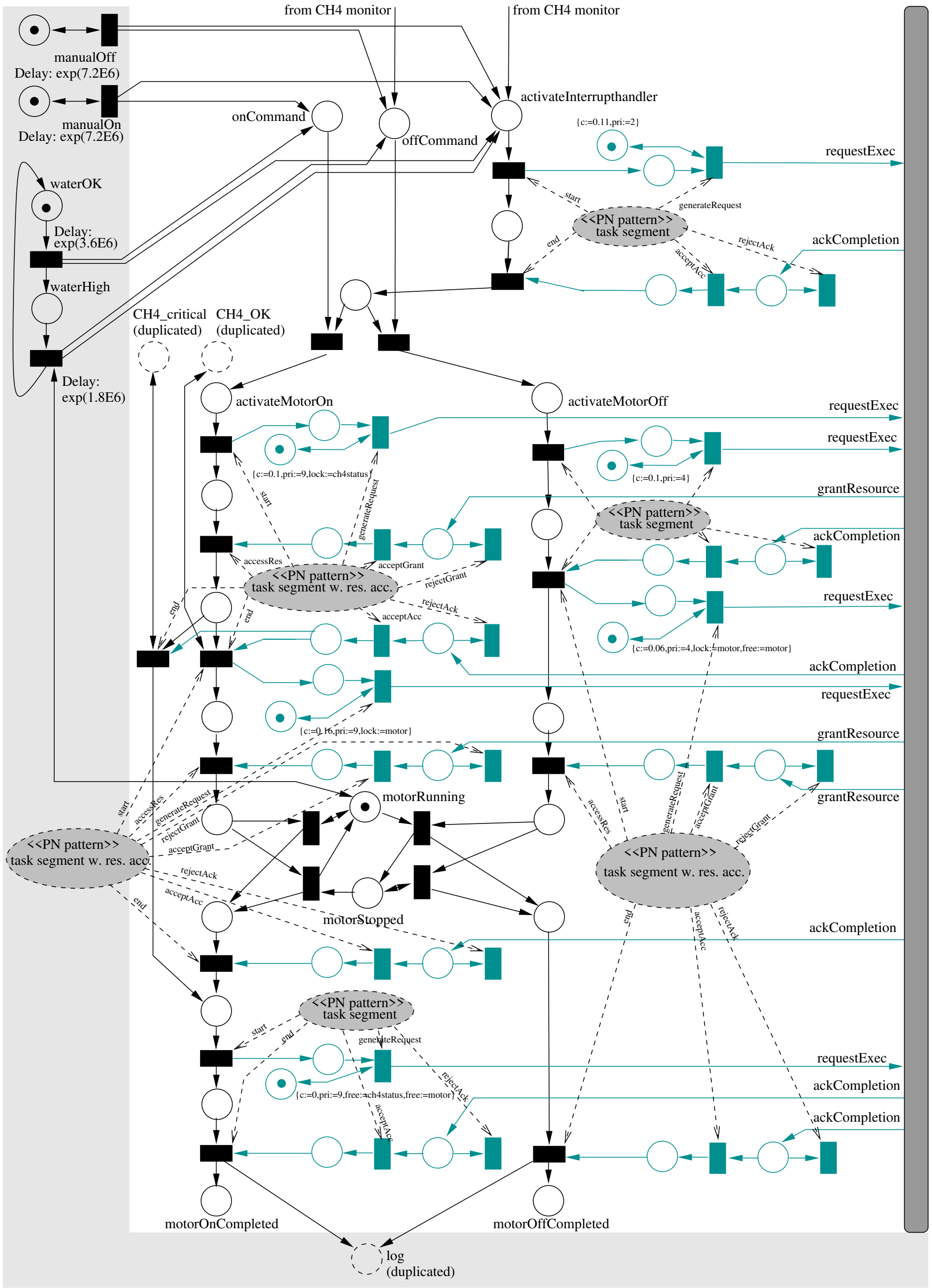

Fig. 52: Part 2 of the full Petri net model for the mine drainage system with timing/scheduling subnet added: sump, manual controls, interrupt handler, and motor on/off tasks.

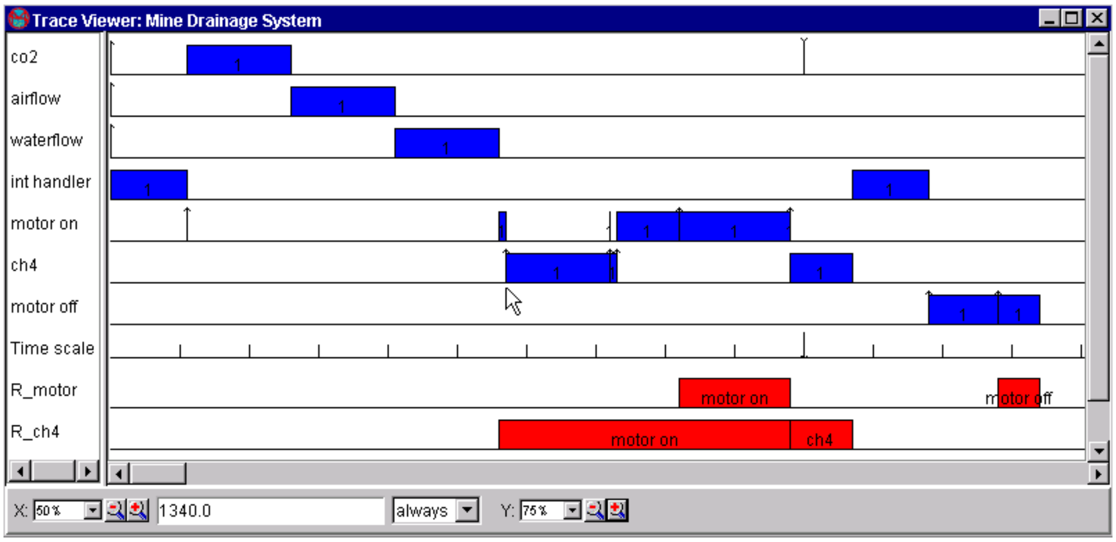

Fig. 53: Simulation result for preemptive fixed-priority scheduling with priority ceiling protocol (first 1.4s). The arrow indicates the release of the CH4 monitor task. Above the time scale the task requests, deadlines, and execution intervals are drawn, below it the locking intervals and locker tasks of the resources are visualized.

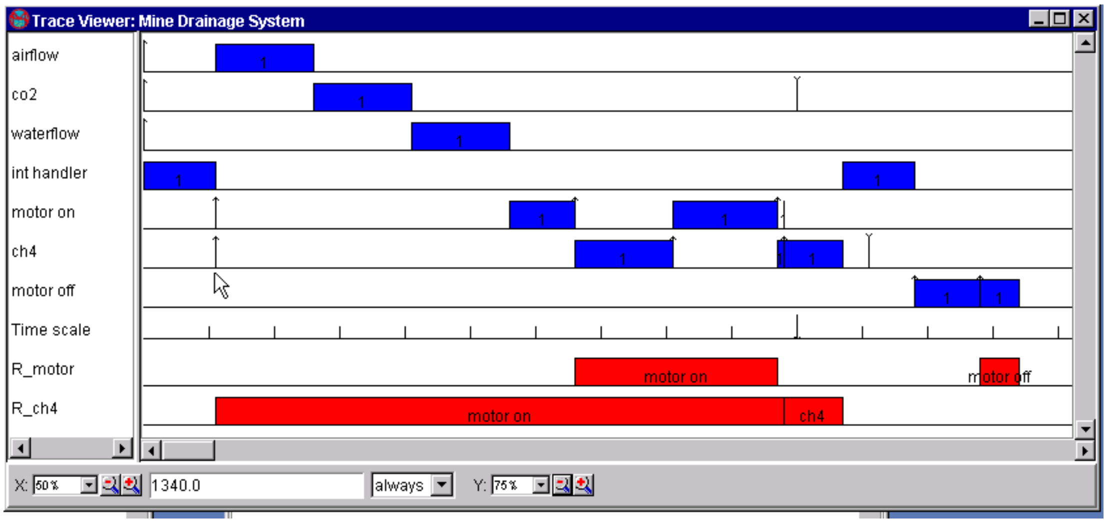

Fig. 54: Simulation result for non-preemptive FCFS scheduling (first 1.4s).

10

Conclusions

10.1 Summary

This monograph presents an approach to the modeling and evaluation of real-time systems which employs an integrated model for the evaluation of functional behavior, timing behavior, and performance. This integrated model replaces the several different method-specific models that are conventionally used as input for the evaluation of different system aspects by analytical means and simulation. The integrated approach reduces the effort and thus cost necessary to build and maintain models and at the same time prevents problems arising from the fact that different partial models may be inconsistent.

The modeling approach presented here is based on Petri nets as underlying formalism. Petri nets have been chosen because they combine an intuitive visual representation with a clearly defined, formally analyzable execution semantics which is particularly geared towards expressing the concurrency and synchronization issues pertinent to multi-task real-time systems.

The aspect of model construction, which is often neglected in proposals for new formal specification approaches, motivated the transfer of the concept of design patterns from the software engineering domain to the domain of modeling with dataflow visual languages. A system evaluation methodology has been developed which contains a pattern-based model construction stage. Design patterns are not only an aid for model construction but also for model documentation. Models created using dataflow visual language such as Petri nets tend to be of high visual complexity because the elements which constitute the model are on a low level of abstraction. Annotating these elements with their generating patterns reduces the number of items that need to be mentally processed which in turn facilitates model comprehension.

Both the model construction and evaluation aspects, including formal reachability analysis and simulation, have been demonstrated in the case study of a mine drainage system. This case study had previously been used in other works to present real-time system modeling and schedulability analysis methods. The approach presented this work obtained new results with respect to both functional behavior and schedulability of the example system.

10.2 Contributions

The main contribution of this work is that it presents the first Petri net based approach to the evaluation of real-time systems using an integrated model as postulated in [83].

In order to achieve this goal, several issues have been resolved in the areas of model construction with dataflow visual languages, real-time system modeling with Petri nets, and evaluation of system models:

10.2.1 Design patterns

- To support the effective and efficient construction of system models the idea of using design patterns has been adapted for the Petri net formalism, which is a representative of the family of dataflow visual languages.
- A set of general Petri net design patterns and idioms has been identified, their dependencies and relations have been mapped out, and their occurrence in and relevance for published Petri net models has been investigated.

10.2.2 Real-time system modeling with Petri nets

- This work solves for the first time the problem of modeling general preemptive scheduling in a scalable way with Petri nets, using a novel modeling style, delegated execution.
- A library of Petri net components for standard scheduling and resource access algorithms from the literature has been built.
- Application specific Petri net design patterns for the real-time system domain have been identified and documented.

10.2.3 Model construction and evaluation

- A methodology for model construction and evaluation based on a generic real-time system pattern implementing delegated execution has been proposed. The resulting model consists of a formally analyzable functional submodel and a complementary timing submodel which leads to a combined model amenable to timing and scheduling evaluation through simulation.
- The concept of a TPPT-partitioned Petri net, the formal foundation of the delegated execution modeling style, has been developed and used to prove characteristic properties with respect to the analyzability of the generic system model.
- For schedulability analysis a new representation of bursty task arrivals and bursty processor capacity has been developed which integrates previous task and processor characterizations and places them into a common mathematical framework.

10.3 Future work

This work constitutes one step towards improving the usability of formal models and specification in the development process of embedded real-time systems. Further research will be needed to identify and categorize new design patterns for Petri nets and other dataflow visual languages, to find optimal pattern annotation techniques, and to provide tool support for the methodology.

A

Mathematical definitions and properties used in Chapter 3

A.1 Min-plus convolution

Def. A.1.1: **(Min-plus convolution)** *For two functions γ_1 and γ_2 define $\gamma_1 \underline{\oplus} \gamma_2$ by*

$$\gamma_1 \underline{\oplus} \gamma_2(t) = \min_{u: 0 \leq u \leq t} \{\gamma_1(u) + \gamma_2(t-u)\} \tag{A.1}$$

Min-plus convolution has the following properties [96]:

- $\underline{\oplus}$ is associative and commutative
- If $\gamma_1(0) = \gamma_2(0) = 0$ then $\gamma_1 \underline{\oplus} \gamma_2 \leq min(\gamma_1, \gamma_2)$.
- If γ_1 and γ_2 are concave then $\gamma_1 \underline{\oplus} \gamma_2 = min(\gamma_1, \gamma_2)$.
- In general, the curve of $\gamma_1 \underline{\oplus} \gamma_2$ can be obtained by moving the origin of one of the functions along the curve of the other. The bottom edge of the resulting figure is the curve resulting from the min-plus convolution.
- If γ_1 and γ_2 are convex and piecewise linear then $\gamma_1 \underline{\oplus} \gamma_2$ is obtained by putting the different pieces of both functions end-to-end, sorted by increasing slopes. The piece with the steepest slope is dropped. For an example see Fig. 55.

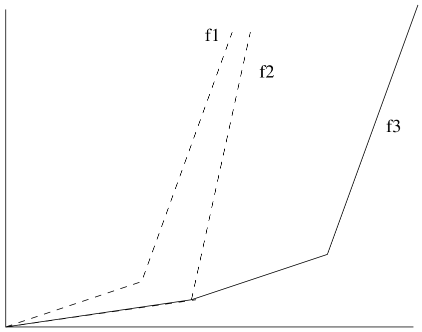

Fig. 55: Min-plus convolution for convex, piecewise linear functions

A.2 Min-minus operator

Def. A.2.1: **(Min-minus operator)** *For two functions* γ_1 *and* γ_2 *define* $\gamma_1 \underline{\ominus} \gamma_2$ *by*

$$\gamma_1 \underline{\ominus} \gamma_2(t) = \min_{u \geq 0}\{\gamma_1(t+u) - \gamma_2(u)\} \tag{A.2}$$

A.3 Max-plus convolution

Def. A.3.1: **(Max-plus convolution)** *For two functions* γ_1 *and* γ_2 *define* $\gamma_1 \overline{\oplus} \gamma_2$ *by*

$$\gamma_1 \overline{\oplus} \gamma_2(t) = \max_{u:0 \leq u \leq t}\{\gamma_1(u) + \gamma_2(t-u)\} \tag{A.3}$$

A.4 Max-minus operator

Def. A.4.1: **(Max-minus operator)** *For two functions* γ_1 *and* γ_2 *define* $\gamma_1 \overline{\ominus} \gamma_2$ *by*

$$\gamma_1 \overline{\ominus} \gamma_2(t) = \max_{u \geq 0}\{\gamma_1(t+u) - \gamma_2(u)\} \tag{A.4}$$

The $\overline{\ominus}$ operator has the following properties ($\gamma_1 \leq \gamma_2$ has the meaning $\gamma_1(t) \leq \gamma_2(t)\ \forall t \geq 0$) [96].

- If $\alpha_1 \leq \alpha_2$ then for all β: $\alpha_1 \overline{\ominus} \beta \leq \alpha_2 \overline{\ominus} \beta$.
- If $\beta_1 \leq \beta_2$ then for all α: $\alpha \overline{\ominus} \beta_1 \geq \alpha \overline{\ominus} \beta_2$.
- $(\gamma_1 \overline{\ominus} \gamma_2) \overline{\ominus} \gamma_3 = \gamma_1 \overline{\ominus} (\gamma_2 \underline{\oplus} \gamma_3)$
- $\gamma_1 \overline{\ominus} \gamma_2 \leq \gamma_3$ is equivalent to $\gamma_1 \leq (\gamma_2 \underline{\oplus} \gamma_3)$

Remark: Intuitively, $(\gamma_1 \overline{\ominus} \gamma_2)(t)$ is the largest difference in value between points on γ_1 and γ_2 that are a distance t apart.

A.5 Subadditivity

Def. A.5.1: **(Subadditivity)** *A function* γ *is said to be subadditive if* $\gamma(s+t) \leq \gamma(s) + \gamma(t)\ \forall s, t \geq 0$.

Remark: This definition is equivalent to $\gamma \leq \gamma \underline{\oplus} \gamma$. With $\gamma(0) = 0$, subadditivity is equivalent to $\gamma = \gamma \underline{\oplus} \gamma$ [96].

Cor. A.5.2: *Any concave function* γ *defined on* $\mathbf{R}^+$ *such that* $\gamma(0) = 0$, *is sub-additive.*

The proof is given in [96] (Proposition 3).

Remark: A subadditive function γ, even with $\gamma(0) = 0$, is not necessarily concave.

Def. A.5.3: **(Subadditive closure)** *The sub-additive closure of a function* γ *is* $\overline{\gamma} = \min_{n\geq 1} \gamma^{(n)}$, *where* $\gamma^{(n)} = \gamma \underline{\oplus} \gamma \underline{\oplus} \dots \underline{\oplus} \gamma$.

Cor. A.5.4: *The subadditive closure of a function* γ, $\overline{\gamma}$, *is subadditive.*

The proof is given in [96] (Proposition 2).

Remark: If the function γ is subadditive then $\overline{\gamma} = \gamma$.

A.6 Superadditivity

Def. A.6.1: **(Superadditivity)** *A function* γ *is said to be superadditive if* $\gamma(s+t) \geq \gamma(s) + \gamma(t)\ \forall s,t \geq 0$.

Remark: A superadditive function f, even with $\gamma(0) = 0$, is not necessarily convex.

Def. A.6.2: **(Superadditive closure)** *The super-additive closure of a function* γ *is* $\overline{\gamma} = \max_{n\geq 1} \gamma^{(n)}$, *where* $\gamma^{(n)} = \gamma \overline{\oplus} \gamma \overline{\oplus} \dots \overline{\oplus} \gamma$.

Cor. A.6.3: *The superadditive closure of a function* γ, $\overline{\gamma}$, *is superadditive.*

The proof is analogous to the one for the subadditive closure.

Remark: If the function γ is superadditive then $\overline{\gamma} = \gamma$.

A.7 Calculations with subadditive and superadditive functions

Theo. A.7.1: *Be* β *any superadditive and* α *any subadditive function, then* $\gamma = \beta - \alpha$ *is also superadditive.*

Proof:

$$\gamma(a+b) = \beta(a+b) - \alpha(a+b)$$

Because of

$$\beta(a+b) \geq \beta(a) + \beta(b) \ \ \textit{and} \ \ \alpha(a+b) \leq \alpha(a) + \alpha(b)$$

it follows that

$$\gamma(a+b) \geq (\beta(a) + \beta(b)) - (\alpha(a) + \alpha(b))$$

$$= \beta(a) - \alpha(a) + \beta(b) - \alpha(b) = \gamma(a) + \gamma(b) \ \ \forall\, a, b \geq 0$$

∎

Theo. A.7.2: *Be* β *any superadditive and* α *any subadditive function, with* $\beta, \alpha \geq 0$, *and* β *nondecreasing. If* $\alpha(t) < \beta(t)\,\forall\, t$ *then* $\gamma = \beta - \alpha$ *is also superadditive and nondecreasing.*

Proof: From Theorem A.7.1 it is known that $\gamma = \beta - \alpha$ is superadditive. It remains to prove that γ is also nondecreasing.

Proof is by contradiction: Assume that there exists an interval I=[t1,t2] where γ is decreasing. As β is nondecreasing, this clearly requires that the slope of α must be steeper than the slope of β in I:

$$\alpha(t_0) - \alpha(t_1) > \beta(t_0) - \beta(t_1),\; t_0 \in I.$$

Using

$$\beta(t_0) \geq \beta(t_1) + \beta(t_0 - t_1)$$

$$\Rightarrow \alpha(t_0) - \alpha(t_1) > \beta(t_1) + \beta(t_0 - t_1) - \beta(t_1) = \beta(t_0 - t_1)$$

Using

$$\alpha(t_1) + \alpha(t_0 - t_1) \geq \alpha(t_0)$$

$$\Rightarrow \alpha(t_1) + \alpha(t_0 - t_1) - \alpha(t_1) = \alpha(t_0 - t_1) > \beta(t_0 - t_1)$$

This is a contradiction to the requirement that $\alpha(t) < \beta(t)\,\forall\, t$. ■

Bibliography

[1] S. Aggarwall and C. Chraibi. Hyperperiodic tasks: A new model for hard real-time systems. In *Proceedings IFAC 12th Triennial World Congress*, 1993.

[2] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language*. Oxford University Press, 1977.

[3] P. Ancilotti, G. Buttazzo, M. D. Natale, and M. Bizzarri. A flexible tool kit for the development of real-time applications. In *Proceedings RTAS 96*, 1996.

[4] P. Ancilotti, G. Buttazzo, M. D. Natale, and M. Spuri. A development environment for hard real-time applications. *International Journal of Software Engineering and Knowledge Engineering*, 6(3):331–354, 1996.

[5] P. Ancilotti, G. Buttazzo, M. D. Natale, and M. Spuri. Design and programming tools for time critical applications. *Real-Time Systems Journal*, 14:251–267, 1998.

[6] Artis, Torino, Italy, www.artis.it. *Artifex 4.1 Tutorial*, 1997.

[7] N. Audsley, A. Burns, R. Davis, K. Tindell, and A. Wellings. Fixed priority preemptive scheduling: An historical perspective. *Real-Time Systems*, (8):173–198, 1995.

[8] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, (11):284–292, Sept. 1993.

[9] N. Audsley, A. Burns, M. Richardson, and A. Wellings. STRESS: a simulator for hard real-time systems. *Software - Practice and Experience*, 24(6):543–564, 1994.

[10] N. C. Audsley, A. Burns, and A. J. Wellings. Deadline monotonic scheduling theory and application. *Control Eng. Practice*, 1(1):71–78, 1993.

[11] T. Baker and A. Shaw. The cyclic executive model and Ada. *Journal of Real-Time Systems*, (1), 1989.

[12] T. P. Baker. A stack-based resource allocation policy for realtime processes. In *IEEE Real-Time Systems Symposium*, 1990.

[13] S. Balaji, L. Patnaik, L. Jenkins, and P. Goel. S-nets: A Petri net based model for performance evaluation of real-time scheduling algorithms. *Journal of Parallel and Distributed Computing*, 15:225–237, 1992.

[14] M. Baldassari and G. Bruno. A methodology and environment for the object oriented analysis and design of real time systems. In *Proceedings Euromicro Workshop on Real-Time Systems*, 1990.

[15] M. Baldassari and G. Bruno. PROTOB: An object oriented methodology for developing discrete event dynamic systems. *Comput. Lang.*, 16(1):39–63, 1991.

[16] I. Bate and A. Burns. Schedulability analysis of fixed-priority real-time systems with offsets. In *Proc. Euromicro Workshop on Real-Time Systems*, 1997.

[17] B. Baumgarten, H. J. Burkhardt, P. Ochsenschläger, and R. Prinoth. The signing of a contract - a tree-structured application modelled with Petri net building blocks. In G. Goos and J. Hartmanis, editors, *Advances in Petri Nets 1985*, number 222 in Lecture Notes in Computer Science. Springer, 1985.

[18] B. Baumgarten, P. Ochsenschläger, and R. Prinoth. *Protocol Specification, Testing, and Verification*, chapter Building Blocks for Distributed System Design, pages 19–38. Elsevier Science Publishers B.V. (North-Holland), 1986.

[19] G. Berthelot. *Petri Nets: Central Models and their Properties*, chapter Transformations and Decompositions of Nets, pages 359–376. Number 254 in LNCS. Springer, 1986.

[20] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE Transactions on Software Engineering*, 17(3):259–273, 1991.

[21] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1998.

[22] G. Bruno, R. Agarwal, A. Castella, and M. Pescarmona. CAB: an environment for developing concurrent applications. In *Application and Theory of Petri Nets*, number 935 in LNCS, pages 141–160, 1995.

[23] G. Bruno and A. Castella. Software models for real-time systems. In *Proceedings Euromicro Workshop on Real-Time Systems 1992*, 1992.

[24] G. Bruno, A. Castella, G. Macario, and M. Pescarmona. Scheduling hard real-time systems using high-level Petri nets. In *Application and Theory of Petri Nets*, number 616 in LNCS, pages 93–112, 1992.

[25] G. Bruno, A. Castella, I. Pavesio, and M. Pescarmona. A new Petri net based formalism for specification, design and analysis of real-time systems. In *Proceedings IEEE Real-Time Systems Symposium*, 1993.

[26] G. Bucci, M. Campanai, and P. Nesi. Tools for specifying real-time systems. *Real-Time Systems Journal*, 8(2/3), 1995.

[27] A. Burns and A. Lister. A framework for building dependable systems. *Computer Journal*, 34(2):173–181, 1991.

[28] A. Burns, K. Tindell, and A. Wellings. Effective analysis for engineering real-time fixed priority schedulers. *IEEE Transactions on Software Engineering*, 21(5), 1995.

[29] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*. Addison-Wesley, 1996.

[30] A. Burns and A. J. Wellings. HRT-HOOD: A structured design method for hard real-time systems. *Real-Time Systems Journal*, (6):73–114, 1994.

[31] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley and Sons, 1996.

[32] G. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer, 1997.

[33] G. Buttazzo and F. Sensini. Optimal deadline assignment for scheduling soft aperiodic tasks in hard real-time environments. In *Proc. 3rd Int. Conf. on the Engineering of Complex Computer Systems*, 1997.

[34] U. Buy and R. Sloan. Analysis of real-time programs with simple time Petri nets. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA'94)*, pages 228–239, 1994.

[35] A. Caloini, G. Magnani, and M. Pezze. Software design of robot controllers with Petri nets: a case-study. In *Proceedings IEEE Int. Conf. on Systems, Man, and Cybernetics*, 1996.

[36] CCITT. Specification and description language CCITT Z.100. Geneva,, 1992.

[37] M. Chen and K. Lin. Dynamic priority ceilings: A concurrency control protocol for real-time systems. *Real-Time Systems*, (2), 1990.

[38] Y.-J. Chen, D. Mosse, and S.-K. Chang. A framework for modelling dependable real-time distributed systems. *International Journal of Systems Science*, 28(11):1025–1043, 1997.

[39] B. Cheng, A. Stoyenko, T. Marlowe, and S. Baruah. The allocation and scheduling of precedence- and timing-constrained tasks with communication delays. In *Proc. Int. Conf. on the Engineering of Complex Computer Systems*, 1996.

[40] H. Chetto and M. Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Transactions on Software Engineering*, 15(10), 1989.

[41] H. Chetto, M. Silly, and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Journal of Real-Time Systems*, (2):1990, 1990.

[42] T.-Y. Cheung. Petri nets for protocol engineering. *Computer Communications*, 19:1250–1257, 1996.

[43] E. Clarke, E. Emerson, and A. Sistla. Automatic verfication of finite-state concurrent systems using temporal logic specifications. *ACM Trans. on Programming Languages and Systems*, 8(2):244–263, 1986.

[44] J. O. Coplien and D. C. Schmidt, editors. *Pattern Languages of Program Design*. Addison-Wesley, 1995.

[45] R. L. Cruz. A calculus for network delay, part i: Network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, 1991.

[46] R. L. Cruz. A calculus for network delay, part ii: Network analysis. *IEEE Transactions on Information Theory*, 37(1):132–141, 1991.

[47] B. Dasarathy. Timing constraints of real-time systems: Constructs for expressing them and methods of validating them. *IEEE Transactions of Software Engineering*, SE-11(1), 1985.

[48] Y. Deng and C.-R. Yang. Architecture-driven modeling of real-time concurrent systems with applications in FMS. *Journal of Systems and Software*, 45(1):61–78, 1999.

[49] S. Donatelli and G. Franceschinis. The PSR methodology: Integrating hardware and software models. In *Application and Theory of Petri Nets*, LNCS, 1996.

[50] B. Douglass. *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*. Addison Wesley, 1999.

[51] C. Ebert. Experiences with colored predicate-transition nets for specifying and prototyping embedded systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part B,*, 28(5):641–652, Oct. 1998.

[52] R. Elmstrom, R. Lintulampi, and M. Pezze. Giving semantics to SA/RT by means of high-level Petri nets. *Real-Time Systems Journal*, 5:249–271, 1993.

[53] R. Esser. *An Object Oriented Petri Net Approach to Embedded System Design*. PhD thesis, ETH Zurich, 1996.

[54] A. Ferscha. *Decentralized Systems*, chapter Modelling Mappings of Parallel Programms onto Parallel Architectures with the PRM-Net Model, pages 349–362. Elsevier, 1990.

[55] H. Fleischhack and B. Grahlmann. Toward compositional verification of SDL systems. In *Proceedings 31st Annual Hawaii International Conference on System Sciences*, volume VII, page 414, 1998.

[56] P. Freedman, D. Gaudreau, R. Boutaba, and A. Mehaoua. The two real-time solitudes: computerized control and telecommunications. In *Proc. 2nd IEEE Int. Conf. on Engineering of Complex Computer Systems*, 1996.

[57] C. Frydman, L. Torres, N. Giambiasi, and M. L. Goc. Building executable models form expert conceptual models. In *Proc. Simulators International XIII*, 1996.

[58] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[59] R. Gerber, S. Hong, and M. Saksena. Guaranteeing end-to-end timing constraints by calibrating intermediate processes. In *proc. IEEE Real-Time Systems Symposium*, 1994.

[60] V. Gerogiannis, A. Kameas, and P. Pintelas. Comparative study and categorization of high-level Petri nets. *Journals of Systems and Software*, 43:133–160, 1998.

[61] T. Ghazalie and T. Baker. Aperiodic servers in a deadline scheduling environment. *Real-Time Systems*, (9), 1995.

[62] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezze. A unified high-level Petri net formalism for time-critical systems. *IEEE Transactions on Software Engineering*, 17(2):160–172, 1991.

[63] C. Ghezzi, S. Morasca, and M. Pezze. Validating timing requirements for time basic net specifications. *Journ. Real-Time Systems*, pages 97–117, 1994.

[64] K. Ghose, S. Aggarwal, P. Vasek, S. Chandra, A. Raghav, A. Ghosh, and D. Vogel. Asserts: A toolkit for real-time software design, development, and evaluation. In *Proceedings Euromicro Workshop on Real-Time Systems*, 1997.

[65] S. Ghosh, R. Melhem, and D. Mosse. Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(3):272–284, Mar. 1997.

[66] J. Gil and D. Lorenz. Design patterns and language design. *Computer*, pages 118–120, Mar. 1998.

[67] J. Goossens and C. Hernalsteen. A tool for statistical analysis of hard real-time scheduling algorithms. In *Proceedings Simulation 98*, 1998.

[68] B. Grahlmann. The state of PEP. In *Proceedings AMAST 1998*, 1998.

[69] K. Gresser. *Echtzeitnachweis ereignisgesteuerter Realzeitsysteme*. Number 268 in Fortschrittberichte Reihe 10. VDI Verlag, 1993.

[70] M. Gries. Modeling a memory subsystem with Petri nets: a case study. In *Workshop Hardware Design and Petri Nets (HWPN 98)*, 1998.

[71] M. Gries, J. Janneck, and M. Naedele. Reusing design experience for Petri nets through patterns. In *Proceedings High Performance Computing (HPC) 1999*, 1999.

[72] M. G. Harbour, M. H. Klein, and J. P. Lehoczky. Fixed priority scheduling of periodic tasks with varying execution priority. In *Proceedings IEEE Real-Time Systems Symposium*, 1991.

[73] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[74] P. Harter. Response times in level-structured systems. *ACM Transactions on Computer Systems*, 5(3):232–248, Aug. 1987.

[75] M. Heiner. Petri net based system analysis without state explosion. In *Proceedings High Performance Computing HPC'98*, 1998.

[76] R. G. Herrtwich and G. Hommel. *Nebenläufige Programme.* Springer Verlag, 1994.

[77] D. Hils. Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages and Computing*, 3:69–101, 1992.

[78] N. Homayoun and P. Ramanathan. Dynamic priority scheduling of periodic and aperiodic tasks in hard real-time systems. *Real-Time Systems*, (6), 1994.

[79] C. Hou and K. Shin. Allocation of periodic task modules with precedence and deadline constraints in distributed real-time systems. In *Proc. IEEE Real-Time Systems Symposium*, 1992.

[80] M. Hsueh, T. Tsai, and R. Iyer. Fault injection techniques and tools. *IEEE Computer*, (4), 1997.

[81] P. Huber, K. Jensen, and R. Shapiro. *Advances in Petri Nets 1990*, chapter Hierarchies in coloured Petri nets, pages 313–341. Springer, 1991.

[82] M. Humphrey and J. Stankovic. Caisarts: A tool for real-time scheduling assistance. In *Proceedings RTAS 96*, 1996.

[83] J. Stankovic et. al. Strategic directions in real-time and embedded systems. *ACM Computing Surveys*, 28(4), 1996.

[84] J. Janneck and M. Naedele. Modeling a die bonder with Petri nets: A case study. *IEEE Transactions on Semiconductor Manufacturing*, 11(3):404–409, 1998.

[85] J. Janneck and M. Naedele. Modeling hierarchical and recursive structures using parametric Petri nets. In *Proceedings High Performance Computing (HPC) 1999*, 1999.

[86] K. Jeffay, D. Stanat, and C. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *Proc. IEEE Real-Time Systems Symposium*, 1991.

[87] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, volume 1-3 of *EATCS Monographs in Computer Science.* Springer-Verlag, 1992.

[88] M. Joseph, editor. *Real-time systems - Specification, Verification, and Analysis.* Prentice Hall, 1996.

[89] D. Katcher. *Engineering and Analysis of Real-Time Operating Systems.* PhD thesis, Carnegie Mellon University, 1994.

[90] D. Katcher, H. Arakawa, and J. Strosnider. Engineering and analysis of fixed priority schedulers. *IEEE Transactions on Software Engineering*, 19(9):920–934, 1993.

[91] M. Kishinevsky, J. Cortadella, A. Kondratyev, L. Lavagno, A. Taubin, and A. Yakovlev. Coupling asynchrony and interrupts: Place chart nets. In *Application and Theory of Petri Nets*, number 1248 in LNCS, pages 328–347, 1997.

[92] M. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour. *A Practitioner's Handbook for Real-Time Analysis.* Kluwer, 1993.

[93] S. Koriem. R-nets for the performance evaluation of hard real-time systems. *Journal of Systems and Software*, 46:41–58, 1999.

[94] H. Kwak, I. Lee, A. Philippou, J. Choi, and O. Sokolsky. Symbolic schedulability analysis of real-time systems. In *Proceedings IEEE Real-Time Systems Symposium*, 1998.

[95] C. Lai and H. R. Callison. A framework for simulation of concurrency control policy in real-time systems. In *Proceedings IEEE Real-Time Technology and Applications Symposium*, 1996.

[96] J.-Y. LeBoudec. Network calculus made easy. Technical Report EPFL-DI 96/218, Ecole Polytechnique Federale, Lausanne (EPFL), 1996.

[97] J. Lehoczky. Fixed-priority scheduling of periodic task sets with arbitrary deadlines. In *Proc. IEEE Real-Time Systems Symposium*, 1990.

[98] J. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive sustems. In *Proceedings IEEE Real-Time Systems Symposium 1992*, 1992.

[99] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings IEEE Real-Time Systems Symposium*, 1989.

[100] J. Lehoczky, L. Sha, and J. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proceedings IEEE Real-Time Systems Symposium*, 1987.

[101] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2:237–250, 1982.

[102] N. Leveson and J. Stolzy. Safety analysis using Petri nets. *IEEE Transactions on Software Engineering*, SE-13(3), 1987.

[103] C. Lindemann. *Performance Modelling with Deterministic and Stochastic Petri Nets.* John Wiley & Sons, 1998.

[104] R. Lintulampi and P. Pulli. Graphical prototyping of tasking behavior. In *Proceedings IFAC Workshop on Real-Time Programming*, 1991.

[105] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[106] J. W. S. Liu, J.-L. Redondo, Z. Deng, T.-S. Tia, R. Bettati, A. Silberman, M. Storch, R. Ha, and W.-K. Shih. Perts: A prototyping environment for real-time systems. Technical report, Department of Computer Science, University of Illinois, Urbana, 1993.

[107] C. D. Locke. Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives. *Journal of Real-Time Systems*, (4):37–53, 1992.

[108] B. Mahony and I. Hayes. A case-study in timed refinement: A mine pump. *IEEE Trans. on Software Engineering*, 18(0):817–825, 1992.

[109] G. Manimaran and C. S. R. Murthy. A new scheduling approach supporting different fault-tolerant techniques for real-time multiprocessor systems. *Microprocessors and Microsystems*, 21:163–173, 1997.

[110] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modeling with Generalized Stochastic Petri Nets*. John Wiley & Sons, 1995.

[111] P. Merlin and D. Farber. Recoverability of communication protocols - implications of a theoretical study. *IEEE Transactions on Communications*, Sept. 1976.

[112] A. Mok and D. Chen. A multiframe model for real-time tasks. *IEEE Transactions on Software Engineering*, 23(10):635–644, Oct. 1997.

[113] D. Moldt. *Hoehere Petrinetze als Grundlage fuer Systemspezifikationen*. PhD thesis, Universitaet Hamburg, 1996.

[114] S. Morasca and M. Pezze. *Real-Time Systems, Theory and Applications*, chapter Using high-level Petri nets for testing concurrent and real-time systems, pages 119–131. Elsevier, 1990.

[115] T. Murata. Petri nets: Properties, analysis, and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[116] M. Naedele. Petri net models for single-processor real-time scheduling. TIK-Report 61, ETH Zurich, Computer Engineering and Networks Laboratory (TIK), 1998.

[117] M. Naedele. Fault-tolerant real-time scheduling under execution time constraints. In *Proceedings 6th Int. Conf. on Real-Time Computing Systems and Applications, RTCSA 1999*, 1999.

[118] M. Naedele. A survey of real-time scheduling tools. TIK-Report 72, ETH Zurich, Computer Engineering and Networks Lab (TIK), Apr. 1999.

[119] M. Naedele and J. Janneck. Design patterns in Petri net system modeling. In *Proceedings ICECCS'98*, 1998.

[120] M. Naedele, L. Thiele, and M. Eisenring. General task and resource models for processor task scheduling. TIK-Report 45, Computer Engineering and Networks Lab, Swiss Federal Institute of Technology, 1998.

[121] M. Naedele, L. Thiele, and M. Eisenring. Characterizing variable task releases and processor capacities. In *Proc. 14th IFAC World Congress*, 1999.

[122] L. Nigro and F. Pupo. Modeling and analysing DART systems through high-level Petri nets. In *Application and theory of Petri Nets 1996*, number 1091 in LNCS, 1996.

[123] L. Nigro and F. Pupo. Using Design/CPN for the schedulability analysis of actor systems with timing constraints. In *Proceedings Design/CPN Workshop '98*, 1998.

[124] Oberon microsystems. Jbed whitepaper: Component software and real-time computing. www.jbed.com.

[125] D. Peng and K. Shin. Modeling of concurrent task execution in a distributed system for real-time control. *IEEE Transactions on Computers*, C-36(4):500–516, 1987.

[126] J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice Hall, 1981.

[127] R. Pettit and H. Gomaa. Integrating Petri nets with design methods for concurrent and real-time systems. In *Proceedings IEEE Int. Conference on the Engineering of Complex Computer Systems*, 1996.

[128] C. Puchol and A. Mok. Integrated design tools for hard real-time systems. In *Proceedings IEEE Real-Time Systems Symposium*, 1998.

[129] R. Rajkumar. *Synchronization in real-time systems: A priority inheritance approach.* Kluwer, 1991.

[130] K. Ramaritham, J. Stankovic, and P. Shiah. Efficient scheduling algorithms for real-time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):184–194, 1990.

[131] K. Ramaritham, J. Stankovic, and W. Zhao. Distributed scheduling of tasks with deadlines and resource requirements. *IEEE Trans. on Computers*, 38(8), 1989.

[132] C. Ramchandani. Analysis of asynchronous concurrent systems by timed Petri nets. Technical Report 120, Project MAC, Massachussetts Institute of Technology, 1974.

[133] S. Ramos-Thuel and J. Lehoczky. On-line scheduling of hard deadline aperiodic tasks in fixed-priority systems. In *Proceedings IEEE Real-Time Systems Symposium*, 1993.

[134] J.-L. Redondo. Schedulability analyzer tool. Master's thesis, Graduate College of the University of Illinois at Urbana-Champaign, 1993.

[135] W. Reisig. *Petri Nets, An Introduction.* Springer-Verlag, 1985.

[136] W. Reisig. *A Primer in Petri Net Design.* Springer-Verlag, 1992.

[137] L. Rice and A. Cheng. Timing analysis of the X-38 space station crew return vehicle avionics. In *Proc. IEEE Real-Time Technology and Applications Symposium*, 1999.

[138] L. Rising. Patterns: A way to reuse expertise. *IEEE Communications magazine*, pages 34–36, Apr. 1999.

[139] B. Selic, G. Gullekson, and P. Ward. *Real-time object-oriented modeling.* John Wiley and Sons, 1994.

[140] A. Semenov. Designing an asynchronous processor with Petri nets. *IEEE Micro*, pages 54–63, March/April 1997.

[141] F. Sensini, G. Buttazzo, and P. Ancilotti. GHOST: A tool for simulation and analysis of real-time scheduling algorithms. In *Proceedings IEEE Real-Time Systems Education Workshop 1997*, 1997.

[142] L. Sha, J. P. Lehoczky, and R. Rajkumar. Solutions for some practical problems in prioritized preemptive scheduling. In *Proceedings of the 1986 IEEE Real-Time Systems Symposium*, 1986.

[143] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, Sept. 1990.

[144] L. Sha and S. Sathaye. A systematic approach to designing distributed real-time systems. *IEEE Computer*, Sept. 1993.

[145] C. Shen, K. Ramaritham, and J. Stankovic. Resource reclaiming in multiprocessor real-time systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(4):382–397, 1993.

[146] B. Shizuki, M. Toyoda, E. Shibayama, and S. Takahashi. Visual patterns + multi-focus fisheye view: An automatic scalable visualization technique of data-flow visual program execution. In *Proceedings Visual Languages 1998*, 1998.

[147] A. Silberman. *RTM - Design and Implementation.* PhD thesis, Graduate College of the University of Illinois at Urbana-Champaign, 1997.

[148] R. H. Sloan and U. Buy. Reduction rules for time Petri nets. *Acta Informatica*, 33:687–706, 1996.

[149] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems*, (1):27–60, 1989.

[150] J. Stankovic. Misconceptions about real-time computing. *IEEE Computer*, Oct. 1988.

[151] J. Stankovic, K. Ramaritham, and S. Cheng. Evaluationof a flexible task scheduling algorithm for distributed hard real-time systems. *IEEE Trans. on Computers*, C-34(12), 1985.

[152] J. Stankovic, M. Spuri, M. D. Natale, and G. Buttazzo. Implications of classical scheduling results for real-time scheduling. *IEEE Computer*, pages 16–25, June 1995.

[153] J. Stankovic, M. Spuri, K. Ramaritham, and G. Buttazzo. *Deadline Scheduling for Real-Time Systems.* Kluwer, 1998.

[154] M. Storch. *A Framework for the Simulation of Complex Real-Time Systems.* PhD thesis, Graduate College of the University of Illinois at Urbana-Champaign, 1997.

[155] M. Storch and J. W.-S. Liu. DRTSS: A simulation framework for complex real-time systems. In *Proceedings Complex Systems Design and Synthesis Technology Workshop*, 1994.

[156] M. Storch and J. W.-S. Liu. A simulation environment for distributed real-time systems. Technical report, Department of Computer Science, University of Illinois, Urbana, 1994.

[157] T. Strayer. *Function-Driven SCheduling: A General Framework for Expression and Analysis of Scheduling.* PhD thesis, School of Engineering and Applied Science, University of Virginia, 1992.

[158] J. Strosnider, J. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Trans. on Computers*, 44(1):73–91, Jan. 1995.

[159] S. Thuel and J. Lehoczky. Algorithms for scheduling hard aperiodic tasks in fixed-priority systems usign slack stealing. In *IEEE Real-Time Systems Symposium*, 1994.

[160] K. Tindell, A. Burns, and A. Wellings. Allocating hard real-time tasks: An NP-hard problem made easy. *Journal of Real-Time Systems*, (4), 1992.

[161] K. Tindell, A. Burns, and J. Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Systems*, 6(2):133–151, 1994.

[162] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 40, 1994.

[163] H. Tokuda and M. Kotera. A real-time tool set for the ARTS kernel. In *Proceeding IEEE Real-Time Systems Symposium*, 1988.

[164] H. Tokuda and M. Kotera. Scheduler 1-2-3: An interactive schedulability analyzer for real-time systems. In *Proceedings COMPSAC '88*, 1988.

[165] M. Toyoda, B. Shizuki, S. Takahashi, S. Matsuoka, and E. Shibayama. Supporting design patterns in a visual parallel data-flow programming environment. In *Proceedings Visual Languages 1997*, 1997.

[166] Tri-Pacific Software, Alameda, California 94501. *PERTS Version 3.0 User Manual.*

[167] Tri-Pacific Software, Alameda, California 94501. *PERTSSim Version 1.1 User Manual.*

[168] J. Tsai, S. Jennhwa, and Y.-H. Chang. Timing constraint Petri nets and their application to schedulability analysis of real-time system specifications. *IEEE Transactions on Software Engineering*, 21(1):32–48, 1995.

[169] A. Valmari. *Lecture in Petri nets I: Basic Models*, chapter The State Explosion Problem. Number 1491 in LNCS. Springer, 1998.

[170] J. M. Vlissides, J. O. Coplien, and N. L. Kerth, editors. *Pattern Languages of Program Design 2.* Addison-Wesley, 1996.

[171] H. Völzer. Verifying fault tolerance of distributed algorithms formally - an example. In *Proceedings of CSD'98*, 1998.

[172] M. von der Beeck. A comparison of Statecharts variants. In *Proceedings third international symposium on formal techniques in real-time and fault-tolerant systems*, number 863 in LNCS, 1994.

[173] S. D. Vroey, J. Goossens, and C. Hernalsteen. A generic simulator of real-time scheduling algorithms. In *Proceedings Simulation 96*, 1996.

[174] F. Wang, K. Ramaritham, and J. Stankovic. Bounds on the performance of heuristic algorithms for multiporcessor scheduling of hard real-time tasks. In *Proc. IEEE Real-Time Systems Symposium*, 1992.

[175] B. Witlox, P. van der Wolf, E. Aarts, and W. van der Aalst. Performance analysis of dataflow architectures using timed coloured Petri nets. In *Workshop Hardware Design and Petri Nets HWPN'98*, 1998.

[176] J. Xu and D. Parnas. On satisfying timing constraints in hard real-time systems. *IEEE Transactions on Software Engineering*, 19(1):70–84, 1993.

[177] P. Zave. The operational versus the conventional approach to software development. *CACM*, 27(2), 1984.

[178] L. Zhang and B. Chaib-draa. A design methodology for real-time systems to be implemented on multiprocessor machines. *Journal of Systems and Software*, 33:37–56, 1996.

[179] W. Zhao, K. Ramaritham, and J. Stankovic. Scheduling tasks with resource requirements in hard real-time systems. *IEEE Trans. on Software Engineering*, SE-13(5), 1987.

[180] W. M. Zuberek, R. Govindarajan, and F. Suciu. Timed colored Petri net models of distributed memory multithreaded processors. Proceedings CP-nets'98 Workshop, Tech. Report PB-532, Department of Computer Science, University of Aarhus, 1998.

[181] R. Zurawski and M.-C. Zhou. Petri nets and industrial applications: A tutorial. *IEEE Transactions on Industrial Electronics*, 41(6):567–583, Dec. 1994.

Curriculum Vitae

Date and Place of Birth:

15.2.1972 in Schweinfurt, Germany

Education:

since 1997	Research assistant and doctoral student in the Computer Engineering Group at ETH Zurich, Switzerland
1992–1997	Electrical and electronics engineering at Ruhr-University Bochum, Germany, and Purdue University, West Lafayette, USA; graduated as Dipl.-Ing. from Ruhr-University Bochum
1982–1991	Rhön-Gymnasium Bad Neustadt, Germany, and Thurston Upper School, Thurston, UK

Practical Experience:

11/1996 – 4/1997	Diploma thesis in collaboration with Vebacom Service (o.tel.o), Bochum, Germany
6/1996 – 7/1996	Internship at Siemens Components, Singapore
10/1994 – 6/1995	Student assistant in the Communication Systems Group, Ruhr-University Bochum, Germany
2/1994 – 3/1995	Internship at Dr. Brandt Messtechnik, Bochum, Germany
10/1991 – 9/1992	Military service in electronic maintenance company