University of Paderborn, Germany
Prof. Dr. rer. nat. Franz J. Rammig
Computer Science, Design of Parallel Systems

Carsten Böke

Automatic Configuration of Real-Time Operating Systems and Real-Time Communication Systems for Distributed Embedded Applications

HEINZ NIXDORF INSTITUT
Universität Paderborn

Address of the author:
Carsten Böke
Heinz Nixdorf Institute
University of Paderborn
Design of Parallel Systems Group
Fürstenallee 11
D-33102 Paderborn, Germany

email: cboeke@gmx.de

Bibliographic notes:
Carsten Böke:
Automatic Configuration of Real-Time Operating Systems and Real-Time Communication Systems for Distributed Embedded Applications
Heinz Nixdorf Institute, Paderborn, Germany
HNI-Verlagsschriftenreihe No. 142/2004
ISBN 3-935433-51-4
http://wwwhni.uni-paderborn.de/

Also:
Submitted as a dissertation to the Faculty of Computer Science, Electrical Engineering, and Mathematics of the University of Paderborn, Germany.
Supervisors:

1. Prof. Dr. rer. nat. Franz J. Rammig, University of Paderborn
2. Prof. Dr. rer. nat. Odej Kao, University of Paderborn

Date of public examination: 19^{th} December, 2003

Keywords:
Operating systems, embedded systems, distributed systems, communication, real-time, configuration, customisation, analysis, schedulability, software design, software synthesis.

Automatic Configuration of Real-Time Operating Systems and Real-Time Communication Systems for Distributed Embedded Applications

Dissertation

A thesis submitted to the
Faculty of Computer Science, Electrical Engineering, and Mathematics
of the
University of Paderborn
in partial fulfilment of the requirements for the
degree of *Dr. rer. nat.*

Carsten Böke

Paderborn, Germany
October 30, 2003

Supervisors:

1. Prof. Dr. rer. nat. Franz J. Rammig, University of Paderborn
2. Prof. Dr. rer. nat. Odej Kao, University of Paderborn

Date of public examination: December 19, 2003

Acknowledgements

This work was carried out during my time at the HEINZ NIXDORF INSTITUTE (HNI), an interdisciplinary centre of research and technology of the University of Paderborn, Germany. I like to thank all my colleagues who helped me to make this work a success. I am greatful to them all, not just for their technical support, but more importantly, for the experiences I shared with them.

First of all, I would like to express my gratitude to Prof. Dr. Franz J. Rammig and Prof. Dr. Odej Kao for supervising and vice-supervising this work, respectively. I sincerly thank Prof. Rammig for the helpful advices and direction he has given me, for his support and constant guidance during this work. Within his working group I had the opportunity to establish and to work on different research projects and to advice students. I was lucky enough to work in a heterogeneous working group, where I did not just benefit from the different research areas of my colleagues, but where I personally enjoyed that many colleagues came from foreign countries.

I like to thank my colleagues who helped me in different ways to realise this project and gave me a good time at the institute. Due to their wide range of interest, knowledge, and research projects they all brought in different aspects to my work.

I like to thank Dr. Carsten Ditze, who founded the basis for my thesis. It was a pleasure for me to contribute to and to learn from a lot of discussions with him about his design of the underlying operating system DREAMS. Thanks also to Dr. Thomas Lehmann for his co-operation during the project. In the same way I owe a significant debit of gratitude to Dr. Ramakrishna Prasad Chivukula, who supported me during the end of my work. It was alo a pleasure for me to share my office with Simon Oberthür and to talk with him about future extensions to TEReCS. I enjoyed greatfully all the discussions and the time with them.

Furthermore, I have to thank all readers of my thesis who helped me not only to fix technical parts. Special thanks here again to Dr. Prasad Chivukula and to Prof. Pramod Chandra P. Bhatt, who revised major parts of the document.

But writing this thesis would not have been possible without the support of additional wonderful people. At the top of the list has to be my dear wife Deina. She supported

me with a lot of patience and encouraged me always when my confidence and patience ran low. Additionally, she had most understanding when we would have preferred to do other things together.

The closing and also very important dedications go to my dear parents Bärbel and Fritz. Without their never ending support and care it would be questionable if I would ever have come so far.

The work on TEReCS had also been supported under the Priority Programme 1040 "Entwurf und Entwurfsmethodik eingebetteter Systeme" by the *Deutsche Forschungsgemeinschaft* (DFG) under contract RA 612/4 with the title "Entwurf konfigurierbarer, echtzeitfähiger Kommunikationssysteme".

Paderborn, October 2003.

Contents

List of Figures

List of Tables

Abstract

The design of distributed software for embedded systems in products like automobiles, trains, airplanes, ships, automatic manufacturing lines, robots, telecommunication systems, etc. becomes incredibly complex due to the huge amount of parallel working and interconnected microprocessors. Exactly their mutual in/out-dependencies and, therefore, their implemented communications make them very complex. It is hard to analyse and predict their behaviour. But for hard real-time control the guarantee of a special behaviour is indispensable. Especially, the operating systems, which are often required to provide the communication functions, introduce a lot of overhead and non-determinism into the systems.

In this thesis we develop a methodology to generate the operating systems and the communication system for a distributed embedded real-time application. The operating systems for each node of the network and the communication system is thereby generated from a customisable library operating system construction kit. The operating systems for each microprocessor and the communication system are assembled from tuneable components of a library. This is done by a new configuration approach named *Puppet Configuration*. The final systems are highly adapted to the requirements of the application. Thus, not required functions are excluded and the remaining functions are optimally configured to serve the given use case.

Additionally, the behaviour of the finally configured distributed system is analysed in order to ensure a temporal correct behaviour. A new *Time-triggered Event Scheduling* scheme is used to attain this information before the system is targeted and implemented. Bottlenecks, overload conditions, bursts, and also idle phases are detected.

The configuration phase and the analysis phase are embedded into a contemporary design methodology. The configuration has impacts on the analysis and the analysis influences the configuration. Both phases are executed alternately until a valid and working configuration is found (or another configuration cannot be generated).

This innovative design approach of using configuration and prediction of its behaviour reveals a lot of potential for shortening the design time and, therefore, the time-to-market of a new embedded real-time system for a distributed controlled product.

CHAPTER 1

Summary

Nowadays, distributed control replaces more and more centralised control. In modern products with embedded systems a number of control units are integrated. The communication aspect between these control units becomes more and more complex and important. In the past decades the industry and sometimes also the research have ignored these distributed systems due to their complexity. A lot of efforts had been spent for the analysis and the design of local systems. However, the research community transferred its schedulability theory in some aspects to distributed systems. Nevertheless, in practice a lot of hand work is still done in order to programme distributed systems. Some of the high-level development tools for real-time control systems, like ASCET-SD [45], StateMate [64], or CAMeL [70, 107, 151], produce target code, but this must often be adapted or even distributed to the final target by hand. This happens, because the tool does not support any operating system or the one which is supported cannot be applied to the final target. For this reason, the operating system services have to be programmed by hand or the code have to be ported to another operating system or target.

In order to specify inherently parallel working applications, synchronous languages had been defined. Synchronous languages are based on the simultaneity principle: In reactive systems multiple actions or computations happen simultaneously. Languages like Lustre [58] or Esterel [12] handle the communication between parallel and distributed systems synchronously. All processes compute one discrete time step and then they communicate at the same time. They simply define a synchronous interleaving between calculation and communication. Thereby, they assume that the communication overhead is constant and can be ignored. However, this is a very pragmatic approach and leads to several problems for their implementation on really distributed systems.

This thesis is related to *embedded systems*. The development of, especially, *distributed systems* is supported. *Real-time aspects* are additionally considered. Thus, the real-time communication between distributed tasks on several embedded control units is mainly considered. Especially, the adaptation of the target operating systems for each control unit with its communication facilities to one specific distributed application is the main topic.

This goal is achieved by using techniques from the *configuration theory*. The implementation of operating and communication services can vary. A specific implementation of a service can only work, when the environment assures certain properties. Operating system code that is written for one specific use case will be reused, whenever that use case is detected.

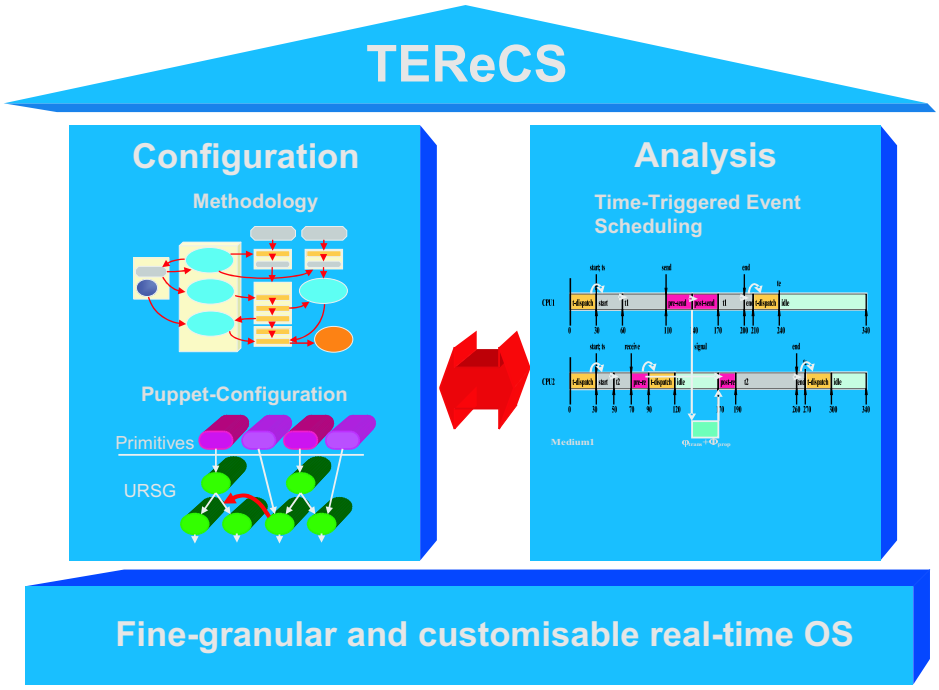

Fig. 1.1: TEReCS is mainly built on two basic concepts: configuration and timing analysis.

This goal is achieved by applying new ideas and concepts to the design domain for distributed embedded applications. A new and contemporary approach to the design of operating and communication systems has been developed. It is highly flexible and extensible. Additionally, the memory footprint and the real-time constraints are considered during the design phase. The design process runs automatically, when the information about the application's behaviour and its requirements are specified. The design process exploits application specific knowledge for the configuration of the operating and communication systems. It reduces dramatically the design time for distributed and efficient operating and communication systems. This is due to a hierarchical and highly flexible configuration concept. Additionally, the timeliness execution of all communicating tasks is checked. On the one hand, this approach supports the application designer with internal knowledge about the operating systems activities in order to identify bottlenecks of the system design. On the other hand, the application designer must not be an operating system expert in order to select or parameterise certain operating system services.

Thus, it becomes possible to support nearly all application scenarios and a huge variety of hardware platforms with only one operating system library. The final target operating systems are slim and efficient, because they support exactly the application scenario and not more. Configuration is a promising approach for software reuse.

Another very exceptional concept is the integration of the configuration and the timing analysis. During the automatic design process configuration and timing analysis have mutual influences on each other. The configuration determines the operating system's overhead, which is considered by the timing analysis, and the timing analysis may forbid certain configuration options due to overload conditions.

All of these aspects led to the development of *Tools for Embedded Real-time Communication Systems* (TEReCS) that are described in this thesis. The tool suite consists of the following tools:

- **TGEN** is the configurator
- **TCLUSTER** implements the hierarchical cluster-aided configuration.
- **TANA** is the timing analyser
- **TDESIGN** encapsulates the design methodology with its design cycle and calls TCLUSTER, TGEN, and TANA.
- **TEDIT** is an graphical editor for the domain knowledge databases that are used for the configuration.

CHAPTER 2

Introduction

2.1 Motivation

One of the big challenges for the development of embedded applications these days is the integration of many co-operating applications into one or even a set of products. Nowadays, many applications are distributed over several *embedded control units* (ECUs). Examples are the entry of intelligent and fully automatic machines and robots into modern production processes and also the "computerisation" of classical transportation vehicles, like trains, airplanes and, especially, automobiles.

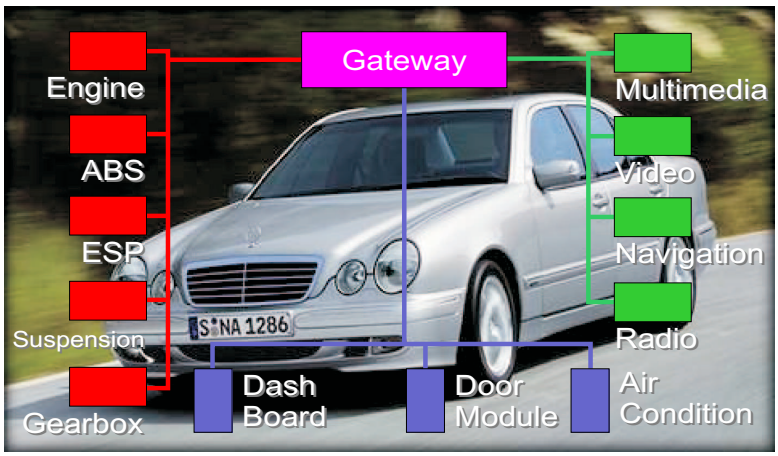

Fig. 2.1: Embedded systems in modern automobiles.

A modern car contains up to 100 microprocessors that control systems like the powertrain with its motor management, chassis, *Anti-lock Braking System* (ABS), *Electronic Stabilisation Programme* (ESP), suspension, airbag, and gear box control, the body module with its dashboard, indicating, lighting, seat, door, and key controllers, and the comfort module

with its air condition, telematics, navigation, radio, video, multimedia, telephone and Internet connectivity. The ECUs are interconnected via different communication links. Their integration is complex and sometimes absolutely critical.

All of these systems require different hardware and also different operating and communication services. Due to the general development process in companies, different versions of these systems in different generations or families of a product and sometimes also in complete different systems are implemented on top of the same microprocessor and operating system. This is mainly done in order to reuse the development know-how of the engineers and to reuse previous software developments. New hardware (microprocessors and communication links) can often only be used for a new product version, when a development environment with an appropriate operating system support is available. Often the operating systems that are used embody the virtual machine idea. Therefore, they implement a lot of general services and are more or less monolithic kernel architectures with coarse-grained module extensions.

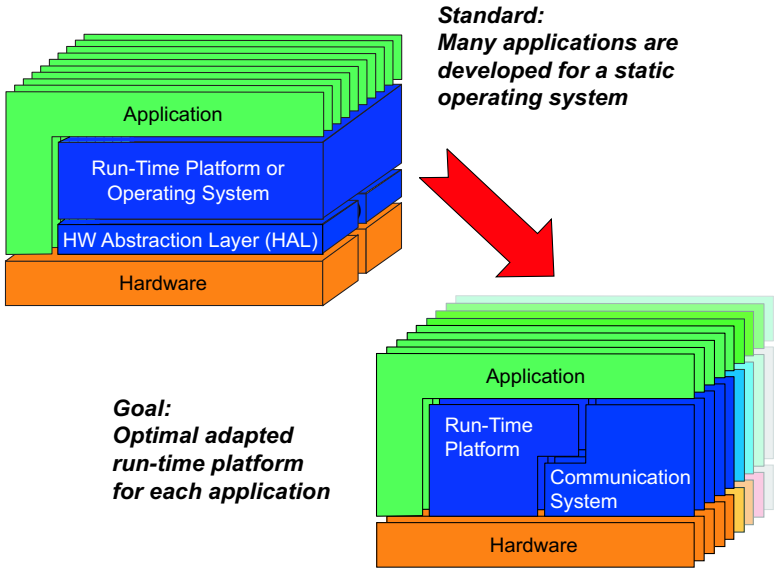

Fig. 2.2: From monolithic and static operating systems towards highly flexible and configurable ones.

There is a gap between the flexibility of the application's specification and the –more or less– monolithic operating and communication system's implementation with its kernel, modules, and drivers. This gap is often closed by implementing the operating and communication system as general as possible. This means that the operating system services are implemented for the general use case. This results into the well known general purpose operating systems, like Linux or Windows^{TM}. Their services are implemented in such a way, so that they can be used for a brought variety of (desktop) applications. This approach implies that a lot of operating system code is unused or behaves often very inefficiently. A lot of overhead is incorporated into the operating and communication system in order to handle every use case. Also, a lot of commercial real-time operating systems are implemented in such a way, so that they can run the most real-time applications (see Section 3.5.3$_{48}$). But instead of having application specific services, the application programmers must cope with the predefined ones. Often they suffer from missing features

of the operating system or they have to adapt the application to the functionalities that are offered by the communication and operating system.

Only at a very coarse-grained level modules can be integrated or removed. It is desirable for the development of embedded applications that the operating system can change the implementation, quality and quantity of its services in the same way as they are requested by all of the different applications or as they are offered by all of the various hardware platforms (see Figure 2.3).

For this reason it becomes more and more popular that operating systems for the embedded market are designed to be customisable. A lot of features can be parameterised, included or excluded. But exactly this job has nearly always to be done manually by the application programmer. According to the application's requirements the application programmer has to configure the operating system. But it is hard for the application programmer to identify which selection or option is the best choice.

Each application embodies specific features of an algorithm and –of course– behaves different. Some of these properties can help to implement the operating and communication system efficiently. A lot of operating and communication system services can be implemented in different ways. Whether the implementation is better or not depends on the application's use case. For example, consider the mutual exclusion of processes by a binary semaphore. The semaphore's implementation normally requests in the general case a waiting queue, which handles all blocked processes. If only two processes use the semaphore, then the waiting queue is superfluous. Only a pointer to the possibly waiting process is required. The implementation is much simpler. This shows that additional knowledge about the application scenario can improve the operating system's implementation.

This arises a general problem: The operating system expert knows in which way a certain service has to be implemented when a specific use case is applied. He can offer a lot of different implementations of the same general service for different use cases. On the other hand the application programmer or operating system user knows in which way the application behaves. But the application programmer normally does not know, which implementation fits best. There exists a gap between the huge number of possible application requirements and the customisable options of a flexible operating and communication system.

The demand to solve this gap between a highly flexible operating system and its optimal implementation according to a specific use case has been the motivation for creating TEReCS. It is exactly one primary goal of TEReCS to solve the following problem: *The operating and communication system should be as flexible as possible, but its flexibility should be transparent to the application programmer.* Note, that the TEReCS project does not want to develop a fine-grained customisable operating system. That means that a lot of different implementations for operating and communication system services already exist. It is the job of TEReCS to encapsulate the knowledge of the operating system expert in order to hide the configuration and customisation process from the application programmer. The application programmer should not deal with operating system services, which he does not want to use. He should also need not to consider the correct selection for a service implementation depending on the use case of the application. Instead, he must tell TEReCS now, which service is used in which way and when. This description defines the actual use case of the application. Seen from the operating system's point of view it de-

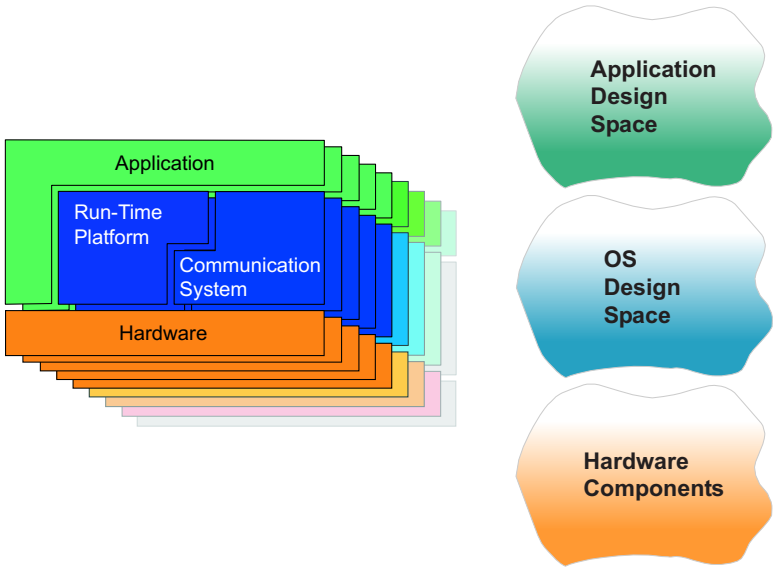

Fig. 2.3: Existence of design freedom for the application, operating system and hardware platform.

fines in which way the application behaves or wants to act. Depending on this behaviour TEReCS automatically selects the correct implementations of the required services, which fit best to that behaviour and produce the overall minimum overhead.

The strong demand for a flexible operating system with different implementations for its services arises another problem: A lot of operating system services assume that other specific services are present or behave in a special way. For this reason, a lot of operating system internal dependencies and requirements exist. In order to assemble a correct and most efficient operating system with minimal overhead, the application programmer must normally select the appropriate implementations of the services that are required for the final application. Additionally, he must also be an operating system expert, so that he can consider and solve these operating system internal dependencies. This mixture of external or application specific knowledge and operating system internal knowledge is strongly separated in TEReCS.

TEReCS requires a description of the application's behaviour, as well as a description of the hardware platform (see Figure 2.4). Additionally, TEReCS embodies the expert's knowledge about the operating system's internal dependencies. TEReCS tries to combine these three inputs in order to configure a final operating and communication system, which serves the application's demands at a minimum level very efficiently. Exactly this process of combining knowledge in order to generate the operating and communication system (see Figure 2.5) is a kind of knowledge transfer from the application further on to the operating and communication system. Thereby, TEReCS transforms the knowledge from the application's domain into knowledge for the operating and communication system's domain (see Section 6.7$_{132}$).

Nevertheless, when a system is nearly built automatically its correctness must be proven. The main focus of this work lies on the configuration of a communication platform for a distributed embedded application. The structural and functional correctness is assured by the configuration and assemble process. If the structure is wrong, then the assemble

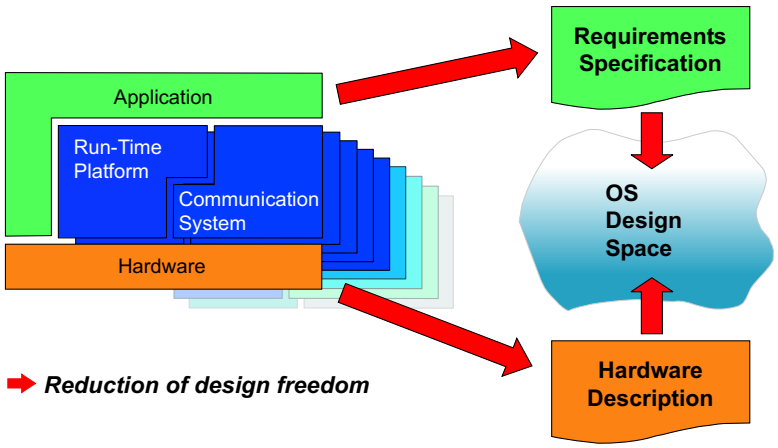

Fig. 2.4: The reduction of design freedom for the operating system by integrating requirements from the application and the underlying hardware.

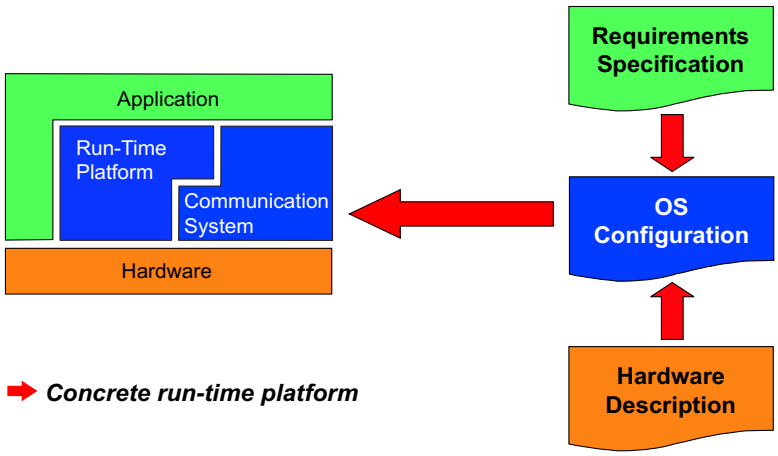

Fig. 2.5: The operating system configuration constraints the design space and completes the operating system description to a final instantiation.

(compile) process will fail. We assume that the design space description of the operating and communication system only permits the configuration of valid (in the sense of structural) and correct (in the sense of functionally working) instances. For embedded applications the real-time domain is often of special interest. The application need not only to produce correct results, but the results have also to be generated within specific time intervals (*deadlines*, see also Chapter 4_{61}). For this reason, the timeliness of an application has to be checked before the system is implemented. Thus, TEReCS contains an analysis tool (see Section 6.8_{133}) which checks, whether all tasks meet their deadlines. Here, the communications between the tasks are considered in detail. The design cycle of TEReCS specifies a loop. Within this loop a configuration is generated and checked as long as the check fails. This implies that the configuration has impact on the analysis and the analysis has impact on the configuration. The loop also ends when a new configuration can not be generated any more.

The concrete problem that is handled in this thesis will be described in the next section.

2.2 Problem

For each node of the interconnection network of a distributed target system an operating system has to be generated. *This thesis concentrates on the methodology for the configuration process of a customisable communication system and multiple customisable operating systems*. These operating systems must be able to execute the tasks of each node. They must also be able to provide the required communication services. The communication and operating systems have to be built from predefined atomic items, which must be *parameterised* and *connected* appropriately. The totality of all available atomic items represents the *component database* or *construction kit library* for the operating systems.

The task of TEReCS is to generate for each node of the target system a configuration description. These *configurations* describe which atomic items of the component database have to be integrated into the instance of the operating system of one node. Additionally, the configurations define all missing parameters for these items and they describe in which way the selected items have to be connected. The main objective is to generate a solution that is optimal in the sense of resource usage. The memory consumption and the punctual execution must just fit in the limits that are given by the system engineer.

In order to achieve this goal the application's requirements and behaviour, and also the hardware architecture and topology must be described by the system engineer (see Figure 2.5). TEReCS contains the design space description for all valid and correct operating system implementations. When the hardware platform contains redundant communication connections, TEReCS automatically tries to select only the "cheapest" connections for the routing of the messages that just fulfil the timing constraints.

Hand written code is still dominating for the implementation of embedded real-time applications. Often, operating system functions (like device drivers and protocols) are manually developed from scratch. This is done in order to have optimally adapted implementations that just meet the desired requirements and save as much resources (memory and processor time) as possible. *TEReCS tries to bridge from the application's specification to the automatic distributed implementation on top of configurable operating systems and an integrated configurable communication system.*

2.2.1 Assumptions and Ignored Issues

During the design process for distributed real-time applications occur obviously two main problems, which are not covered by TEReCS. First, the load balancing of the tasks onto the processors is assumed to be done by the engineer or a third-party tool. Second, the estimation of the average or worst-case execution times of the tasks and also the estimation of the operating system overhead is assumed to be possible with other tools (like CHaRy [1]). TEReCS assumes that the mapping of the tasks onto the processors and the worst-case execution times of the tasks and the operating system services are known a priori.

TEReCS wants to know from the application programmer, which tasks run on which processors. It wants to known in addition their periods, their worst-case execution times, and their deadlines. It also wants to know which operating system services are called by each task. For the timing analysis it is also desirable to know, when these system calls

occur relatively to the start time of the task. For the definition of these relative start times all operating system overhead of previous calls is assumed to be zero.

Because TEReCS concentrates on the communication system, the system calls that send or receive messages have to be marked explicitly. Additionally, the corresponding `send()` and `receive()` of two tasks have to be connected. This connection is assumed to be an uni-directional message transfer. The maximum size of the message and the period of the transfer (*minimum message inter-arrival time*) must also be given. Optionally, the data structure of the message can be declared. If this is given, TEReCS can automatically integrate protocol code (when the operating system supports this) in order to transform the endianess of the data for heterogeneous hardware architectures. Also optional protocol specifications for this communication connection can be specified. These include requests for error detection and correction, acknowledging, a synchronous send (*rendezvous*), the receiver buffer capacity (to be there or not), a list of hardware links not to be used (this option is internally required between the configuration and analysis phase), and the port object's name that is used for the send and receive call.

TEReCS also assumes that the system behaves statically. This means that tasks do not occur sporadically and are not created and terminated dynamically on demand. Such tasks can be modelled in TEReCS only when these sporadic tasks are assumed to be always present; thus resources are wasted. The tasks and the communications are assumed to be periodic. Aperiodic or sporadic ones must have assigned a minimum period.

TEReCS was designed especially for the generation of configurations for inter-node communicating operating systems. It is completely independent of the target operating system to be used. The only requirements a target operating system (OS) should fulfil are:

- The OS must be configurable
- The OS should support various communication protocols and devices
- The size of the objects to be configured should be of the same granularity
- The OS should be tunable at a fine-grained level (but this is not a must)
- The configuration objects are atomic
- The objects can be distinguished by type, costs, and their required sub-relationship dependencies to other objects
- The objects must be parameterised and their dependencies to other objects are to be configured
- The configuration decisions can be mapped to the selection of concrete choices out of alternative objects
- The scope of the configuration should be the selection of the objects per processor and their inter-relationships
- In particular, the objective of the configuration is to build a service platform for the application from these objects and to map the communications onto services/protocols of the operating system, real existing hardware devices, and media (resource reservation/routing) by producing minimal costs and providing a timeliness execution

2.3 Extensible μ-Kernels versus Configurable Library Operating Systems

Operating systems can be distinguished into two main categories. The first category comprises all operating systems that integrate a *kernel*. The kernel embodies all basic services of the operating system. These are mainly the resource management functions like the dispatching of tasks to the processor, their synchronisation, and the communication. Often, the code of the kernel is executed in a more privileged level of the processor in order to maintain specific registers, which are normally protected against the application code. For this reason, the application requests for operating system functions by *traps* or *system calls*. These mechanisms change the privilege level and enter the kernel. The kernel also often maintains its own memory area, which cannot be accessed by the application tasks. This assumes that a *memory management unit* (MMU) exists, which can be programmed accordingly. Kernels are distinguished into $monolithic$, μ-, $nano$-, $femto$-, and even $pico$-kernels by the amount and quality of services they offer. Additional services can often be implemented by operating system tasks, which run outside the kernel and behave like servers.

For configuration purposes such kernel-like operating systems support for *modules* and *demons* that can optionally be integrated. Modules are services that are directly integrated into the kernel during compile-time or loaded during run-time (see page 48ff.). Demons are the operating system's server tasks that run in the background of the applications. They can be started or terminated on demand. Another configuration option for this category of operating systems is to change the complete kernel. This can mainly be done during the compile-time or the load-time of the kernel. The configuration options of kernel-like operating systems are often limited and act on a very coarse-grained level.

The other category of operating systems implement their services in a code library that is linked directly to the application code. They miss a kernel and mainly act as a service platform. Often, such systems are called *run-time platform* instead of operating system. Their operating system services are simply requested by a function call. Examples for such systems are the CTools for Transputers [65] or DREAMS. DREAMS was developed especially for the purpose to support fine-grained customisation facilities of the run-time platform. Therefore, DREAMS is configurable in the source code before compile-time at the object level (for details see page 51 and Section 6.6.1.1$_{127}$).

DREAMS is used for TEReCS as the target system under consideration. But DREAMS is used in TEReCS only as a demonstrator [15]. TEReCS is flexible in the way that it can be adapted to nearly every configurable operating system, which is compliant to the properties that are mentioned on page 13. Its input language is designed to fit for the general case and also its output language can be adjusted to the required format.

Because the aspect of a distributed application and the communication dependencies are in the main focus of TEReCS, the next section will clarify what is meant by the distinction of *operating system* and *communication system*.

2.4 Operating System versus Communication System Configuration

Generally, the communication services are an integral part of the operating system of one computer. In this case often only the device drivers for the communication links and the

implemented protocol stacks are considered. The operating system is seen from a local point of view. The local system is the main service and execution platform, which supports for communication to externally located services on (sometimes dedicated) servers. Also the applications are local to one node. Sometimes they act as clients and require communication to servers.

But in the embedded world a distributed system often implements also a distributed application. The interconnected network of processor nodes serves as one virtual machine. Here, the global view onto the whole distributed and parallel working system is preferred. The client/server aspect plays a minor role. The distributed tasks often have an equal status. The communication dependencies between the tasks are immanent and very important for achieving the final goal. The interconnection network between the processor nodes and the operating system services on each node represent the communication system. In the real-time domain not only the task scheduling on one processor, but also the global scheduling on all processors, as well as the scheduling of the messages on the links have to be considered and analysed in order to assure a temporal correct behaviour. The view to an operating system, which has a communication system integrated, can completely be inverted: Here, the communication system of the virtual distributed machine has many local operating system instances integrated on each processor node.

TEReCS uses the second global view. The main focus of TEReCS lies on the generation of optimally adapted local operating system instances. The communication dependencies between the distributed tasks are mainly considered. Moreover, it is the main motivation for TEReCS to support an optimal and temporal correct communication inside the distributed virtual machine. The communication aspect mainly drives the configuration and analysis in TEReCS. This is the reason why in this thesis the words *operating system* and *communication system* are often used in conjunction or are explicitly be distinguished.

2.5 Chapter Outline

The following list provides a closer view on the organisation and intentions of each chapter.

Chapter 1 starts with a summary.

Chapter 2 contains an introductive overview over the topic and gives the motivation for this thesis. The problem, which TEReCS tries to solve, is explained herein.

Chapter 3 refers to “Configuration” in general. In that chapter the general theory about configuration is presented. Additionally, it gives a short overview about existing configuration systems, as well as an overview about configurable operating systems.

Chapter 4 presents a general and brief overview about the state of the art in “Real-time Analysis”. This chapter presents a reasonable good introduction to the schedulability theory of real-time systems. It is important to have this overview in order to understand the problem, which TEReCS has to solve in this domain. The theory of process scheduling is extended by approaches that handle communication aspects.

Chapter 5 tries to give an answer for the question: How is it possible to come "from a taxonomy" for operating systems "towards their configuration space"? Such a complete design space description of the operating system under consideration is required by our configuration tool TEReCS. The idea of the "Puppet Configuration" process in TEReCS is developed in this chapter.

Chapter 6 is mainly dedicated to the herein developed tool suite TEReCS. A model and method is presented for an automatic configuration process (see Section 6.3 and Section 6.4). In this model the detailed information about the structural properties of the configurable communication and operating system are hidden from the user. The configuration process is extended by a hierarchy concept in Section 6.5. The languages for the inputs of TEReCS' configurator and analyser are briefly described in Section 6.6. Hereafter, the knowledge transfer from the application to TEReCS in order to configure the operating systems and the communication system is explained (see Section 6.7). In Section 6.8 the concept of the real-time analysis in TEReCS is presented. The chapter ends with some examples that show the mutual influence of the configuration (see Section 6.9) and analysis phase (see Section 6.10) in TEReCS.

Chapter 7 presents briefly some results that can be achieved by using TEReCS' methodology.

Chapter 8 concludes this thesis with an outlook.

The ***Appendix*** refers to a simple configuration example, as well as one for the timing analysis. A description of the input languages for TEReCS is also given.

Each main chapter ends with a summary that describes the contribution of that chapter to this thesis.

2.6 Hints for Reading

The Chapter 3 and the Chapter 4 give an introductive overview about *configuration* and *real-time analysis*. Readers who are familiar with these topics can skip the chapters. The Chapter 2, Chapter 5, Chapter 6, and Chapter 7 should be read in the given order.

The thesis also addresses topics in the fields of operating systems, embedded systems, real-time control, communication, and object-oriented design. It is assumed that the reader is familiar with the basic concepts and terms in these areas.

The index register can be consulted to expand abbreviations that are used. Some index entries may be of special interest:

Operating system contains a list of presented operating systems.

TEReCS contains a list of all explicitly defined new terms concerning TEReCS.

Configuration enumerates several concepts for this topic

Configuration system lists the existing configuration systems that are described in this thesis.

To ease locating references to figures, tables, and subsections, far away references are indexed by the corresponding page number, like Section 6.9$_{142}$.

The Appendix A and the Appendix B give two simple examples for the inputs and outputs to the configurator TGEN and the timing analyser TANA, respectively.

CHAPTER 3

Configuration

> A configuration system is an expert system which helps to assemble components into an aggregate according to some goal specification and using expert knowledge.
>
> Bernhard Neumann, 1988 [91, p. 4]

This chapter wants to introduce the theory of configuration and, additionally, some practical systems are presented. At the beginning the aims and the meaning of configuration are clarified. The overall goal of configuration in different application domains is described. Then, a detailed presentation of the methodologies that can be used for configuration are presented. The different approaches are classified after their representation models and their algorithms. The inputs and outputs to the configuration algorithms are described in general. A survey of existing configuration systems completes this chapter. Of coarse, an overview about the advantages and aspects of configuration for operating systems and for communication systems follow. Herein, also some already existing operating systems that make use of configuration are presented.

At the end of the chapter the reader will understand the problem that configuration solves and he will have an overview about the different approaches that can be used to achieve this goal. He will also have an idea of the aspects where configuration can help to construct operating and communication systems for special use cases. The first section of this chapter starts with an overview about the aims in general, why configuration is used instead of other techniques for building a final tailor-made system.

3.1 Aims of Configuration

Meet Requirements

The overall aim why a system or a component is designed to be configurable is the strong desire to save costs in realising the final system. The scope of costs may comprise of space, weight, execution time, power, energy consumption or some other resource needs.

Moreover, it is intended that the system's properties meet as closely as possible the requirements of an application or an environment. This means a final *configuration* of the system measured in terms of a cost function should fit best and should match all structural, functional, resource and timing requirements.

Reuse Components

Additionally, a configurable system can support reuse of its components for different implementations. All similar implementations serve the general purpose, where the system exists for, but can be distinguished in their internal structure, functionality, costs and *quality of service* (QoS) they provide. This is based on the idea that different requirements should be served by different (optimally adapted) systems and, therefore, save resources (or costs) for unused features.

Assemble System

The requirements that a final system has to comply are specified by a *requirements specification*. In a formal way all requested final properties that the system has to fulfil are described herein. It is the task of the *configuration process* to assemble the correct (and minimal) set of components with the correct arrangement that has these properties (and serves for the required functionalities) at minimal costs.

Guide User

Often the requirements specification evolves dynamically during the configuration process. Thus, the configurator interactively asks for certain properties the final system should have. In this case it is the task of the configurator to guide the users through the process. Only valid choices and actually valid property options are presented.

Assure Correctness

Additionally, the configurator takes care of the correctness, i.e. the final system works and behaves in the appropriate and desired way. This includes that the configurator is aware of structural inter-component relationships such as *"a component A can only work, if another component B is present"* (implication). Such relationships, constraints, or rules are specified within a database. This database serves as a *knowledge base*, which stores all domain specific knowledge about the dependencies and properties of the system's components. Within the algorithm of the configuration process the (domain's) *control knowledge* is stored, which defines how a final configuration can be derived by selecting appropriate components and fulfilling the requirements specification (see Figure 3.1).

Automatic Configuration

A configuration algorithm that dynamically requires user interaction is called *interactive configuration process*. On the other hand, a requirements specification can be provided as static input to the *configuration tool* before the configuration process starts. If the configurator requires no other user interaction in order to create the configuration of the final system, it is considered to be an *automatic configuration process*.

Hide Expert Knowledge

In order to support for a highly flexible system, that can be adapted to certain requirements, a component library with customisation features and the *expert knowledge* about the assembly process is required. This expert knowledge comprises the static information about the component's interrelationships and customisable properties, as well as the control knowledge of the process. A configuration tool can hide this expert knowledge – which includes a lot of implementation details of a system – from a user (who wants to use a finally configured system with certain properties).

Provide System Abstraction

In this way configuration provides an abstraction of the system. Thus, just as operating systems provide a virtual machine (hiding all information of the concrete hardware) by its API, the configuration tool provides a virtual view to the system by its requirements specification. Nevertheless, the idea of supporting different resources can be supported by the configuration tool. The system itself and the application can be planned inde-

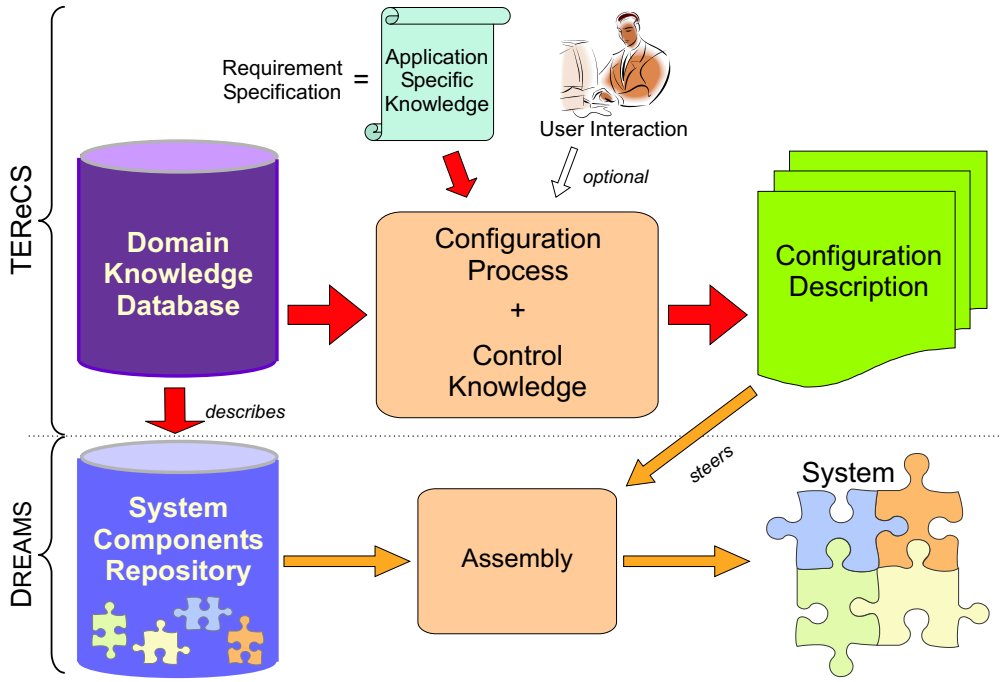

Fig. 3.1: Input/output flow for the configuration of technical systems.

pendently from the present resources. If the system's component library supports for different types and amount of resources, then only those components will be considered while configuring, which satisfies the given resource requirements. Or seen from the bottom: Only for the present resources the appropriate components are integrated into a final configuration (if required).

This implies that there exists another specification, which describes the given or existing resources and constraints that the configuration can use or must satisfy. Obviously, this specification can be empty. But, often it exists and is also a part of the requirements specification.

Exploit Application-internal Knowledge

Previously, it was mentioned that the requirements specification defines the properties a final configured system must fulfil and that it can be seen as an abstraction. But where do these properties/abstractions come from? They are derived from the application. *Internal knowledge* about the application forms the basis for the requirements specification. In this way, the configurator exploits the internal knowledge from the application for generating an optimal (resource and cost minimal) system. The configuration algorithm –or more specifically: the control knowledge– defines a *knowledge transfer* from the application to the system under consideration (see Section 6.7$_{132}$).

In a general sense, application's configuration is used to support users to create newer and fully specified derivatives from customisable systems. Mainly three fields can be identified, in which configuration is exploited to its fullest extent. These are *"Technical Systems"*, *"Software Management"* and *"Software Synthesis"*. Whereas research concentrates on the more natural aspect of configuration in the field of *"Technical Systems"*, commercial companies develop a huge variety of products in the area of *"Software Management"*. In *"Software Management"* configuration is principally used to support soft-

ware developer teams to handle big software projects and all of their software fragments. *"Technical Systems"* in fact mainly drove the scientists to create a theory about configuration. Mostly, configurators for *"Technical Systems"* have been created for business services in order to support sales managers during the creation of (correct) sales offers for customers. Configuration for *"Software Synthesis"* is a quite new research topic. It is inspired by configuration for *"Technical Systems"*. The configurator TGEN for TEReCS, which is developed in this thesis, belongs to this area.

In the next three sections, these three areas are briefly investigated. We present, how they can exploit configuration for specific purposes. Examples for configuration systems are given later on in Section 3.4$_{41}$ after the basic ideas and principles have been presented.

3.1.1 Technical Systems

The primary application domain for configuration systems are technical systems. A technical system often consists of a variety of parts. Each part provides for some properties or facilities in order to yield the overall objective. Such properties or facilities can be the component's spatial design, weight, dimension, energy conversion mechanism (e.g. from electrical power to a physical force), melting point, thermal conductivity, or other physical properties. The spatial arrangement of the parts and their connections define naturally a component structure or hierarchy. Moreover, energy or mass flows and forces can be observed between their physical connections, which naturally define interrelationships and dependencies among the components.

Furthermore, it is the engineer's natural view to build complex systems by arranging components. This procedure is known as *constructing*. If the components are pre-defined and are not to be created from scratch, then this procedure becomes identical to the configuration problem.

In fact, technical systems and the manner of their construction have been the reason why the theory of configuration had been developed. The engineer's desire to be supported by a software tool drove the automation of the construction process. The engineers desire a tool, which takes care of constraints, with respect to the correct arrangement of components (by checking the dependencies). And consequently, the engineer will be replaced by such a tool, when the expert's knowledge is completely integrated.

Many examples of this kind can be found. Here, the pioneer tool XCON (see page 41), which configures computer systems at DEC, AKON (see page 41), which configures telephone switchboards at Bosch/Telenorma, and $^{\mathrm{art}}deco$ (see page 44), which configures hydraulic circuits and had been developed at the University of Paderborn, should be mentioned.

3.1.2 Software Management

> Configuration management is the key to managing and controlling the highly complex software projects being developed today.
>
> Clive Burrows, 1999 [24, p. 1]

Most *Software Configuration Management* (SCM) systems are *version-oriented*, where each component exists in several versions organised as *variants* (alternative versions) or *revisions* (consecutive versions). But *Configuration Management* (CM) tools have developed from simple version-control systems targeted for individual developers into systems capable of managing developments by large teams operating at multiple sites around the world. The key capabilities of SCM tools are the identification and control of software and software-related components as they change over time. The key features supported by SCM tools are:

Parallel Working. Originally, SCM systems' primary task is to prevent several users (programmers) from attempting to change the same component (piece of software code) at the same time.

Change Management. Version control maintains a history of the changes to a component as it evolves over time and allows users to access a particular version – not just the last version created. Moreover, the problem of tracking, change control, the presentation, and the analysis of management information derived from these sources are issues of SCM.

Build and Release Support. An intelligent creation process can reduce build times dramatically by reusing partially configured items from previous implementations.

Process Management. Many users, particularly those seeking an external quality approval such as ISO 9000 or from a particular Software Engineering Institute have standard development processes, which they expect their development teams to follow. The process management features in SCM tools allow the developer to ensure that components progress through chosen lifecycle phases before being released.

Web Management. SCM support for Web and particularly Intranet pages and their embedded objects.

Documentation Support. It is a big challenge to support for documentation of the decisions and work done on software projects resulting into different versions and branches. Document management systems offer facilities to handle large and complex systems and to support users to retrieve special documents, their dependencies and descriptions from a repository.

In contrast to the configuration problem in this thesis, in SCM the component set, which makes up the product, is often predefined, e.g. in a makefile. The central configuration problem there is to select a suitable version for each of the components. Also a fundamental difference is that in SCM a component occurs only once in the configuration; in a generic configuration it can be repeatedly included.

If variants are regarded as components the two problems become more similar. Nevertheless, in SCM it is easier to select appropriate variants due to an existing version or release numbering (versions must match).

It is the main concern of SCM to support the component development, whereas in configuration it is the main task to search for a valid and appropriate component aggregation.

Because there is a strong need for SCM for commercial and high quality software development of large systems, research and the market concentrate on this area. Therefore, SCM defines its own research field and its community is organising well known workshops (like the *SCM* workshops associated with the IEEE ICSE conferences; though in their own separate proceedings, or the *Software Technology Conference* (STC)). An overview of SCM systems is given in [29]. A nice introduction and tutorial about problems in SCM – including a sophisticated comparism and classification of SCM systems – is given by Conradi et al. [30]. A WWW starting point for references and links related to SCM are Brad Appleton's "Assembling Configuration Management Environments" (ACME) project pages [3].

3.1.3 Software Synthesis

In this thesis configuration is used for software synthesis. In more detail, the code for a distributed run-time system supporting real-time communication can be generated out of software components. The set of required components, their arrangement and their parameterisation have to be calculated. In fact, no code (in terms of a programming languages like C or C++) has to be written, but a configurator has to decide how and which code blocks it has to select and arrange. Even a programmer has to decide – at a relative low level – which statements he chooses and he arranges in which manner. Thus, it exists also a language SCL [42] for the operating system library kit DREAMS, in which a configuration for the operating system can be described. Such a description is the output of the configurator for each computer node of a distributed system on top of which DREAMS should run. The DREAMS toolkit then produces the run-time code of the operating system for each node. While finalising the operating system source code, its under-specification is removed by incorporating the configuration description. After this, the code can be compiled.

What is generated or synthesised by TEReCS is the SCL description for a DREAMS operating system for every node of a distributed system. The main issue considered while doing this is that the system runs under real-time restrictions and, therefore, the communications underlie some timing restrictions. This means that the distributed operating systems must be able to communicate with each other and the time spent for a communication (end-to-end delay) must not exceed a certain deadline. These constraints are considered while the configurations are generated. TEReCS supports the reuse of software and considers timing constraints during early stages of the system creation.

3.1.3.1 State of the Art

Software synthesis for embedded applications is not a new approach. Commercial software synthesis products concentrate on design-tools for the development of prototypes

or evaluation examples, like STATEMATE [64] for *StateCharts* [60], or tools like ASCET-SD [45]. In most of these tools the automatic code generation for the pure application is of key concern. Unfortunately, the application often makes use of a more or less static application management system or operating system (like ERCOS [45] for ASCET-SD™ or VXWORKS™ [148] for STATEMATE). The generated code serves only as a starting point for developing effective production code. An operating system supports multitasking, interrupt management, IO-interfacing and communication in a very overhead-prone manner to satisfy all the general cases. In [19, 43] it is shown, that a customised and tailor-made application management system has major advantages regarding performance and resource demands in contrast to a conventional μ-kernel operating system.

STATEMATE
ASCET-SD

In the research field different approaches exist towards automatic generation of embedded real-time applications. The SFB 501[1] [9, 10] deals with the synthesis of systems out of ready-made components. The overall goal is to create the application and its tailor-made runtime-platform out of components. The view of a system is a collection of customised components. A more narrowed down local sight on the whole system is followed in this SFB. Each component is developed and described by a so-called value-added interface. This interface describes the component by spreading a design space representing all possible design decisions. The dimensions of the design space represent the categories of customisable parameters of the component. Dependencies between components are regarded as correlations between different categories. Thereby, a component can also demand another component (or a set of components).

SFB 501

Goossens et al. [31, 138, 139] describe a technique for finding a suitable schedule of competing threads of an application by a so-called MULTI-THREADED GRAPH CLUSTERING. Their work is representative for all approaches using graphs for creating a proper solution. Their overall aim is the software synthesis concentrating on applications designed to run on a single processor. The solution is obtained using a *Constraint Graph* [75] whose vertices are threads of the application and the edges describe the precedence relation and timing constraints between the threads. They concentrate on finding a feasible schedule between threads by clustering threads to disjunctive thread frames. The approach is based on merging threads (vertices) to a maximal cluster opposite to searching paths in the graph. A combination of static off-line scheduling and dynamic scheduling at runtime is used.

MULTI-THREADED GRAPH CLUSTERING

The AMPHION [86] is a domain-oriented design environment (DODE). It is based on a formal specification and a deductive programme synthesis. The reuse of problem specifications –not of programmes– is central for domain-oriented *knowledge-based software engineering* (KBSE). For users without background in formal mathematics, developing a formal problem specification seems to be very difficult. Therefore, AMPHION incorporates techniques from visual programming in a *graphical user interface* (GUI). The domain is specified by defining a domain theory and some theorem-proving tactics. An interface compiler automatically generates user interface tables for the GUI from the domain theory. A theorem prover (SNARK) uses the domain theory and the theorem-proving tactics to generate an application programme. SNARK uses first order logic. A specification checker gives the users aid in developing the specification. The check is done by trying to solve an abstracted problem. If it cannot be solved, then it employs heuristics to localize the problem in the specification and to give the user an appropriate feedback. The general textual specification looks like:

AMPHION

[1] Collaborative research centre 501 of the German Research Foundation (DFG)

```
lambda (inputs)
  find (outputs)
    exists (intermediates)
      conjunct1 & ... & conjunctN
```

where `conjunct` is either a constraint, like $P(v_1, \ldots, v_m)$, or an equality defining a variable through a function, like $v_k = f(v_1, \ldots, v_m)$. The substitutions for the output variables are constrained to be terms in the applicative target language, whose function symbols correspond to the subroutines in a library. The domain theory consists of an abstract theory, a concrete theory and an implementation relation between them. The abstract theory needs enough semantics to derive an abstract solution consisting of abstract operations from an abstract specification. The implementation relation is used to generate a concrete solution, taking into account subroutine pre-conditions.

3.2 Configuration vs. Customisation and Adaptation

The *Online Dictionary of Library and Information Science* [106] describes "configuration" as follows:

> *"**Configuration**: The physical arrangement and functional relationships of the various components of a computer system, usually established to meet the needs and preferences of its users. The term configurability has been coined to refer to the ease with which a computer system can be modified or customised to meet changing needs and requirements."*

A *configuration* describes how a complete system has to be built out of a set of components. It defines which components are part of the system as well as their arrangement, e.g. structure of the system and the component's relationships. Additionally, properties of the system's components are defined. In this scope a system can be a technical system (e.g. the hardware of a computer system) or a software system (e.g. a program or system software like a middleware or the operating system). The process of creating such a configuration is called *"to configure"*. When a system's aggregation of its components or their properties can be changed retaining its correctness (i.e. the system still works and fulfils its general purpose) the system is said to be *configurable*.

More general in its meaning is the term *customisation*. A system, a component or an object is said to be *customisable*, if some of its properties can be set to specific values out of a given set of values. For these properties alternative instantiations exist for their final values. These properties can include also structural or taxonomical information of the item. Formally the description of a customisable system results into an *under-specification* of the system.

The *configuration* of a customisable system describes a final result where each set of alternatives have been pruned to one selection, e.g. all under-specifications were removed.

The term *configuration* is mostly used for the static case, which means, a configuration will not change during the lifetime of the customisable item. Whereas it is said to be *adaptable*, if the configuration can change dynamically even during the lifetime of the item.

3.3 Methodologies

> The meta-goal of design is to transform requirements, generally termed functions, which embody the expectations of the purpose of the resulting artifact, into design descriptions.
>
> John S. Gero, 1990 [48, p. 26]

The overall goal of this work is to support for a method to design and create in a very fast and optimal way the software for the operating and especially communicating system for a distributed embedded real-time application. In order to achieve this we have to describe the process, the requirements or inputs to the process, and the output, which describes the resulting end-design. Even more, we have to formulate some constraints or properties, which have to be considered during the design phase. Obviously, the domain for which a system should be designed has great impact on these points. Nevertheless, we can identify general principles that are universally valid. It is the purpose of this section to structure and to categorise these basic principles. Designers always use a lot of knowledge about their domain – independently if this is in arts, architecture, construction, engineering or even a domain like economics or literature – in order to create, respectively design a product, system, or process. Here, we would like to concentrate on design methods, where this knowledge – often named *expert knowledge* – is formally described and, therefore, is part of the process itself. Often – especially in arts – it is hardly possible to describe this expert knowledge formally. But, if it is possible, one can benefit from this in many respects.

One big advantage one can exploit then is that one can make reuse of the expert knowledge very often and very fast. Even more, the process for the creation of a design can be – more or less – automated. This implies that the overall design process can be accelerated. Additionally, it follows that the time spent for the design process and consequently the quality of a product with respect to the time-to-market aspect can be improved.

Even more, if the amount and granularity of the atomic items or parameters, which have to be arranged during the design, can be limited and are of a certain complexity, then the *design space*, which spans the set of all possible solutions or arrangements, becomes smaller. This is well known in the design of component systems, where the complete system is designed out of components, which embodies special and well defined solutions for a complete subproblem. For example, it makes a difference to create a programme from scratch in a language like C or to assemble it from Java™ objects. Thus, the reuse of already made components reduces the complexity of the design process. Hence, it becomes possible at all to formally describe the design space and more over, the correct solutions or the valid paths inside that design space.

B. Stein et al., 1998 [131, p. 1] say

> *"The purpose of a design process is the transformation of a complex set of functionalities D (= demands) into a design description C (= configuration)."*

They generally state that a design process has inputs and outputs. The inputs are demands or requirements and the output is the final design description. This design description can generally be seen as a configuration. But in fact a configuration is a special

form for an output, where the final design is an arrangement of components, which may have been enriched by some parameters. The key point is that a configuration determines the selection of the components from a pool and determines their relationships (and selected amount).

For a (semi-)automatic design process it is essential that it has – besides requirements – a second type of input. This is a description of the pool of available components (library) and additionally a description of the possible relationships or properties or underlying conceptual structure of the components. These descriptions are seen as the expert knowledge that is required for the configuration process. Sometimes this expert knowledge also contains a description of all valid configurations or some restrictions or constraints, which describe what can be configured and what cannot be. All of this information can be seen as the *general domain knowledge* including physical laws or industrial standards. It is valid for all problems of an application domain. On the other hand the requirement or demand input to the configuration process is called *problem-specific domain knowledge*. It helps to restrict the design space. Because sometimes these data are accumulated in course of the problem solving process, they are sometimes called *dynamic*. Whereas the design process itself or in particular the configuration algorithm represents knowledge about the control of how to achieve a valid configuration. This is sometimes named *problem-solving domain knowledge* [91].

The idea of configuring a design is that the overall design process is delayed. In fact, it is split into two phases: The first is the creation of the knowledge base (containing the general domain knowledge) and the definition of the problem-solving domain knowledge in form of an algorithm. This is done by a domain expert. In the second phase, which is problem specific, the domain knowledge (requirements) must be formulated and the configuration algorithm is used to find a solution in terms of a configuration. This second phase, named *configuring*, is done more often for a huge variety of problems. Whereas the first step is done only once and an expert often makes only updates.

While the expert specifies the knowledge base and the algorithm, he always (more or less) has in mind all possible problems and their solutions. Hence, he describes a solution in a very general way. More in detail, his description of the general domain knowledge is a solution to a generalised problem, which has to be extended for a problem specific solution. Thus, the general domain knowledge is an *under-specification* of a solution. The possible solutions of one missing specification are limited and can often be enumerated. (Explicitly in the description or implicitly in the algorithm.) Therefore, the expert makes use of non-determinism for the specification of all solutions. This non-determinism spans the design space. For a final solution it has to be removed by selecting one specific choice for a missing specification. The under specification is identified and removed by incorporating the problem specific knowledge into the general domain knowledge.

But it is essential for configuration problems, that a lot of solutions exist, which have different costs. What a configuration algorithm in fact does is searching for a cost optimal solution. This brings up two main peculiarities of configuration: The algorithm defines a **search** and evaluates the **costs** of a possibly found (structural or functional correct) solution. Hence, configuration can be seen as a combination of search and optimisation strategies.

3.3.1 Searching in Graphs

To configure means to search.

Benno Stein, 1997 [127, p. 15]

In this and the following subsections a brief overview about the search theory in graphs and cost functions is described. This is done, because the algorithm in TEReCS (see Chapter 6_{105}) is strongly dependent on searching in graphs. We will concentrate only on aspects that are important for configuration. Most of the theory is taken from [128, 97, 94]. In the following subsections a possible categorisation of approaches, which are used to implement the search and the optimisation, are given.

If a solution for a problem cannot be directly derived, then it will be split into sub-problems (divide-and-conquer), which have to be solved. The sub-problems can also be non-trivial, that means they will be split into sub-problems in the same way, etc. This recursive algorithm spans a tree, which is directed from a problem to its sub-problems. Its starting node (root) represents the original problem and its terminal nodes (or leaves) define directly solvable or trivial problems (with their solutions attached). The complete, so-called, *solution graph* must be known for solving the original problem of the starting node.

In the previously defined solution graph all successor problems of a node must be expanded and respectively visited. The edges to successors form an AND-relation. But it is also possible to consider these edges as alternatives leading to any of two or more possible problems that either have to be solved. These edges will form an OR-relation. Not every sub-problem must be solvable, leading to dead terminal nodes without having a solution. In this *problem graph* a path from the starting node to any solved terminal node defines a solution.

Normally, these directed graphs do not contain cycles. Often, they are trees. A combination of these two graphs exists, which is called *AND/OR-tree*. The edges to successors can be in an AND- or in an OR-relation. This means that some of the successor nodes must all be solved and some have to be solved alternatively. There exists a *canonical form* in which *all* successors of a node form either an AND-relation or an OR-relation. The general form can be transformed to the canonical form by introducing new sub-nodes (see Figure 3.2). In the canonical form a node that has only successors being in an AND-relation is named AND-node, whereas a node having only successors being in an OR-relation is named OR-node. The solution in a canonical AND/OR-tree G is defined by a sub-graph G', where

- the starting node $s \in G'$
- $\forall\, n \in G'$ and n is an AND-node: $\forall\, n'$ is successor of n in $G \Rightarrow n' \in G'$
- $\forall\, n \in G'$ and n is an OR-node: $\exists$ exactly one n' that is a successor of n in $G \Rightarrow n' \in G'$
- all terminal nodes $n \in G'$ define solved problems

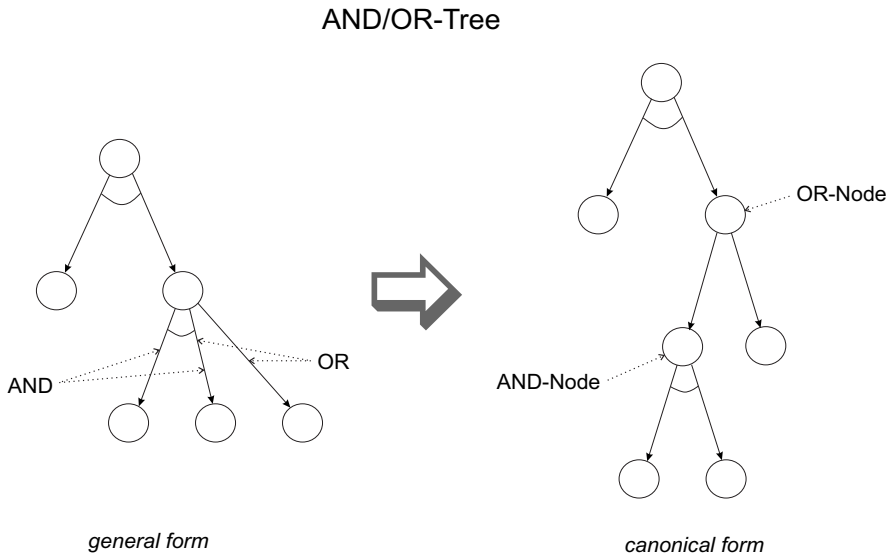

Fig. 3.2: Example for a transformation of a general AND/OR-tree to its canonical form.

G' is then the *solution graph*. A *solution-base graph* G'' is defined nearly the same way, except that each terminal node must possibly be solvable (i.e. is not leading only to dead nodes).

For example, in the game theory special forms of AND/OR problem graphs are well known. Each level of the problem tree defines sub-problems or reaction-problems alternatively. For instance, in a chess game possible moves define sub-problems and possible reactions of the adversary define reaction problems. A level of sub-problems contains only OR-nodes whereas a level of reaction problems contains AND-problems only. This kind of tree is called *problem reduction graph* (see Figure 3.3).

A canonical AND/OR-tree can be viewed as a *hyper-graph*. In a hyper-graph *hyper edge* are allowed to connect a set of nodes. If the hyper-graph of an AND/OR-tree is considered, where all outgoing edges of an AND-node form hyper-edges, then the solution graph of the AND/OR-tree can be transformed into a solution path in that hyper-graph.

A solution in the solution space of a problem graph or an AND/OR-tree is defined by a path or a sub-graph, respectively. In order to find the solution the graph must be traversed. Additionally, multiple solutions may exist. The problem is to find a solution, which is optimal in terms of costs for the solution. Thus, an optimisation problem can inherently be given.

The configuration problem can also be modelled as an AND/OR-tree or -directed acyclic graph (and this model is used in this thesis). In the next few paragraphs we discuss search algorithms to find solutions in the graphs. In order to find a solution the graph must be traversed starting from the root node to its leaves. Thereby, it has to be decided, which node of all the successor nodes of the actual visited node has to be visited next. This procedure of scheduling the sequence is called "*expansion*" of the node. All expanded, but not visited nodes, have to be stored in an appropriate order into a so called *OPEN-list*. All already visited and fully expanded nodes are stored in a *CLOSED-list* managed

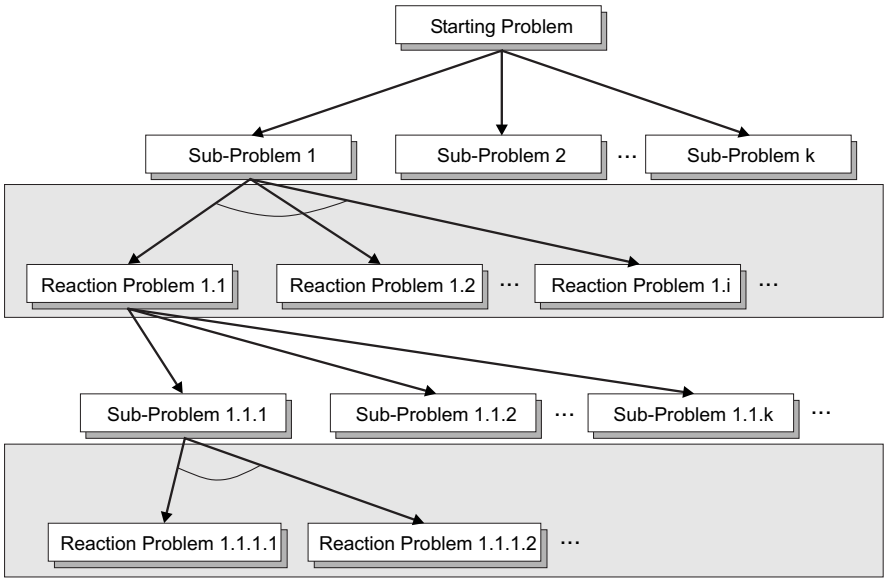

Fig. 3.3: Example for a problem reduction graph (see [128, p. 8]).

in FIFO[2]-strategy as in a queue. Hence, the *CLOSED*-list stores the traversal path through the tree.

The graph traversal algorithms can be generally distinguished by the sequence in which nodes are expanded. They are called to be *blind* or *uninformed*, if the sequence only depends on information of previously visited nodes. They are called *informed* (guided / directed), if the sequence depends additionally on information about the goal or on general domain knowledge.

Two well-known representatives for uninformed traversal algorithms are *depth-first-search* and *breadth-first-search*. In *depth-first-search* (DFS) the *OPEN*-list is handled in LIFO[3]-strategy and, hence, it is organised as a stack. This means that first all nodes directly leading to the first (or left most) leaf are visited, then the nodes leading to the second leaf, etc. Or in other words, sons are preferred for visiting instead of brothers (see Figure 3.4). Therefore, the *CLOSED*-list represents the actual traversal path from the starting node to the actual visited node in the graph. If a leaf with no solution (dead end) is found, then the next first node of the *OPEN*-list has to be visited and expanded. This means to do some *backtracking*. This strategy of selecting the first alternative node leading to a possible other solution is called to be *monotone*. More intelligent or *not-monotone* backtracking goes back to a node, which has possibly been responsible for this path that led to the dead-end. But then a causal dependency management is required to achieve that. Representatives are *dependency-directed* and *knowledge-based backtracking*.

DFS

In *breadth-first-search* (BFS) not yet expanded nodes are handled in FIFO[4]-strategy. That means the *OPEN*-list is managed like a queue. By this strategy brothers are visited before sons (see Figure 3.4). The strategy will find the solution with minimal depth (i.e. shortest path from the starting node to the leaf).

BFS

[2] first-in-first-out
[3] last-in-first-out
[4] first-in-first-out

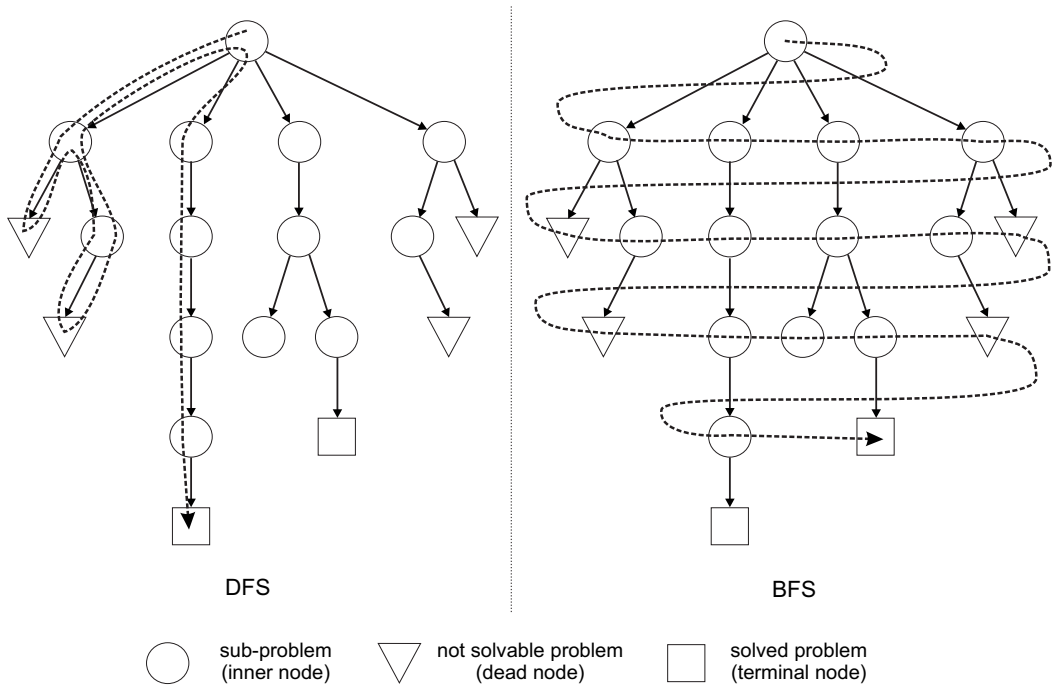

Fig. 3.4: Examples for the traversal paths, if *depth first search* (DFS) or *breadth first search* (BFS) is applied to a sample graph until first solution is found (see [128, p. 17+31]).

All the previously described algorithms try to find a solution, not necessarily an optimal solution. Assuming that each node produces costs, the optimal criteria is to find a so-
CFS lution path, whose sum over the node's costs is minimal. The *cheapest-first-search* (CFS) selects a member of the *OPEN*-list first that produces minimal costs.

HC Another very simple strategy is *hillclimbing* (HC). The *OPEN*-list is again handled like a stack. But the nodes are pushed on the stack according to the costs they produce, where the nodes with higher costs are pushed first. This strategy is often used by humans, if they search for an optimum. But obviously this strategy does not necessarily lead to a global optimum; it can result in a local optimum.

Informed strategies promise better search results. All of the following algorithms presented next are members of this category.

BF The informed *best-first search* (BF) makes use of an evaluation function $f(n, p, g, K)$, which often is an heuristic, in order to determine the potential for finding an optimal solution, when the node n is visited next. Parameters of this function can be the previously visited path p, the goal g and general domain knowledge K. The node n from the *OPEN*-list is selected for expansion, whose $f(n, p, g, K)$ is maximal.

GBF A generalisation of BF is the *general best-first search* (GBF). It requires three functions $f_1(G'')$, $h(n')$ and $f_2(n)$, because it calculates the most promising node in two steps. (1) It estimates, for all successors n' of nodes in the *OPEN*-list, the total cost for a solution by a heuristic $h(n')$. The heuristic $h(n')$ estimates the expected costs for the sub-problem graph $G_{n'}$, whose starting node is n'. By integrating these results it calculates the (estimated) costs for all possible solution-base graphs G'' by using $f_1(G'')$, where G'' is an

extension of the actual solution-base-graph extended by successor nodes n'. The function $f_1(G'')$ calculates something like the sum over all edges and nodes of G'', expecting that a small graph (with minimal f_1) leads to a better solution. Due to the function $f_1(G'')$ and $h(n')$ the most promising new solution-base graph G_0 is selected. (2) It will proceed with the most expensive successor node $n' \in G_0$ (or the node $n' \in G_0$, whose error for estimating $h(n')$ is maximal). Therefore, it will calculate $f_2(n)$ for $n \in G_0$ and $n \in$ *OPEN*-list. Assuming that a costly node (with maximal f_2) leads early to a bad solution, it is selected first for expansion. (A cheap node does not necessarily lead to a better solution, if considered that the path is long enough.) This algorithm is applicable to problem graphs and especially to AND/OR-trees. It traverses the complete graph G by making use of the explicit graph G', whose nodes are in the *OPEN*-list (see Figure 3.5).

```
1. Put start node s ⟶ OPEN;
2. Based on explicit graph G', function f1, heuristic
   h:
   Compute the most promising solution-base graph G0;
3. Based on function f2:
   Select a node n that is both on OPEN and and in G0;
4. Expand n;
   Add all n' = successor(n) to OPEN and to G';
   Install back pointers to parent n to all n';
   Compute foreach successor n' the heuristic h(n');
5. IF ANY successor n' is a terminal node THEN
   (a) Label n' "solved" if a goal or "unsolvable" if
       not;
   (b) Apply the SOLVE-LABELING-PROCEDURE:
        i. Set n' = parent(n');
       ii. IF n' is an OR-node and one of its successors
           is labeled "solved" THEN label n' also
           "solved";
      iii. IF n' is an OR-node and all of its successors
           are labeled "unsolvable" THEN label n' also
           "unsolvable";
       iv. IF n' is an AND-node and all of its successors
           are labeled "solved" THEN label n' also
           "solved";
        v. IF n' is an AND-node and one of its successors
           is labeled "unsolvable" THEN label n' also
           "unsolvable";
       vi. IF new labels have been created and n' ≠ s
           THEN Goto i;
   (c) IF s is labeled "solved" THEN return G0;
   (d) IF s is labeled "unsolvable" THEN return
       "failure";
   (e) Remove from G' nodes whose label can no longer
       influence the label of s;
6. Goto 2;
```

Fig. 3.5: General Best-First Search (GBF) algorithm.

GBF does not find the optimal solution. But if the function f_1 is optimistic and the termination criteria of the algorithm is delayed (see Figure 3.6), it will be optimal. Then this algorithm will be called GBF*.

GBF*

1. Put start node $s \longrightarrow$ OPEN;
2. Based on explicit graph G', function f_1, heuristic h:
   Compute the most promising solution-base graph G_0;
3. *IF ALL leafs of G_0 are labeled "solved" THEN return G_0;*
4. Based on function f_2:
   Select a node n that is both on OPEN and and in G_0;
5. Expand n;
   Add all $n' =$ successor(n) to OPEN and to G';
   Install back pointers to parent n to all n';
   Compute foreach successor n' the heuristic $h(n')$;
6. IF ANY successor n' is a terminal node THEN
   (a) Label n' "solved" if a goal or "unsolvable" if not;
   (b) Apply the SOLVE-LABELING-PROCEDURE:
       i. Set $n' =$ parent(n');
      ii. IF n' is an OR-node and one of its successors is labeled "solved" THEN label n' also "solved";
     iii. IF n' is an OR-node and all of its successors are labeled "unsolvable" THEN label n' also "unsolvable";
      iv. IF n' is an AND-node and all of its successors are labeled "solved" THEN label n' also "solved";
       v. IF n' is an AND-node and one of its successors is labeled "unsolvable" THEN label n' also "unsolvable";
      vi. IF new labels have been created and $n' \neq s$ THEN Goto i;
   (c) IF s is labeled "unsolvable" THEN return "failure";
   (d) Remove from G' nodes whose label can no longer influence the label of s;
7. Goto 2;

Fig. 3.6: GBF* algorithm, if heuristic $h(n)$ is optimistic.

The BF and GBF algorithms give a framework for defining search strategies. The specification of the functions f_1, f_2 and h have great impact on the complexity, efficiency and overall cost of the algorithm in terms of time that is used and memory that is required. In the following section we will present some possible function definitions and the resulting names for the search strategies.

3.3.2 Special Search Strategies by Defining Cost Functions

In order to select the next node for expansion in the previous section the functions f_1 and f_2 have been mentioned. For defining them we will first define a *weight* or *cost function*. Therefore, we introduce local graph properties by assigning a local weight (in terms of merits Q or costs C) like:

- weights of inner nodes, e.g. calculation costs
- weights of edges, e.g. transport costs
- weights of terminal nodes, e.g. terminal payoffs

By having this we define the weight of a solution graph G with starting node n as

$$W_G(n) = F[E(n), W_G(n_1), W_G(n_2), \ldots, W_G(n_b)]$$

where

$n_1, n_2, \ldots, n_b$ are all direct successors of n,
$E(n)$ calculates the local weight of n and its incident edges to its successors,
F is a function that describes how to allocate and charge the costs or merits.

Such a weight function is considered to be *recursive*. The assignment of $W_G(n)$ to all nodes is named *cost* or *merit labelling* . The calculation of this labelling can be done *"bottom-up"* from the terminal nodes up to the starting node. This means, it has to be done in reverse order in contrast to the searching. The function f_1 can be defined as the labelling of n: $f_1(G) = W_G(n)$.

In order to prevent the traversing of the complete graph for the weight labelling, the weight label of a node is estimated instead of exactly calculated. For this estimation the heuristic $h(n)$ is used. In a pure OR-graph (problem graph, which is manageable by BF) the estimated cost label $\hat{C}(n)$ is defined as

$$\hat{C}(n) := \begin{cases} h(n) & : \; n \text{ is in OPEN} \\ min_i\{F[E(n), \hat{C}(n_i)]\} & : \; n \text{ is in CLOSED,} \\ & \quad n_i \text{ is successor of } n \end{cases}$$

The costs of a path $P(n)$ from starting node s to a node n is defined by

$$\hat{C}_{P(n)}(s) := h(n) + \sum_{n' \in P(n)} E(n')$$

If in BF the function f is defined as $f(n) := \hat{C}_{P(n)}(s)$, then BF is called Z. If additionally the termination criteria of the search algorithm is also delayed, then Z is named Z*. Z, Z*

In an AND/OR-graph the estimated cost label $\hat{C}(n)$ is defined as

$$\hat{C}(n) := \begin{cases} h(n) & : \; n \text{ is in OPEN} \\ min_i\{F[E(n), \hat{C}(n_i)]\} & : \; n \text{ is OR-node,} \\ & \quad n \text{ is in CLOSED,} \\ & \quad n_i \text{ is successor of } n \\ F[E(n), \hat{C}(n_1), \ldots, \hat{C}(n_b)] & : \; n \text{ is AND-node,} \\ & \quad n \text{ is in CLOSED,} \\ & \quad n_i \text{ is successor of } n \end{cases}$$

If in GBF the function f_1 is defined as $f_1(n) := \hat{C}(n)$, then GBF is called AO. If additionally the termination criteria of the search algorithm is also delayed, then AO is named AO*. AO AO*

Often it is required that the function F is *order-preserving*. That means that for all possible heuristics $h(n)$ F always produces the same order of costs. And therefore, the same path to node n' will be selected. Formally this means

$$\forall\, n_1, n_2, n' \text{ is successor of } n_1 \text{ and } n_2, E, h_1, h_2 :$$
$$F(E(n_1), h_1(n')) \geq F(E(n_2), h_1(n')) \Rightarrow F(E(n_1), h_2(n')) \geq F(E(n_2), h_2(n')).$$

A *summing cost function*, i.e. $F = c(n, n') + h(n)$ like $\hat{C}_{P(n)}(s)$, is order-preserving. The algorithm Z* used with a summing cost function is named A*. A*

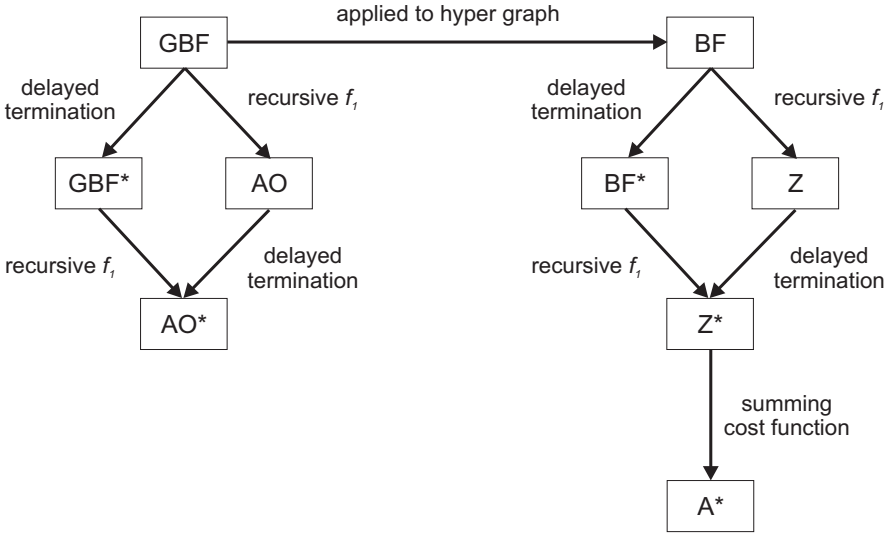

Fig. 3.7: Hierarchy of BF algorithms for searching in AND/OR-trees (see [128, p. 26]).

Nilsson found that A* always finds an optimal solution, if $h(n)$ is an optimistic function (Result 4 in [94]). Therefore, for configuration problems mostly A* is implemented.

DFS, BFS and CFS strategies are special cases of A* or Z*:

- BFS is A* with $h(n) = 0,\ c(n, n') = 1$
- DFS is Z* with $h(n) = 0$
- CFS is A* with $f(n') = f(n) - 1,\ f(s) = 0$

The following section tries to show, how search can be used for configuration. Moreover, other approaches for configuration besides search will be presented. This will result into a categorisation of configuration strategies.

3.3.3 Overview of Configuration Approaches

This subsection tries to categorise and structure the well-known algorithms used for configuration. It distinguishes the algorithms by their main control flow and the used data

structure for the domain knowledge.

All configuration problems have a few common characteristics. Günter et al. [50] state that these are:

- A set of objects of the domain and properties of these objects must be defined (domain knowledge).
- A set of relations between these objects (taxonomical or compositional) have to be defined (domain knowledge).
- A requirements (task) specification (problem specific knowledge) of the demands, which a final configuration has to accomplish, have to be defined.
- The control knowledge about the configuration process is coded in a configuration algorithm, which has the above three items as input.

A categorisation of the approaches can follow many aspects. For instance, Stein [126, 127] distinguishes these on the basis of either the structure or the function being of main interest for the configurable system. Principally, he considered the data structure for the domain knowledge representation. He derived the following classification:

- *Structure-oriented:* describe the connections between components. All objects are seen from a global view of the complete system. Their interrelationships are of main interest.
  - *associative:* define structural connections of the components. e.g. *if component A and B are present, then component C must also be present and component D must not be present*.
  - *compositional:* describe the aggregation of the system top-down. A typical AND/OR-tree can represent the structure.
  - *taxonomical:* describe the system by an OR-tree, where each node refines its parent node.
- *Function-oriented:* describe the properties or behaviour of the components. The view to the objects is more local.
  - *property-based:* all objects are defined by pairs of a property type and its concrete value.
    - *resource-based:* only two types exist: offered and required resources. Required resources must be delivered by other components, which offer them.
  - *behaviour-based:* components have ports and constraints besides properties. Ports are possible connections to other components. A component being added to a configuration must be connected to a component, which is already part of the configuration. Constraints can relate properties and ports to each other and can compare same properties of different ports. Components can only be connected, if their ports are compatible (or of same type) and all constraints are satisfied.
    - static
    - dynamic

Subsequently, Stein et al. [129] propose to distinguish how the model is being formulated. More precisely they classify after the model calculus. The calculus can also have great impact on the control algorithm. The model can be refined, compiled or constructed:

Component Model Refinement: At the top level a system description is very abstract and incomplete. That means a lot of details are over looked (or neglected). Therefore, it is not yet implementable. In a step-wise and hierarchical procedure missing details are added and specified until the system can be implemented.

Component Model Compilation: Problem-specific control knowledge (algorithm) is added to the problem-solving knowledge in such a way, that the configuration process itself is able to find a solution (in an efficient way).

Component Model Synthesis: The system has to be composed from a set of model fragments. All possible fragments are arranged in a certain topology. Which fragments have to be arranged in what (sub-)topology, has to be decided.

Besides these theoretical classifications of configuration approaches, I propose the following categorisation for configuration systems that had been developed in research and industry up to now. The systems can be classified into the following categories:

- using description logics
  - rule-based expert systems
  - truth maintenance systems
- model-based
  - concept hierarchies
  - structure-based
  - resource-based
- constraint-based
- case-based
- simulation engines

The different categories will be explained in the following paragraphs:

RULE-BASED In *rule-based configuration* the domain knowledge is completely described by rules using a descriptive logic. These systems are developed as *expert systems* and are influenced in a major way by the research done in the area of *artificial intelligence* (AI). Main drawbacks, which have been found during many implementations, are that the knowledge acquisition in the sense of coding the domain knowledge into rules is very difficult. Moreover, it is hard to maintain a big rule set or to keep it consistent[5]. Also, it is often difficult to adapt the rules to new cases or application scenarios. The lack of modularity is another reason why users and experts find this way of configuring very complex, unclear and confusing. Nevertheless, this way of describing domain knowledge is very formal and, therefore, useful, if special conditions or properties of a final configuration must be proved formally (*validation*).

TMS Doing *configuration with truth maintenance systems* is used to manage beliefs in given sentences. It provides justification for conclusions, recognises inconsistencies and supports

[5] Consistent rules require that there does not exist a contradiction between any two rules or their implications.

for default reasoning. An explanation of a conclusion can be constructed by tracing the justification of the assertion. It may tell that some sentences are contradictory. In the absence of firmer knowledge, it reasons from default assumptions. A *truth maintenance system* (TMS) can be categorised as follows:

Justification-Based Truth Maintenance System (JTMS): It is a simple TMS where one can examine the consequences of the current set of assumptions. The meaning of the sentences is not known.

Assumption-Based Truth Maintenance System (ATMS): It allows to maintain and reason a number of simultaneous, possibly incompatible, current sets of assumptions.

Logical-Based Truth Maintenance System (LTMS): Like JTMS it reasons with only one set of current assumptions at a time. It is more powerful than JTMS, because it recognises the propositional semantics of sentences, i.e. understands the relations between p and $\neg p$, $p \wedge q$ and $p \vee q$, and so on.

A sentence and a conclusion representation for TMS is described in [46]. There, labels are attached to arcs from sentence nodes to justification nodes. This label is either "+" or "–". Then, for a justification node we can talk of its *in-list*, the list of its inputs with "+" label, and of its *out-list*, the list of its inputs with "–" label.

CONCEPT HIERARCHIES

Like in object-oriented design in *configuration with concept hierarchies* the representation of the object knowledge is structured into a hierarchy. They can mostly be classified into *taxonomical hierarchies*, where *is-a*-relations are predominant, and *compositional hierarchies*, which describe *has-parts*-relations. The domain knowledge specifies facets or skeletons or skeleton plans [47, 102] that describe alternative sets of values, objects, or properties. Often only this information about selected alternatives is included into the solution. Whereas *is-a*-relations (inheritance of properties from its super classes) are used to structure, to reuse properties, and to support information hiding, *has-parts*-relations link aggregates (containers) and their components together.

STRUCTURE-BASED

In *structure-based configuration* a natural compositional and hierarchical structure of the objects serves as a guideline for the control of the problem solution by using a divide-and-conquer strategy. Mandatory components form AND-related successor nodes of an aggregated node. Whereas the optional choices for a component form OR-related successor nodes. Thus, the solution space spans an *AND/OR-tree* [102] (and sometimes a directed acyclic graph) with the final configuration as the root. The AND-relations define sub-problems in which the OR-relations, which give alternative sub-solutions, have to be solved. The main distinction within concept hierarchies is that these structures show directly how to find a solution. They are widely used for technical systems, which can naturally be structured into components. (Tools: XCON, PLAKON, KONWERK, see Section 3.4_{41})

RESOURCE-BASED

Resource-based configuration means that the components of a configuration are considered to produce and consume resources. The main property exploited is the fact that resources provide for a specific functionality or some (virtual) raw materials. On the other hand, the components of a configuration also require or have to import some resources in order to export their functionalities or raw materials to others. In fact, each component of a configuration demands resources and at the same time supplies resources. Thereby, different types of resources are distinguished. The primary goal of a valid configuration is

that in sum all internally supplied resources (of one type) and all demanded resources (of that type) are of equal amount and are minimal. Exceptionally, the external supplied resources (or functionalities or raw materials) of the whole configuration have to be excluded from this sum, because these are the absorbed requirements of the final system. The aim of the configuration process is to achieve a balanced and minimal demand-and-supply of all resources. In this way the configuration problem is regarded as a typical optimisation problem. Between all components of a configuration there exists a competition for resources. A typical problem solving strategy is to begin with a starting configuration. The deficits are then recognised and thus equated by the instantiation of new components, and so on.

CONSTRAINT-BASED

The main objective regarded in *constraint-based configuration* is the relationship between an object and its properties. The constraints which can be formulated describe that a property must or must not be valid, is in a certain range, may be limited, or is of any other similar constraint. Thus, a constraint restricts the solution space for a local property of an object (or even of a lot of objects). In [91, 120] a categorisation of *constraint types* is given:

synthesis constraints:	affect feasible solutions
interaction constraints:	arise from interactions between structural subsystems
causal constraints:	equations or equilibrium of physical laws
parametric constraints:	restrict object attributes
evaluation constraints:	rank alternatives
hybrid constraints:	combination of the above

An important property of a constraint is that it must be activated before it affects other object's properties. Often constraints are attached to the objects and the condition for its activation is often based on the insertion of this object into the configuration, but can also reason about other local properties of the object. This fact raises the problem that constraints are accumulated incrementally during the configuration process. This is called *constraint propagation*. Even more, they form a *constraint network* because of mutual influences. The problem is to satisfy all constraints at the same time. This problem is (beyond configuration) also well known as the *Constraint Satisfaction Problem* (CSP).

CASE-BASED

The main idea on which *case-based configuration* is based assumes that similar demands lead to similar solutions. Hence, by identifying only the changes in a new request compared to an old one, this gives hints on the solution. Moreover, the old solution may be reused by applying only minor replacements. Or in other words: Sub-solutions must be repaired according to the changes in the requirements. But this raises some problems: (1) Old requests and their solutions must be stored in a kind of case repository. (2) How do changes in the requirements affect an already made (old) configuration? These are the main topics research is dealing within the area of *case-based reasoning* (CBR).

SIMULATION-BASED

When configuration is done by a simulation system the selection of appropriate (sub-) solutions is done as mentioned before, but the evaluation of the cost function or even the proof of correctness of a configuration is done by simulating the configuration in its environment. A simulator evaluates one or more configurations for their usability in the future. Like in a chess simulation of a game, the benefit or merit of a specific configuration is estimated and based on this decision it is used in the future or rejected.

Nevertheless, besides the above given classification of approaches for configuration sys-

tems there exist programs that are members of more than one category. Because they make use of more than one approach at the same time or sequentially or periodically. They are called to be *hybrid configuration systems*.

A typical representative is XCON (see page 41), which has been one of the first configurators on the market. It makes use of rules, but they are grouped into hierarchically structured rule contexts.

The so called *skeleton-based configuration* [102] is mostly done by the use of AND/OR-trees. Therefore, it is a structure-based approach. But inherently the tree also often defines a hierarchy.

PLAKON and its successor KONWERK follow concerning modelling the structure-based approach, but for conflict resolving they define a constraint network, which is solved by constraint propagation.

Also the TEReCS configurator TGEN, which has been developed during this work, is a structure-based approach that uses an AND/OR-tree for the general domain knowledge representation. But additionally constraints can be defined, that give powerful hints (or restrictions) during configuration.

Results of the research community show that some of the approaches are similar in its expressive power. In [90] it is shown that the skeleton-based configuration can be transformed into a resource-based configuration and vice versa. They show that resource-based and resource-based configuration plus rules are equivalent and, moreover, both can be transformed into an AND/OR-tree of a skeleton-based configuration problem. Additionally, they prove that the problem of finding an optimal configuration for these two approaches is NP-complete.

3.4 Literature Survey

Configuration is mostly done for technical systems, product data management or customer consulting and sales services for business processes. Configurable software systems can also be found in the area of Java™ programming (like it is offered by Jini and Java Enterprise).

One of the first configuration systems, which was commercially used, is R1/XCON [7, 89]. It was developed at DEC and was used to configure the computer systems, which were offered by them. The customers had different requirements and in order to have a computer system as cheap as possible, the purchase orders contained only basic function units. But it is the vendor's task to deliver a working and operationally correct computer system. So, additional components must be added in order to have a complete and correct specification of the system. The decisions to be made were concerning type, properties, placement, electrical connection, power consumption, etc. In order to make the orders consistent by adding or substitution of components, DEC's sales managers had to check the orders for these. The time and persons spent for this task were incredibly high, because of the huge amount of orders. DEC's aim was to automate this process. The result was a rule-based system containing nearly 6,200 rules for over 20,000 parts. R1/XCON

AKON [92] is following the resource-based approach. It is used by Bosch/Telenorma AKON

and configures telephone switchboards.

SICONFEX Siemens AG developed SICONFEX [61, 78] in order to configure the operating system of their SICOMP computers. This configurator program has a lot of similarities with TGEN, which is developed in this project. Input for SICONFEX is the actual hardware configuration, on top of which the operating system should run, and application specific customer requirements. The output are the terms of configuration statements for a generator program. The static domain knowledge is structured into a universe of physical objects, existing software modules and hypothetical memory partitions. It uses frame-like object structures, conceptional taxonomies, inheritance mechanisms, rules, message passing between objects, active values and LISP functions in order to describe the domain knowledge. A hybrid problem solving algorithm uses optimisation, heuristic and hypothetical decisions and labels for backtracking.

MMC-KON MMC-KON was developed by Baginsky et al. [6] at Siemens AG, Erlangen. It is used to configure distributed automation systems based on the SICOMP MMC 216 multi-microcomputer system. As a first step MMC-KON generates a function plan containing the processors for the automation functions. Then various criteria have to be taken into account, e.g. process structure, integrity, inter-process communication, bus capacity, etc. Next, each processor is configured by selecting suitable modules. The modules are placed into sub-racks with consideration given to the number of slots, the available power supply, preferred module locations and combination options. In the next step parameter sets are assigned to the modules and finally, the sub-racks are arranged to cubicles considering placement constraints. The modelling is done by object-frames, which are related to each other by *is-a* and *has-parts* relations. The system permits free user interaction at any time and the changing of configuration steps at any time. These characteristics are in distinct contrast to XCON (see Section 3.4) where the order is fixed and it precludes user interaction. Similar to SICONFEX also in MMC-KON the problem specific knowledge or the so-called task definition is acquired interactively. This phase is supported by the domain knowledge and the display capabilities of a modern workstation (graphical user interface).

ALL-RISE The ALL-RISE system [120, 121] configures the preliminary structural design of buildings. Its input is the architectural or spatial plan of a building represented by a three-dimensional grid. The output will be a set of feasible alternative "structural systems" ranked according to their appropriateness. The internal model is dominated by schemas (frames) having *is-a* and *has-parts* relationships. The top node of this tree represents an acceptable structural system at its most abstract level. Subordinate nodes represent either alternative specialisations or partial designs. A complete (successful) design is a subtree with exactly one successor for *is-a* branches and all successors for *has-parts* branches (like it is in a conceptual AND/OR-tree). A lot of the domain knowledge is encoded by constraints besides alternative *is-a* and *has-parts* relationships. But in contrast to other work done on constraint-based reasoning [52, 55] the developers of ALL-RISE did not implement an independent constraint propagation algorithm capable of satisfying multiple constraints simultaneously. They preferred a step-by-step procedure at the domain knowledge hierarchy. This is very similar to the approach that is used in TEReCS (see Section 6.3.4$_{110}$).

PLAKON The PLAKON shell [34] was mainly developed in 1986 – 1989 in a BMFT[6] joint project

[6] German department of research and technology

TEX-K [32, 33, 52, 93] by Battelle Institut (Frankfurt), Siemens AG (Erlangen), Philips GmbH (Hamburg), URW (Hamburg) and the University of Hamburg. PLAKON seems to be the most advanced system dealing with configuration tasks. In PLAKON several complex configuration methods as well as a mighty conception-hierarchy description language are implemented. PLAKON is designed to support diverse applications among which are configuration for multi-computer systems (MMC-KON, see page 42), configuration of computer vision systems for quality control in manufacturing, configuration of automated systems for industrial x-ray analysis, configuration of systems for laboratory experiments, generating work plans for mechanical manufacturing and configuration of electrical engineering aggregates using standard components. PLAKON is based on some general observations [91] for technical systems:

- In technical systems the configuration process is governed by highly structured knowledge about components and aggregates.
- Domain knowledge is naturally organised into an **object-oriented (oo) hierarchy** based on *is-a* and *has-parts* relationships and into a **constraint network** relating object properties to each other.
- Constraints require special treatment, because: (1^{st}) They do not conform with the oo-style and (2^{nd}) constraints tend to affect the order of configuration steps.

Due to these reasons the domain knowledge is represented by a conceptual hierarchy modelling these oo-relations. Additionally, the control algorithm is built from a global configuration cycle where various configuration steps are selected according to the constraint activation (see Figure 3.8). This shows that in PLAKON the steps of the control flow

```
1. Determine phase and strategy
2. The strategy determines focus and selection criteria
3. Possible configuration steps are determined and
   placed onto an agenda:
   (a) Top-down refinement:
       i. decomposition along has-parts relationships
      ii. specialization along is-a relationships
   (b) Bottom-up composition:
       i. aggregation along part-of relationships
   (c) Parameterization:
       i. value assignment or restriction
   (d) Instantiation of a new object
4. A configuration step is selected according to a
   selection criteria
5. The step is executed in a particular value selection
   procedure
6. The constraint net is activated optionally
7. New elaboration is checked for conflicts and
   termination
8. Goto 1.
```

Fig. 3.8: Algorithm of the configuration cycle in PLAKON.

are highly dynamically ordered and that the order is influenced by the domain knowledge.

KONWERK KONWERK [56] was developed in a BMFT joint project named PROKON of the University Halle, University Hamburg, RWTH Aachen and HTW Zwickau from 1991 – 1995. It was based on results made with the PLAKON shell [57]. It follows similar approaches and can be seen as a successor of PLAKON.

$^{\text{art}}deco$ An outstanding configuration tool for technical systems in the area of fluidics is $^{\text{art}}deco$ [13, 22, 129, 130]. A hydraulic circuit consists of mechanical, hydraulic and electronic components. The system was developed in order to design a circuit in compliance with the customers' demands. Hence, the design process is supported graphically and the checking of a system is automated. Therefore, in $^{\text{art}}deco$ a graphical as well as a technical language for the specification of the system and its constraints was developed. All technical information of an object are described locally. Thus, the user can easily and independently specify new objects. Moreover, this leads to a second major concept in $^{\text{art}}deco$: the global behaviour of a technical system is derived from its local component descriptions. Their knowledge representation model distinguishes *elements*, *gates*, *connections* and *sources* (or *sinks* resp.). These elements will be used to describe a system's topology. To describe the technical behaviour of the components, a formal language had been developed that provides numerical and symbolic description of behaviour as well as non-continuous descriptions. Additionally, constraints are specified, which have to be checked for a correct system. These include topological, connection and functional constraints. Functional constraints are processed using the method of constraint propagation, whereas the others are checked by unification of global variables.

Günter et al. [51] present five commercial configurators for technical systems and mainly sales services. For detailed information we refer to their article, but in order to give an overview a brief summary is given here.

CAS The CAS-Konfigurator was developed by SOLYP Informatik GmbH, Nürnberg. The system was mainly developed for the generation of technically correct offers where components, which have a rich variety of different implementations, are selected from a product catalogue. The selection and parameterisation is done interactively. The knowledge representation is based on a hierarchical frame concept, which allows multiple inheritance. Additionally, forward-chained rules and decision tables can be used in order to specify the domain knowledge. All of these have to be specified in C++ templates. During the interactive selection process the given constraints are checked. Inconsistencies are solved by chronological backtracking. The result of a configuration can be viewed as a part list, selection tree or a spatial design.

COSMOS COSMOS was developed by repas AEG Software GmbH, Berlin. It is based on previous work [62] done at DaimlerChrysler AG, Research and Technology, Berlin. The system follows the resource-based approach. The database contains resource descriptions and component definitions. The authors assume that the lifetime of resources is much greater than that of components. Thus, the administrative overhead for the management of the database is less. A property, worth mentioning, is the fact that the configuration process runs nearly fully automatically.

ET-EPOS The TDV GmbH (Karlsruhe) had developed and distributed the ET-EPOS system. Its main purpose is to support sales managers. The domain knowledge is specified in the

System	R1/ XCON	A- KON	SICON- FEX	MMC- Kon	ALL- RISE	PLA- KON	KON- WERK	art *deco*	TGEN (TEReCS)
Primary Application Area:									
Sales Offers	×								
Technical Systems	×	×		×	×	×	×	×	
Software Synthesis				×					×
Operating Systems			×						×
Techniques:									
Objects			×	×				×	×
Modules			×	×					
Parameters						×	×		×
Rules	×		×						
Constraints					×	×	×	×	×
Resources		×							
Structure			×	×	×	×	×	×	×
Concept Hierarchy	×					×	×		×
Methods:									
Interactive			×	×				×	
Automatic									×
Heuristics			×	×					
Constraint Propagation						×	×	×	
User Interaction				×					
Change of Conf. Steps				×		×	×		

Tab. 3.1: Comparison of some research driven configuration tools.

form of tables in the manner of spreadsheets like they are implemented in Excel^{TM}. Because of this smart usability and user-friendliness it is widely accepted among users. The tables describe decision trees, which are based on the standard DIN 66941. The system is split into two programs: (a) an administrative tool to maintain the tables and for knowledge acquisition and (b) the user system in order to create configurations.

SCE

The SCE (Sales and Configuration Engine) from SAP AG (Walldorf) is a product configurator for sales offers as well as for the configuration of technical systems. It has an excellent connection and data interface to the SAP R/3 logistics module. It is used in order to support sales managers for offers of technical complex systems. In SCE a TMS is used for configuration.

SECON

SECON from Camos GmbH (Stuttgart) consists of three programs for the maintenance of the database, the development of graphical user interfaces for the configuration and the run-time system for the configuration itself. The knowledge base is represented by a structure-based approach. A (graphical) class editor helps to organise the inheritance and aggregations of the classes. The configuration process is also interactive, but it is supported by automatic checks for consistency. Inconsistencies are displayed by an explanation component. They have to be solved manually by the user.

Nearly all of the commercial systems have a comfortable graphical user interface for the interactive selection and parameterisation of components. The configuration process is often done interactively by the user, whereas he is supported by automatic consistency

System	CAS	COSMOS	ET-EPOS	SCE	SECON
Application:					
Sales Offers	×		×	×	×
Consistency Check	×	×	×	×	×
Automatic Process		×			
Acquisition Tool			×		×
Techniques:					
Objects	×	×	×	×	×
Parameters	×	×	×	×	×
Rules	×		×		×
Constraints				×	
Resources		×			
Methods:					
Interactive	×	×	×	×	×
Heuristic	×	×		×	
Structure-based			×		×
Resource-based		×			

Tab. 3.2: Comparison of some commercial configuration tools (adapted from [51]).

checks. A summary is given in Table 3.2.

3.5 Advantages of Configuration within a Real-Time Operating System

The main aspect why configuration is required for embedded real-time operating systems is due to the demand of using minimal resources. A small memory footprint and the efficiency how fast services are executed are two requirements which lie in the nature of embedded real-time control and are driven by economic and qualitative requests. Therefore, configuration is an ideal tool to fulfil these requirements in contrast to the requests of the applications. The trade-off between a small and efficient design and the request for a broad support of a variety of general and special services can be solved by configuration. The services of operating systems are traditionally designed in a more or less generic way. This is, because they must cover a lot of different application scenarios. But for embedded real-time control, services are developed just to support the given scenario in a very efficient way. This leads to a huge variety of different implementations of the same general service for different pre-requisites. To implement all these services into a single operating system is not desirable. Here, configuration helps to select the appropriate services and to create an optimal operating system. Moreover, configuration not only selects the ideal service but can also help to configure or to build this service out of a given service skeleton.

3.5.1 Goals of Operating System Configuration

We can identify the main goal for the use of configuration in operating systems. Not all of the functions and services, which the operating system can support, should be inte-

grated into the used instance in order to save memory and, because the function is never required. For example, nearly all of the general purpose operating systems for desktop or server applications, like Windows^{TM} or Linux, come along with a broad variety of drivers for nearly every available hardware device. For example, they support a huge amount of graphic and communication adapters, other I/O controllers, mass storage systems, extensions for hot plugging, etc. In the actual system often only one graphic and one communication adapter is installed. For this reason, only the appropriate drivers for the present hardware devices have to be loaded into memory.

Often, the presence of a special driver request for further drivers. For instance, when a hard disk is present, not only the driver for its access is required, but also a driver, which manages the file system structure on that device. Another example is the communication adapter. If this is an Ethernet card, then additionally, the Ethernet protocol stack (as a driver or a set of drivers) is required.

In modern operating systems the size of a configuration item is often the size of a driver or a module. This means that only complete drivers or modules can be included or excluded from the operating system's implementation. This holds especially for commercial systems.

The decision about the integration or exclusion of drivers or modules is mostly done at the setup or installation time of the operating system on the computer. Moreover, the setup programme, which is delivered together with the operating system, works like a simple driver collector. In advanced systems this setup programme detects automatically the present hardware devices and selects the appropriate drivers for loading during boot-time or for integration into the kernel. When the automatic hardware detection fails or is not present, then the user must manually select the required drivers. Additionally, the previously mentioned course-grained dependencies between the drivers are considered. These dependencies are often defined as simple implication rules of the form $a \rightarrow b$, which means that the driver b is also required, when the driver a is loaded. A rule of the form $a \rightarrow \neg b$ can often not be defined. This rule means that if a is loaded, then b must not be loaded. The lack of this specification possibility results often into the well-known driver incompatibilities of simultaneously loaded drivers.

The granularity of configuration is often only at driver or module level. The reason for this might be the compromise between manageability and resource consumption. But this granularity level is not sufficient for embedded real-time applications, because they request for a more efficient resource usage. Even inside a module configuration must be applied in order to tune its behaviour. This means that an operating system function is adapted or customised to the given application scenario. For example, if the application does not use the UDP protocol of the TCP/IP stack, then it possibly can be removed from the protocol hierarchy. Often, the TCP/IP stack can only be completely included or excluded. Consequently, the granularity of the operating system's configuration should be at functional – or for object-oriented versions – at object level, which means at source code level.

However, in the embedded world the operating system can be selected for serving only one application. For desktop or server computers the operating system has to serve a lot of applications. It is not known in advance, when and in which combination they will be started. The life-time of the operating system is much greater than the life-time of an application. But for the embedded scenario, the life-time seems to be identical. The

operating system and the application exists both as long as the product, in which they are integrated, works. This offers more potential for an optimal, small and efficient implementation of the operating system. Knowledge about the applications behaviour can help to configure the operating system's code more efficiently. The application scenario should have direct impact on the implementation of an operating system service.

3.5.2 Examples

We will give a few examples, which show what items of the operating system can be configured in dependency to the application's requirements or behaviour.

Resource Protection. The use of a resource must normally be protected by one or more critical sections in the case of simultaneous access (mutual exclusion). But often it is not known in advance, if this special use case of the resource will be applied or not. So, for the general case a designer protects the critical section, e.g. by the use of a semaphore. But if the resource is then not used in parallel, the useless synchronisation creates a non negligible overhead. Configuration helps to overcome this problem. The use of the semaphore will be inserted or deleted due to the special use case.

This feature can be generalised to any operating system service. The circumstances, under which the service is used, determine exactly the optimisation potential that can be exploited for the implementation of the service.

Device Drivers. In general an operating system must support all hardware resources with drivers. This means to include code and to use memory space in order to manage the status of the resource. A general-purpose operating system must support a huge variety of hardware. But, if the hardware resource is not used, then the driver can be eliminated. This can also be achieved by configuration. Moreover, if not all of the functionalities of the device are used, then only parts of the driver need to be integrated.

Hardware Architecture. Not only the usage of a hardware device determines, whether a driver must be loaded or not. But also, whether the device is present in the actual hardware architecture, should lead to the integration or exclusion of a dedicated driver.

Service Dependencies. The integration or removal of services itself requests for configuration. High-level services depend on low-level services and the existence or non-existence of a service requests or forbids other services.

3.5.3 State of the Art

Customisation and configuration is not a new approach for adapting a system to the requirements of an application. In the field of (real-time) operating systems some interesting ideas for customisation can be found. The systems can be classified according to support for configuration during compile-time and/or run-time. Furthermore, in general customisation can be distinguished by its level of manipulation of the system. First, systems can only select monolithic modules to be included. Examples are Linux™ [108], QNX [96] or RTEMS [141]. For instance, device drivers in QNX or Linux can be loaded or not. In Linux this is even possible at run-time. In RTEMS special functionalities (e.g. rate

QNX

RTEMS

monotonic scheduling, etc.) can be added to the system by selecting particular modules. Additionally, RTEMS provides for dynamic system extension by allowing the application to bind special exception handlers to certain kernel events (like the timer interrupt is bound to the context switching).

Second, systems can be customised by choosing different possible (μ-)kernels. PURE [114] is an example for an operating system of this class. Actually, PURE provides 6 members of a kernel family supporting for different trade-offs for functionality and overhead. PURE

Third, systems support customisation at the level of functionality. A representative for this class is VIRTUOSO [143, 144, 145]. In VIRTUOSO the user can choose among four levels: interrupts are not supported, interrupts are supported, light-weighted threads can be scheduled by a Round-Robin scheduler, or a preemptive priority scheduling is available. The choice of a level means that higher levels completely comprise lower levels. VIRTUOSO

Fourth, customisation is applied at source code level. The ECOS system [35] is a representative for this class. In ECOS the integration and selection of appropriate source code into the system is chosen at compile-time by setting appropriate pre-processor macros. This approach is very similar to that in DREAMS [41, 42] (see also Section 6.6.1.1$_{127}$). ECOS

Fifth, customisation can be made at target code level. This normally means that the software reconfigures its code at run-time. SYNTHESIS [87, 88] adapts its code by partially evaluating and recompiling condition statements depending on available input data at run-time. This can result in changing compare instructions and conditional jumps by unconditional jumps and vice versa. This eventually leads to the elimination or integration of complete code fragments. SYNTHESIS

Most of the systems do not provide customisation tools or assistants. They rely on the talent of the users to create the best configuration of the system. Only few systems support customisation frontends that assist the user in creating a configuration. VIRTUOSO can be customised with the help of SoftStealth^{TM}. SoftStealth allows to select or to remove particular levels of the kernel automatically considering dependencies between the levels. ECOS comes with a configuration tool that allows to select or deselect particular components. The selection works in the same way as known from setup programmes of huge software packets like Windows^{TM}, Microsoft Office$^{(R)}$ or Linux^{TM}. In dialog boxes features can be selected, deselected, or chosen from a set of items in a graphical manner. Outstanding is that dependencies between the components are automatically considered. This means, for example, that the selection of a particular component includes the selection of all required sub-components. However, the choice among alternative items still must be done by the user. This results in setting up a lot of configuration items. Furthermore, the user must exactly know the meaning of each item in order to have the chance to select the best alternative.

TEReCS intends to support the user in this way. Only knowledge about the application has to be specified instead of directly customising the system. So, required knowledge about internals of the execution platform is reduced to a minimum. The configuration tool of TEReCS implements rules to translate application properties into configuration options for customisation. These have to be specified by experts creating the TEReCS knowledge base. This issue is handled in Chapter 6.4$_{112}$.

3.5.3.1 Survey

LINUX In LINUX the user can insert or remove special services into the system. But why a service must be integrated or which service should be integrated (see example "Resource Protection" and "Device Drivers" on page 48), is left to the user. Only operating system immanent dependencies are considered more or less automatically (see example "Service Dependencies" on page 48).

CHOICES CHOICES [26] is one of the first operating systems, which configures the application and the operating system specific functionalities together resulting in mutual influences. The kernel for a node of the system is selected during boot-time.

SYNTHESIS SYNTHESIS [101] belongs to the group of so-called *adaptive operating systems*. Configuration takes place during the run-time of the system. Therefore, code of the operating system will be compiled and linked into the kernel while running. Thus, a run-time compiler is included in the operating system. This approach is based on the idea that dynamically available data of the application can be exploited for the operating system by *partial evaluation* of application code and its actual input data, i.e. statistical information about the input data is used to optimise the code. So, condition checks, jumps and, therefore, dead code of, for example, `if-then-else` statements can be removed by the compiler. This procedure will be activated whenever new information about the input data is available. This mechanism is not restricted to the application code, but also applied to parts of the operating system itself (*interrupting*, *context switching*).

APERTOS APERTOS [149] is designed in an object-oriented manner. It belongs to the group of so-called *reflective operating systems*. Each configurable object has a companion in the form of a meta-object. The meta-object observes the actual object implementation and its status and replaces the functional object by another implementation out of its assigned object hierarchy when certain conditions are true. This means, the exchange of behaviour is the result of the exchange of an object's implementation triggered by the assigned meta-object. By this way functionality and implementation are strictly distinguished in this meta-object/object hierarchy. All objects belonging to a specific meta-object are spanning the meta-object's design space and implement the same functionality.

CHAOS Inside the CHAOS system [49] the operating system functionality is adapted to the actual real-time constraints. The complete system is transaction-based. The language COLD, which specifically was designed for CHAOS, describes the control flow of the application by defining nested transactions.

EXOKERNEL EXOKERNEL [44] basically consists only of a driver for the processor supporting *context save/restore*, *interrupting* and functions for *status evaluation and change*. Extensions to these can be loaded as pre-compiled code into the kernel during run-time by dynamic linking. The extensions will be picked from a library.

VXWORKS A very well-known and widely used commercial operating system is VXWORKS™ [148], which was specially designed for embedded systems. It is built highly modular. For a lot of operating system services multiple alternative implementations with different behaviours exist. A configuration of the operating system is to be designed manually under a graphical user interface (Tornado Design Suite) during a so-called setup phase. That way modules like schedulers, synchronisation primitives, drivers, special hardware services, etc. can be selected from a list and thereby integrated into a final configuration.

Dependencies between the modules, in the meaning that a module A only works, if module B is also selected, are automatically checked by the tool. Additionally parameters for the modules are assigned by filling out special dialog boxes. Such parameters can be the base address, where a driver can find its device, the interrupt number, baudrate, etc.

PEACE

The PEACE operating system [11, 113] was originally developed for massively parallel architectures. The design of PEACE is extremely fine-grained. It supports the transparent loading of higher services during run-time. PEACE is extraordinary in many respects: An incremental loading mechanism for a multi-processor architecture is combined with a distributed object management system. Objects can be instantiated on every node and then they can communicate with other objects by *Remote Object Invocation* (ROI) [95]. PEACE is built on basis of a variety of alternatively selectable μ-kernels. A consequence of this kernel family concept is that each node of the system can have other operating system characteristics. The kernels mainly differ in their functionality, i.e. if they support for multi-tasking, memory protection, synchronisation, local or external communication, ROI, etc. Which kernel has to be used for a node is determined by explicit selection during boot-time of the node.

PURE

PURE [114] is quite new library-based construction kit for operating systems. It has been developed from its predecessor PEACE. Therefore, PURE has a lot of similarities with PEACE, e.g. the kernel family concept. But the so-called *nucleus family* of PURE consists of 6 members, which were especially developed for embedded systems. They differ in their functionality and the overhead they spend. The nucleus family is organised in nearly 100 classes and 14 levels. The simplest family member merely supports for asynchronous interrupt handling. Additional functionalities are integrated step-by-step on higher levels comprising synchronisation, multi-threading, preemption, etc.

AMBROSIA

The real-time operating system AMBROSIA [99] had especially been developed for controllers in the automotive industry. Its exceptional property is its possibility to scale its functionality on two levels. By the selection of required modules it is *macroscopically scaled*. Additionally, the modules itself can be adapted to the requirements of the application. This mechanism is called *microscopic scalability* there. In order to support for these scalability features, the modularisation language AML and a configuration tool EOS had been developed. Although AMBROSIA is called to be a real-time operating system, it only implements a heuristic priority-based preemptive scheduler. The priority inversion problem is not considered. Likewise, there is no support for the priority ordering of the processes. In contrast to this, it supports for the periodical activation of processes. Additionally, timer handling for temporary process deactivation is implemented. So, processes can sleep for a relative period or until an absolute time. For inter-process communication AMBROSIA supports for a rendezvous mechanism for synchronisation purposes or an asynchronous procedure. The communication is purely packet-oriented. Concrete hardware drivers only exist for serial communication via RS-232 or via a CAN bus, which is widely used in automotive applications.

DREAMS

DREAMS[7] [39, 40, 41, 43] is a library-based construction kit for embedded operating systems and run-time platforms. It has been developed during several projects and had been completed during a Ph.D. thesis [42]. The complete system had been designed object-oriented in C++. Operating systems and run-time platforms for even heteroge-

[7] **D**ist**r**ibuted **R**eal-Time **E**xtensible **A**pplication **M**anagement **S**ystem

neous processor architectures can be constructed from customisable components (*skeletons*) out of the DREAMS's kit. This construction process is also called "configuration process" in DREAMS, although it is originally done completely by hand. This process is done a priori during the design phase of a system. By creating a configuration description all desired objects of the system have to be interconnected (by *Inheritance*) and afterwards fine-grained customised (by defining *Aggregation*, *Linking*, *Membership*, and *Life-Time* of the skeleton objects). For details of the customisation features of DREAMS please refer to Section 6.6.1.1$_{127}$. The possibility to customise objects had been integrated into DREAMS on the basis of the pre-processor and was named *Skeleton Customization Language* (SCL). The primary goal of that process is to add only those components and properties that are really required by the application. Therefore, a run-time library is built instead of a kernel. The construction kit DREAMS strictly distinguishes between optional and mandatory components. The creation of a final configuration description for DREAMS has been automated during the DFG project TEReCS [8] [14, 15, 16, 19] and this thesis. During that project a methodology was developed in order to synthesise and configure the operating and communication system for distributed embedded applications. The basic idea for TEReCS is the description of services and their dependencies among each other and to the hardware of the system. The configuration descriptions have to be generated for the operating system of *each* node of a distributed system. Thereby, only those services must be integrated into the run-time platform that are really required by the application and that produce minimal costs in terms of memory consumption and execution time.

3.6 Advantages of Configuration within a Real-Time Communication System

Also within the real-time communication system configuration makes sense. Services like routing, (de-)fragmentation of messages, congestion control, error correction, acknowledging or bandwidth management are parts of the communication system. Considering the OSI model [66] of the ISO these services are implemented as layers of a protocol stack. Each entity of a protocol layer may depend on a lower level and requests for an appropriate implementation on the other side of the communication line. Each protocol layer consists of services for sending and receiving, which forward the data downwards (to the hardware) or upwards (to the application) in the stack. Whether a service (or layer) is requested and how it can be implemented best, is mainly determined by the Quality-of-Service that is requested by the application and by the properties of the underlying hardware of the network. Nevertheless, nearly the same type of rules can be applied for the configuration of those communication services of a protocol stack. A side effect will be that the protocol stack will become more flexible. Not only the integration or exclusion of layers and, therefore, functionalities can be achieved, but also more flexible combinations can be obtained. Their order can be changed or layers of originally independently designed protocol stacks can be easily combined, because configuration demands for compatible interfaces or ports for alternative implementations (see "behaviour-based configuration" on page 37).

The potential of configuring the communication system is as high as that of the operating system. While the present task set on a processor and the present hardware determines

[8] **T**ools for **E**mbedded **Re**al-Time **C**ommunication **S**ystems

the required characteristics of the operating system, this is also true for the communication connections established between all the distributed processes. But also the routing, which means the resource allocation for the data transfer on its way from the source to the destination, has a huge effect and influence on the required communication services per node of the network.

An optimisation criterion for the creation of a good configuration is not only the mapping of the processes onto the processors, but also the routing of the messages through the network. But routing is an integral part of the communication system. Thus, resource allocation must be taken into account during its configuration.

3.6.1 Goals of Communication System Configuration

The primary goal for configuring the communication system is to minimize the system overhead and the delays of the communications. For real-time applications it is sufficient to assure that no delay leads to a deadline miss (see Section 4.4 $_{76}$). Nevertheless, the demand for a minimum system overhead is universal.

The system overhead is mainly determined by the operating system's implementations of the communication drivers and protocols. Which protocol (respectively, which protocol driver) has to be selected, is not trivial. This strongly depends on the requirements of the application and the available communication device. For instance, on the one hand, the application may require for encryption of the messages. This request for a protocol layer to code and decode the messages. On the other hand, a low level protocol, for example, one for error detection, can already be installed in hardware inside the communication device. Thus, it must not be installed as a software protocol layer service.

3.6.2 Examples

Further examples for the request to configure the communication system are the following:

Resource allocation for the routing of the messages. The routing of the messages through the interconnection network must be considered during configuration. On each node, which has to forward the message, resources are required in order to fulfil this service. Especially, one or more routing tasks are required, that receive the messages and send them out on the appropriate link. Often, a router task per incoming connection exists. The configurator has to establish these tasks. Additionally, these router tasks must be considered during the timing analysis, because they require processing time and cause buffer delays for the message transfer. The configurator possibly can eliminate such tasks, when the routing is not required for specific paths. That way, resources can be saved.

Meet application demands for the message transfer and storage. The application can require special services for the message transfer, like encryption, error correction, acknowledgments, order preserving, synchronous or asynchronous sending, buffering, etc. These can be modelled as requirements that have to be matched during the configuration process. Therefore, the configurator has to include appropriate protocol drivers into the sender's and receiver's protocol stacks.

Implementation of implicit constraints. These constraints arise, when there is a gap between the application demand for the message transfer and the physical capability of the link and media. For example, the packet size, which can be transfered by a specific hardware link, can be smaller then the message, which is to be transfered via that link. Then, the message has to be split into several packets at the sender's side. At the receiver's side the packets have again to be assembled to the original message. This means that a fragmentation and de-fragmentation protocol has to be integrated into the protocol stacks.

Combination of different protocol layers. Inside a heterogeneous interconnection network, the links vary in their physical hardware and require different (low-level) protocols for the message transfer. For instance, serial connections, CAN busses, Ethernet lines, token rings, etc., can be mixed. When a messages is to be routed through such a network, then the message has to pass different protocol stacks. Then, a kind of bridge is required, which receives messages from one protocol stack and sends the message to another protocol stack. This bridge works similar to a routing service, except that it receives and sends messages on different protocol stacks. Therefore, they embody a protocol layer conversion service. The configurator has to assure the interface compatibility between the different protocol stacks.

Compatibility of the partner's protocol stacks. The configuration process have to assure that the protocol stacks of the sender's and receiver's sides of every message transfer are compatible.

Interference with the operating system. The protocol drivers for the message transfer have multiple interconnections to the operating systems. The previously mentioned router or bridging tasks have to be appropriately scheduled by the local operating systems. The configurator has to specify correct priorities, deadlines , periods or activation intervals for those tasks upon the messages' interarrival times (message periods). Some communication devices produce interrupts, which have to be served by the operating system. Sometimes, the drivers have also to poll the devices. Both, interrupting and polling, produce system overhead and require special services (interrupt or timer management). Additionally, messages have to be stored locally, before they are forwarded or delivered to the receivers. This means, that memory buffers for the temporary storage of messages have to be created and managed.

Bandwidth allocation or time slot reservation. Bandwidth allocation or the reservation of time slots for the message transfer (in classical communication known as *congestion control* and *media access*) are indispensable for real-time communication. The routing algorithm and the real-time analysis for the communication both must deal with these problems. Therefore, it is recommended that both modules are part of the configuration phase. When the media access is not possible or leads to delays that produce not feasible schedules, then the configuration is invalid. Thus, another configuration (possibly with another routing) has to be created.

3.6.3 State of the Art

In some communication systems, which are mainly part of a real-time operating system, (offline) configuration is used in order to create appropriate and efficient operating system support and to optimise the overall system during a detailed analysis (see

AMBROSIA/MP). Other systems use the configurability in order to adapt ports and to connect them appropriately (see CHIMERA). Yet in other systems (like REGIS) the layers of an application specific protocol stack become configurable. Specific protocols can be integrated or removed from the stack, whereas the protocol stack compatibility of the sender's and receiver's site are automatically maintained. In the following subsection these systems will be described in more detail.

3.6.3.1 Survey

AMBROSIA/ MP

Ingo Stierand developed the AMBROSIA/ MP extension [135] to the operating system AMBROSIA (see page 51). It extends AMBROSIA by the possibility to generate the operating system and the interconnection network for a distributed application. It defines an execution and communication model for simultaneously running processes on a multiprocessor (MP) platform. It solves the problem of the process placement and the routing of the messages. Therefore, a detailed schedulability analysis of the processes and the communications is integrated. Blocking times due to the use of global semaphores and the communication of messages –unfortunately only of fixed length– are considered. The optimisation of the placement and the routing problem are solved by a genetic algorithm.

A simple static process model with fixed priorities is used in AMBROSIA/MP, so that the simple rate monotonic (RM) scheduling approach can be used for each processor. Only, the linear ordering of the priorities had been weekend to a partial ordering. For a process a priority interval is defined, in which its final priority will be. For each processor, a strict order for the priorities of the processes is defined in order to apply the Rate Monotonic scheduling (see page 72). But the priorities of processes of different processors define no order. Instead a precedence graph is defined according to the communication dependencies.

Stierand defines a global schedule to be feasible, if, and only if, all response times R_i of each process τ_i are lower or equal than their deadlines D_i. He slightly defines the response time different to the classical approach:

$$R_i = C_i + B_i + \underbrace{\sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j}_{\forall \text{ processes } j \text{ with a higher local priority}} + \overbrace{\underbrace{\sum_{k \in ep(i),\, k \neq i} C_k}_{\forall \text{ processes } k \text{ with the same local priority}}}^{new}$$

where C_i is the execution time and B_i is the blocking time of task τ_i. The blocking time B_i is defined by the maximum hold time $z_{i,j}$ of any semaphore σ_k with higher ceiling priority than the priority of process τ_i and, which is used by any process τ_j that has the same or a lower local priority:

$$B_i = \max(z_{j,k} \in \beta_i)$$

where

$$\beta_i = \{z_{j,k} \mid j \in lp(i) \overbrace{\cup\, ep(i) \setminus \{i\}}^{new} \wedge\ ceiling(\sigma_k) \geq P_i\}$$

Stierand also considers the operating system overhead for acquiring and releasing of semaphores. The communication time, which is used for the *remote procedure calls* when

global semaphores are used, is also considered:

$$C_i = C_i^{COMP} + \underbrace{\sum_j n_{ij}^L (C_i^{L\_P} + C_i^{L\_V})}_{\text{use of local semaphores}} + \underbrace{\sum_k n_{ik}^G (C_{ik}^{G\_P} + C_{ik}^{G\_V})}_{\text{use of global semaphores}}$$

$$C_{ik}^{G\_P} = C^{RPC\_IN} + C_{ik}^{SEND} + C_k^{L\_P} + C_{ki}^{SEND} + C^{RPC\_OUT}$$
$$C_{ik}^{G\_V} = C^{RPC\_IN} + C_{ik}^{SEND} + C_k^{L\_V} + C_{ki}^{SEND} + C^{RPC\_OUT}$$

For the analysis of the communications AMBROSIA/MP defines uni-directional communication channels between two communicating processes. It is assumed, that all messages over the channels have the same size. This restriction was made due to the fact that AMBROSIA only supports communications via a CAN bus. CAN messages can contain a maximum of 8 bytes. Outstanding is that AMBROSIA/MP handles communications, which have to be routed over several nodes (hops) when no direct link exists between the nodes where the communication partners have been placed. It is also considered that the communication capacity (bandwidth) of a link is limited. For a successful routing it has to be checked, whether no link is overloaded and all communication delays are lower than their defined maximum deadline: $R_{ij}^{SEND} \leq D_{ij}^{SEND}$.

The modelling and implementation of the communication is made with buffers (mailboxes) and dedicated communication processes that forward and receive the messages over the communication links. There, all user processes, communication processes, buffers, and communication links are modelled as the nodes of a communication graph. Its edges define the communication paths and will be labeled with the required capacities. The capacities and buffer sizes are determined during the optimisation phase. For the reservation of the bandwidth a special token protocol is implemented.

During the optimisation phase the mapping of the processes onto the processors and the routing of the messages over the communication links is determined. Additionally, the local priorities of the processes, the activation intervals of the communications for the token protocol and the required buffer sizes are calculated. The placement and routing problem is defined by the DNA of a genetic algorithm. Therefore, Stierand also defines a quality function, which represents the quality of a solution. It is important for the genetic algorithm that the quality function also ranks impossible solutions. For instance, the quality function will just give a negative value, when any deadline is not met. The overall quality function is the weighted sum of the three quality functions that rank the process mapping, the priority assignment and the routing.

After a feasible mapping and routing is found, the operating systems of each processor node in the system have to be configured. A configuration is described by the instantiation of components for the operating system in the AMScribe language. For each user and communication process a process component will be instantiated on the appropriate node. For each message a routing component with the routing information as a parameter is instantiated on every node, over which the message is routed. Additionally, on every node some system components are instantiated, which represent all the system calls to the operating system. A system call can be the sending or receiving of a message to or from a communication controller, as well as the acquiring or releasing of a local semaphore. Whether a component must be instantiated or not, is simply evaluated over a logical existence term. For example, the code for the semaphore σ_i has to be integrated

into the operating system for processor P, when there exists a process τ_i on P that uses σ_i. A component can also be instantiated more than once. The instantiation can be relative to the amount of already installed objects of a concrete type. For instance, assume there already exist four objects of type S_1 and three objects of type S_2 in the configuration. The following definition

`COMPONENT C IF (EXISTS` S_1 `and EXISTS` S_2`) THEN PAR(a) END;`

results into the integration of 12 instances of the component of type C, where the parameter `PAR` of all components will be set to a. This is done, because for all combinations of instances of the types in the condition the instantiation is made.

The automatic calculation of the process placement and the routing are two positive aspects of AMBROSIA/MP. Limitations of AMBROSIA/MP are the fixed message size and the use of a fixed communication protocol. Also, only four system calls, which are the `acquire()` and `release()` of a semaphore and the `send()` and `receive()` via a CAN controller, are handled by the analysis.

CHIMERA

The CHIMERA Methodology [132, 133, 134] is a software engineering paradigm for the implementation of real-time software for multi-sensor systems. It supports the development of dynamically reconfigurable component-based real-time software. CHIMERA combines the *port automaton theory* with object-based (not object-oriented) design, which results in objects that are called *port-based objects*. These objects have input, output, and resource ports. Each object represents a single task. The communication between local instances is done by connecting output ports to input ports. External communication is done via resource ports, which access appropriate devices. The ports are named and only such ports with identical names can be connected. A final configuration of all participated objects is legal, if, and only if, all input ports are connected to only one output port. It is allowed to fan an output to multiple inputs. In CHIMERA exists *join connectors* (with the semantic of calculating the average) in order to combine several outputs to one input.

All communications in CHIMERA are done via a *global state variables table*. The port automaton theory assumes that the most recent data is always present at the input ports. Message queues are *not* appropriate, because they accumulate the data in their queue and the next data retrieved is not the most recent received one. So, connections of ports are implemented by state variables. But reading or writing state variables by asynchronous parallel tasks cause integrity problems, which must be solved by using critical sections during the access. These synchronisation mechanisms cause dependencies between the objects. These dependencies are not allowed in the port-automaton model (autonomous execution). Therefore, each port-based object gets its own copy of its referenced state variables in a local state variables table. The key is to ensure that updates between both tables are done only, when the task does not access its local table. Updates are done, when the task is not executing. This way, CHIMERA uses an *inside-out method* of programming, rather than the software invokes the operating system via system calls.

The CHIMERA real-time operating system provides tools to support the software models defined by the CHIMERA methodology, so that real-time software can be executed predictably using common real-time scheduling algorithms. The operating system services are always executing and invoke methods of the port-based objects. By this way, programmers have not to take into account communication and synchronisation. The

model assumes port-based objects can be in one of four states: NOT CREATED, OFF, ON and ERROR. The state transitions are results of signals, which are sent to the tasks, like: `spawn`, `on`, `wakeup`, `off`, `re-init`, `kill`, and `clear`. Each transition is done via calling a method of the object: `init()`, `on()`, `cycle()`, `off()`, `kill()`, `error()`, `clear()`, `re-init()`, and `sync()`. Before and after each method call the table data are updated, distinguishing in/out constants and in/out variables. A two-step initialisation and termination is used to support dynamic reconfiguration. High overhead initialisation and termination code is performed during the `init()` and `kill()` methods, whereas tasks can be activated or suspended quickly using the `on()` and `off()` methods.

DAVINCI The tool suite DAVINCI [142, 147, 146] from *Vector Informatik GmbH*, Germany, supports the developer of distributed automotive applications. The Vector Informatik GmbH has great knowledge and experience in this area and especially with tools for the design and analysis of communication infrastructure and software for the CAN bus. DAVINCI seems to be the most recent commercial development in the area of design and communication support for distributed embedded control applications. DAVINCI supports the reuse of control software, the exchange of data via different networks, and the automatic design of the run-time environment for the distributed embedded control units. DAVINCI includes a methodology for the developmnet of control applications. The development of the application software is based on the specification of *software components*. A software component encapsulates the control algorithm, which can be specified in form of a finite-state-machine or in native C code. All information flow into the component and from this component is modelled by in/out signals. A software component is handled as a black box. The input and output signals can be connected with signals of the same type of other components. Then the components form a network. Additionally, they can build a hierarchy. A component can consist of sub-components. Hereby, the signals of the super-component are redirected to the signals of its sub-components. A complete network of sub-components can be integrated into a super-component.

In order to develop the overall functionality of an automotive software components have to be assembled to a complete software application. Hereby, the components must appropriately be connected to each other. Still open signals have to connected to so-called *Device Accessors*. The device accessor represents an abstract sensor or actuator.

Before the complete code of the automotive application can be generated, the topology of the hardware platform must be described. Hereby, the different available embedded control units are described together with the available actuators and sensors. Also the available communication busses between the controllers and sensors/actuators have to be described. Actually, DAVINCI supports CAN and LIN.

In a third step of the applicaion development the software components have to be mapped onto the available embedded control units. This defines precisely which signals have to transferred via messages over the buses and which signals can be exchanged by inter-process communication of the operating system. DAVINCI automatically incorporates appropriate decice driver code for the communication buses, as well as appropriate system calls to the operating system. The priority of the messages and the transfer modes and protocols are automatically determined. Also the firmware in order to access the sensors and actuators is generated. It is worth to note that a sensor/actuator can be placed on another controller, as on which the software component is mapped, which uses this sensor/actuator.

DAVINCI also automatically configures an OSEK compatible operating system for each embedded control unit. DAVINCI incorporates a design methodology for closed distributed control applications for automotives. It is the goal of DAVINCI to find and to define reusable and sharable software components. The communication matrix of the components will be automatically generated. The application specific code for each embedded control unit is also automatically generated. The bus communication and also the access to sensors and actuators is generated. The operating system of each embedded control unit is parameterized and configured in order to run the required tasks and communication drivers.

Thus, DAVINCI shares the ideas of the automatic code generation and the reusability with TEReCS. However, the automatic check of the timing requirements of a real-time application is not done. Nevertheless, DAVINCI embodies a variety of diagnostic interfaces and functionalities, which allow to test the complete system or a component offline on a PC or during run-time on the embedded control unit. But the configuration of the operating and communication system is limited. The configuration aspect is far away from the fine-grained approach that is used in TEReCS.

REGIS

The REGIS system [100] is a distributed programming platform which adds protocol stacks to communication endpoints. The system is more general and not only applicable to real-time applications. It uses standard transport layers (like UDP/IP or ATM) as the basis for data transportation. The basic (low-level) protocols and devices are still managed by an operating system. Its key task is to describe a user-defined protocol stack by a graphical or textual representation. This is done in the language DARWIN. The basic idea is a strict hierarchical protocol stack through which data travel up- or downwards. Each layer has a well-defined interface. REGIS distinguishes between provision and requirement interfaces. Possible connections are only allowed between a provision and a requirement interface. By this means, REGIS defines services that have providing and requiring dependencies between protocol layers. The system is implemented in object-oriented C++. One of the main advantages is the support for dynamic protocol stacks. Layers in the hierarchy can be introduced or removed during the lifetime of a connection. This provides for dynamic binding of protocols during run-time. Predefined layers exist for sequencing, time-out, fragmentation, etc., resulting in the support for reliable protocols. Code reuse is supported on the layer level.

3.7 Contribution of the Chapter

This chapter has characterised the problem which is solved by configuration. The configuration aspect is central to this thesis. Therefore, some of the advantages and disadvantages of the existing approaches have been discussed. The different principles for the configuration have been introduced. This should help to select or adapt an appropriate algorithm for the automatic configuration of operating systems and one communication system. This chapter should also serve as a motivation for the idea that configuration is appropriate for the construction of such systems.

CHAPTER 4

Real-Time Analysis

> . . . a sufficiently fast computer can satisfy the (timing) requirements, hence real-time computing is equal to fast computing. This is wrong.
>
> John A. Stankovic, 1998 [125, p. 4]

Subject to this thesis is the configuration of embedded real-time operating and communication systems. For this reason, *"real-time"* plays a crucial role for the correctness of the system. It is essential that the overall configuration process considers the real-time aspect. This chapter will give a brief introduction into the theory of real-time analysis. After *real-time* is defined the basic approaches in the research for the analysis of the process scheduling are presented. The main goal that is handled by the schedulability analysis is the verification that all tasks will execute completely within a fixed and specified interval. Different assumptions about the task set's properties lead to various algorithms that assure this timeliness execution. After the basic concepts of the process scheduling are presented the main approaches that have been transfered to the communication case are briefly explained. For a nearly complete overview about this topic also the server approach for the process scheduling of sporadic tasks and the synchronisation problems due to resource access constraints are briefly described.

It should be mentioned that this chapter only presents the algorithms and achieved results. The proofs for their correctness will not be given here. For these the interested reader will be referred to the literature that is referenced in the appropriate sections.

4.1 Real-Time versus Non-Real-Time

Normal "visible" computers are equipped with an user interface (display, keyboard, mouse, joystick, touch-screen, etc.) and mainly run applications for the user's sake. Often, they present calculations or retrieval results or final layouts to the user according

to some given inputs. In order to hold the *response time* for their reaction to the user's input as short as possible, their computing power has to scale accordingly to the problem's complexity. The performance of such computers is measured by the execution time in which they produce their results. In the case of serving multiple applications or user requests in parallel (or semi-parallel) fairness and throughput become additionally important. For all of these reasons a lot of resources are given to such computers (many CPUs, high clock frequencies, huge memory, high speed communication adapters, etc.).

These characteristics become less important in the field of *real-time computing*. Real-time applications are mainly implemented for "embedded systems". An *embedded system* typically consists of a *controlling system* and a *controlled system* [125, p. 1]. Often, the controlling system is not visible. For example, the computers inside a washing machine or the motor management for the engine inside an automobile are not directly visible to an user of the controlled system. They are "embedded" into a technical system. Thus, the controlled system can be seen as the environment for the computer. The controlling system's main purpose is the interaction with the controlled system. Thereby, it supports the controlled system in achieving its prime function.

The interaction of the controlling system with its environment is done by observing and manipulating. Information is collected via several attached sensors. Upon this information a representation of the state of the environment is created inside the computer. Based on this representation reactions to these, so-called *stimuli*, are calculated. These reactions are manifested by generated output values, which are transmitted to the controlled system via actuators. Therefore, such a controlling system is also called *reactive system*.

Often, the phases of reading information, calculating reactions and writing output values is periodically done. Moreover, the *controller* is embedded into a closed loop, because its output values have directly physical impact on the controlled system. Hence, the status of the controlled system normally changes and new input values have to be read. In order to create correct control signals it is imperative that the state of the controlled system, as perceived by the sensors, is consistent with the actual state of the real environment. This implies that the internal time representation of the controlling system is equal to the external or **"real"** time of the environment.

Moreover, it is essential for real-time systems that their reactions take place within a certain time period or before a certain time limit expires. Thus, an upper time bound for their response time exists, which is called *deadline*.

Systems or applications can be distinguished in *non-*, *soft-*, *firm-* or *hard-real-time* systems. They are classified by the utility that the output values have. The generation time of an output value is thereby related to its deadline [25, p. 230-232]. In non-real-time systems the utility stays constant over the time. In a soft real-time system the utility function stays constant until the deadline is reached. Then, it decreases linear to zero. That means, after the deadline is reached, the results become more and more useless until they expire totally. In a firm real-time system the utility function or usage of the values immediately becomes zero after the deadline is reached. In a hard real-time system the utility function goes immediately towards the negative infinity. This means that the profit or usage of the values turns into impairment, damages or defects of the controlled system and, maybe, of its environment. Often, this means that catastrophic consequences can result from missing a deadline. For instance, the system crashes or it seriously damages its environment or even humans. In soft or firm systems it might be tolerable that a deadline is missed.

Typical examples for hard-real-time systems can be found in transportation vehicles like automobiles, trains, aircrafts and ships in their power-train control, engine control or in systems like steer-by-wire and brake-by-wire. Mostly, malfunctions of these systems obviously lead to severe problems, which are menacing even human's lives. A malfunction in real-time systems happens, when an actuator is not set in its predefined time window accordingly to the required value due to timing problems in the calculation path.

The weather forecast or the decision on stock exchange orders can be seen as firm-real-time systems. If their results are produced too late, then their usage might be worth nothing. Moreover, it is widely accepted that this can happen, respectively that their results can be wrong.

In the field of transmitting real-time video or audio data *soft* deadlines are defined. If the data are received too late, then the display or their play back might be disturbed. This results into jitters or artificial artifacts. But nevertheless, the information that is contained in the stream is transmitted. Only the quality of the stream might be decreased. In this context the term *Quality of Service* (QoS) is often used in order to specify the probability or the estimated average amount of probably missed deadlines (and its standard deviation and related values).

It is the primary goal of real-time systems that all tasks meet their deadlines. And it should be mentioned clearly, that the fact how fast they execute is (nearly) of no interest; but that they will *never* reach the deadline! Ditze [42, p. 10] stated:

> *"Timeliness is the dominating QoS requirement here, which fundamentally differs from the aim of HPC applications. Performance and fairness properties are regarded less important."*

Buttazzo summarizes the basic properties that real-time systems must have to support critical applications [25, p. 12]:

Timeliness. Results have to be correct not only in their value but also in the time domain.

Design for peak load. Real-time systems must not collapse when they are subject to peak load conditions, so they must be designed to manage all anticipated scenarios.

Predictability. To guarantee a minimum level of performance, the system must be able to predict the consequences of any scheduling decision. If some task cannot be guaranteed within its time constraints, the system must notify this fact in advance, so that alternative actions can be planned in time to cope with the event.

Fault tolerance. Single hardware and software failures should not cause the system to crash. Therefore, critical components of the real-time system have to be designed fault tolerant.

Maintainability. The architecture of a real-time system should be designed according to a modular structure to ensure that possible system modifications are easy to perform.

In order to achieve the timeliness, hard real-time systems are designed under the assumption that everything is known what can happen and when it happens. This means

everything is predictable. In fact, that all tasks will meet their deadlines under these assumptions, will be predicted (verified or falsified). More in detail, this is done by a so-called formal *real-time analysis*. In the past decades a lot of research have been done in this field. Because the configurator, which had been developed in this thesis, deals with real-time communication, also a simple real-time analysis approach has been integrated into TEReCS (see page 133). For this reason in this chapter some basic and important research results for the real-time analysis will briefly be presented.

The request for maintainability of a real-time system can easily be handled by configuration. Configuration is an appropriate way in order to maintain all options of a customisable system. But configuration and real-time analysis must be integrated. Both they cannot be seen isolated. Real-time analysis should steer the configuration or the configuration must qualify its solution by an analysis. This is ensured in TEReCS (see Section 6.9$_{142}$ and Section 6.10$_{147}$).

4.1.1 Definition of Real-Time System

Because the configuration of a real-time system is the main topic of this thesis, a definition of a real-time system will be given. In order to clarify the sketched picture of the previous section a few definitions from researchers of the real-time community are presented here. Often Young is cited, who defined a real-time system as

> *" . . . any information processing activity or system which has to respond to externally generated stimuli within a finite and specified period."* S. Young, 1982 [150]

A quite similar definition had been provided later by Randell:

> *"A real-time system is a system that is required to react to stimuli from the environment (including the passage of physical time) within time intervals dictated by the environment."* B. Randell, 1995 [105]

These two definitions define a real-time system by its reactions to external stimuli and – that is important – by the definition of a time bound (*deadline*), which defines when this reaction has to take place after their appearance. Stankovic [122] and Kopetz got the heart of these properties:

> *"A* real-time computer system *is a computer system in which the correctness of the system behaviour depends not only on the logical results of the computations, but also on the physical instant (*deadline*) at which these results are produced."* Hermann Kopetz, 1997 [72]

It should again be realised that the timeliness is an essential property of real-time systems. Moreover, the system's behaviour might be incorrect, if some deadlines are missed. Because timing faults may cause severe damages or injuries, the correctness of a real-time system must be guaranteed. The deadlines define *timing constraints*, that the system must satisfy. The correctness is formally verified by a so-called *schedulability test*. This test checks, whether for all possible execution orders of the system's processes and threads the reactions will be before their deadlines. This test can only be done, if the *worst-case*

execution time of the processes and threads, their *communication delay* and their *resource* and *precedence constraints* are known in advance (are predictable). *Resource constraints* arise because of *mutual exclusion* and *precedence constraints* arise because of output-to-input dependencies. Normally, this test is done for static systems (i.e. changes of these assumptions do not occur during execution) before their run-time (offline). If these assumptions change during execution, because new processes have to be started or their execution time or their resource and precedence constraints change, then this system is known to be dynamic. Then, the so-called *acceptance test* for new processes or constraints has to be done during run-time (online).

An overview about these schedulability and acceptance test will be given in Section 4.2 and Section 4.4$_{76}$.

4.2 Real-Time Analysis for Process Scheduling

In the past decades excellent results are produced in the area of *real-time analysis* for process scheduling. Due to the fact that the configuration developed in this thesis also deals with a complete customisable operating system and a lot of the results can –more or less– directly applied to real-time communication, some of the basic and most important results will shortly be introduced in this chapter.

For *predictable real-time systems*, i.e. all of its properties are well known, various algorithms had been developed in order to proof formally whether all tasks will meet their deadlines under all circumstances. In detail this means that each process of the system will terminate before its deadline. Therefore, the order of execution of each task is constrained in a certain manner by the task scheduler. Nevertheless, a lot of remaining possible execution orders exists. It is the task of a so-called *schedulability test* to verify or falsify upon the given properties and constraints, if a concrete and given set of processes will meet their deadlines or not. Input to this test are the task set and all of the system's properties. The test will mostly be done offline before the system is implemented. But if the system's properties will change during run-time or the task set will change, then the test must be done online during execution in order to allow the change or not. Only if changes do not lead to deadline misses, they can be accepted. Thus, the online test is often called *acceptance test*.

As previously mentioned the execution order of the tasks is constrained. That means that the scheduling of the task set underlies some restrictions. For this reason a variety of scheduling algorithms had been developed. The scheduling algorithm and its schedulability test should guarantee the timeliness execution of a given task set (*feasible schedule*). This must formally be proofed. An algorithm and its schedulability test are only correct, if under all circumstances and for all possible task sets a feasible schedule will be executed when the schedulability test accepts the task set.

This means that real-time analysis defines proofs on two levels: On the higher level the scheduling algorithm and its schedulability test must be verified and on the second level the formal schedulability or acceptance test is identifying only valid task sets. A scheduling algorithm and its schedulability test are called to be optimal, if they accept the task set when there exist at least one valid schedule so that all deadlines are met and the scheduler will execute this schedule.

4.2.1 Definitions

Before some analysis approaches for real-time systems will briefly be presented, we will give some definitions of characteristics and parameters of a real-time computing system. Most of these are timing constraints or properties because timeliness is the crucial characteristic of a real-time system. Processes or tasks are the basic items, which carry out the calculation paths of the implemented algorithms. Therefore, a set of tasks $\Gamma = \{\tau_i \mid i = 1, 2, \ldots, n\}$ is identified to be subject of the analysis.

The following properties of a task τ_i are of interest for the real-time analysis:

- **Arrival time** a_i: is the time at which the task becomes ready for execution; it is also referred as *request time* or *release time* indicated by r_i.
- **Computation time** C_i: is the time necessary to the processor for executing the task without preemption. The *worst case execution time* of a task is denoted by $WCET$.
- **Deadline** D_i: is the time before a task should be completed.
- **Start time** s_i: is the time at which the task starts execution.
- **Finishing time** f_i: is the time at which the task finishes execution.
- **Response time** R_i: $R_i = f_i - r_i$ is the time, measured from the release time, at which the task is finishing.
- **Lateness** L_i: $L_i = f_i - d_i$ represents the delay of a task completion with respect to its deadline. Note, that if a task completes before its deadline, its lateness is negative. Often scheduling algorithms try to minimize the lateness of each task or the maximum lateness L_{max} of all tasks.
- **Tardiness** or **Exceeding time** E_i: $E_i = \max(0, L_i)$ is the time a task stays active after its deadline.
- **Laxity** or **Slack time** X_i: $X_i = d_i - a_i - C_i$ is the maximum time a task can be delayed on its activation to complete within its deadline.

In general, a task set can consist of tasks with different characteristics. Often, for the analysis only specific characteristic are allowed in order to simplify the implementation or the analysis itself. A task set can be classified by

Periodic or aperiodic tasks. The instances of a periodic task τ_i are regularly activated at a constant rate. The interval T_i between two consecutive activations is the *period* of the task. A periodic task set consists only of periodic tasks. An algorithm that is implemented only (!) of periodic tasks is often called to be time-driven (see Section 4.4.2.3$_{84}$). If there exists aperiodic tasks or interrupts (a special kind of a low-level aperiodic task), then the system is modelled event-driven. Periodic tasks are activated synchronously while aperiodic tasks have an asynchronous activation.

Independent tasks or tasks with resource or precedence constraints. Precedence constraints arise because of output-to-input dependencies between the tasks. That means that a task τ_i cannot run before task τ_j, because τ_j produces an output, which

is required as input for task τ_i. All dependencies of such a kind can be modelled in a directed *precedence graph*. Resource constraints occur when tasks must have exclusive access to resources, because they have to run critical sections in order to protect the integrity of the resource. The mutual exclusion let tasks block other tasks so that they cannot proceed with their calculations. If this happens, a task switch have to be done, which has direct implications onto the schedule. If there exists no resource or precedence constraints, then the tasks are said to be independent.

Preemptive or non-preemptive execution. A task τ_i can be assigned a priority p_i. A higher priority of a ready task means that the task must be executed before all other lower-priority tasks that are ready. This implies that if a higher-priority task becomes ready while a lower-priority task is actually executed, then the lower-priority task must be suspended and the execution of the higher-priority task must immediately be continued. This fact is known as *preemption*. (In non-priority systems preemption often denotes the fact that a task is suspended because its time slice of being executed is expired. Then the next task is executed by a so-called Round-Robin scheme.)

Static or dynamic priority assignments. The priorities of the tasks can be assigned statically so that they never change during the run-time of the system or they can be modified while execution and are therefore dynamic.

Soft, firm or hard real-time tasks. The *criticalness* is a parameter related to the consequences of missing a deadline (see page 62).

Deadline or period driven analysis. The deadlines define a border for the maximum execution time of a task. Thus, the deadline relative to the last task activation must obviously not be greater than the period of the task. But they can be identical. If they are equal, then the analysis identifies the periods with the deadlines. Under this assumption it is often enough to analyse the processor utilisation. Otherwise, the response time or processor demand have to be analysed (see Table 4.2$_{74}$).

For a periodic task set the *hyperperiod* H can be defined as the *least common multiple* of the periods of all tasks. This is important, because the execution order of all tasks (which is named *schedule*) will obviously be repeated after a multiple of H.

These additional assumptions are often made for periodic tasks:

- All instances of a periodic task have the same worst case execution time C_i.
- All instances of a periodic task have the same deadline d_i relative to the beginning of a period.

For a periodic task τ_i the *phase* Φ_i can be defined as the release time of the first instance of the task. The task identifier $\tau_{i,j}$, the release time $r_{i,j}$, the absolute deadline $d_{i,j} = \Phi_i + (j-1)T_i + d_i$, the start time $s_{i,j}$, and the finishing time $f_{i,j}$ are sometimes supplied with an additional index j denoting the jth instance.

Very important for real-time systems is often the so-called *absolute release jitter* $ARJ_i = \max_j(s_{i,j} - r_{i,j}) - \min_j(s_{i,j} - r_{i,j})$. It is the maximum deviation of the start time among all instances. The *absolute finishing jitter* $AFJ_i = \max_j(f_{i,j} - r_{i,j}) - \min_j(f_{i,j} - r_{i,j})$ is the

maximum deviation of the finishing time among all instances. These values must stay constant over time and should be minimal for real-time control.

In this context, a task set is said to be *feasible* or *schedulable* if all tasks finish within their deadlines:

$$\forall\, i = 1, 2, \ldots, n \;\; \forall\, j : \quad f_{i,j} \leq d_{i,j} \; .$$

Optimality criterions or cost functions can be given, which define metrics for a performance evaluation. These functions allow to compare the performance of different schedules. According to one function a schedule can be optimal or not. Common optimality criterions are (see [25, p. 41]):

- Average response time:
$$\bar{R}_i = \frac{1}{n} \sum_{i=1}^{n} (f_i - r_i)$$
- Total completion time:
$$t_c = \max_i(f_i) - \min_i(r_i)$$
- Weighted sum of completion times:
$$t_w = \sum_{i=1}^{n} w_i f_i$$
- Maximum lateness:
$$L_{max} = \max_i(f_i - d_i)$$
- Maximum number of late tasks:
$$N_{late} = \sum_{i=1}^{n} miss(f_i, d_i)$$
where
$$miss(f_i, d_i) = \begin{cases} 0 & \text{if } f_i \leq d_i \\ 1 & \text{otherwise} \end{cases}$$

4.2.2 Approaches

All of the approaches for achieving a feasible schedule can be categorised by the properties that a given task set must fulfil, for which environment they are written, and which property of the schedule they try to optimise. Graham et al. [54] proposed a notation for such a classification. They classify all algorithms by using three fields $\alpha \mid \beta \mid \gamma$ (see [25, p. 51]), that have the following meaning: The first field α describes the machine environment on which the task set has to be scheduled (uniprocessor, multiprocessor, distributed architecture, and so on). The second field β describes the task and resource characteristics (preemptive, independent, precedence constraints, synchronous activation, and so on). The third field γ indicates the optimality criterion to be followed in the schedule.

In the following paragraphs the most important scheduling algorithms for real-time systems are presented. Only the basic idea of the algorithm and a motivation for its correctness will be given. For details and formal proofs we would like to refer to the literature and especially to the book *"Hard Real-Time Computing Systems"* from Buttazzo [25].

In general, the algorithms can be classified by the prerequisites, which they demand. The algorithms can primarily be classified whether they can handle periodic or aperiodic tasks. The first section describes algorithms for aperiodic tasks, the second section algorithms for periodic tasks.

4.2.2.1 Aperiodic Task Sets

Earliest Due Date (EDD). This algorithm is also known as Jackson's Algorithm. All tasks for this algorithm have to arrive synchronously, that means at the same time, and run only once, but can have different execution times and deadlines. Neither they must have precedence constraints, nor they must use resources in exclusive mode. It is also assumed that all tasks must run on a single processor. The algorithm proposed by Jackson [68] minimizes the maximum lateness. (Graham's class: $1 \,|\, sync \,|\, L_{max}$)

By EDD the tasks have to be executed in the order of increasing deadlines. For proofing the correctness assume that exchanging the order of two consecutive tasks in any schedule will not increase the maximum lateness of the complete schedule. But the maximum lateness of these two tasks will be smaller if they are ordered appropriately.

Earliest Deadline First (EDF). This algorithm had been proposed by Horn [63] and handles tasks with arbitrary arrival times. This is especially the case when tasks arrive dynamically during execution of other tasks. It is very easy to find a feasible schedule, if *preemption* is allowed. That means that the actual task can be suspended in order to execute a more important task. A feasible schedule and minimizing the maximum lateness can be achieved if always the task with the earliest deadline is executed. Dertouzos [37] showed that EDF is optimal in the sense of feasibility. (If there exists a feasible schedule, then it will be found. Graham's class: $1 \,|\, asynch \rightarrow preem \,|\, L_{max} \rightarrow feasible$) In fact, an algorithm that minimizes the maximum lateness is also optimal in the sense of feasibility. The contrary is not true.

The acceptance test for a task must certify that for all tasks its worst case finishing time f_i is before its deadline d_i. The worst case dynamic finishing time of task τ_i is equal to the sum of the worst case dynamic finishing time of task τ_{i-1} with next lower deadline and the remaining worst case execution time $c_i(t)$ of task τ_i. (Thus, all tasks τ_i are always ordered by increasing deadline: $d_i < d_j \Leftrightarrow i < j$.) Notice, that $c_i(t)$ has an initial value equal to its computation time C_i and can be updated whenever the task τ_i is preempted.

$$\forall\, i = 1, \ldots, n: \quad f_i = f_{i-1} + c_i(t) = \sum_{k=1}^{i} c_k(t) \leq d_i$$

Tree search. If the tasks arrive asynchronously but preemption is not allowed, then EDF is no longer optimal. The problem of minimizing the maximum lateness and the problem of finding a feasible schedule become NP-hard [71, 82, 83]. For this category branch-and-bound algorithms search in all possible schedules for a feasible schedule. They perform well in the average case but degrade to exponential complexity in the worst case. The

structure of the search space is a tree, where in each level a new task is introduced into the schedule. This results to a tree depth on n for n tasks and a tree with $n!$ (n factorial) leaves. The branch-and-bound technique proposed by Bratley et al. [20] stops searching a path from the root to a leave, when adding a task to the actual schedule will lead to a missed deadline. This reduces dramatically the search space but the algorithm has in worst case still a complexity of $O(n\ n!)$. (Graham's class: $1 \,|\, no\_preem \,|\, feasible$)

Spring. This algorithm had been created for the SPRING kernel [123, 124] designed at the University of Massachusetts by Stankovic and Ramamritham. The algorithm belongs also to the tree search algorithms, but it uses a heuristic function in order to determine the task for extending the schedule. Thereby, only one path of the tree is evaluated, but for each extension of the schedule the heuristic function have to be evaluated for all tasks. Thus, the complexity reduces to $O(n^2)$, but it is not guaranteed that the algorithm will find the feasible schedule if it exists (it is not optimal in finding).

The heuristic function in *Spring* can be set to the minimal arrival time (resulting into scheduling after First Come First Served), the computation time (resulting into scheduling after Shortest Job First) or the deadline (resulting into scheduling after EDF). In order to handle additionally tasks with exclusively used resources the heuristic function can be set to the minimum *estimated starting time* $T_{est}(i)$. T_{est} is the maximum of the arrival time and all of its *earliest available resource times* (EAT) for all of its required resources. EAT for a resource k is determined upon the already defined partial schedule. Additionally, in *Spring* precedence constraints can be considered. Thus, the heuristic function only selects a task, when all of its predecessors are already scheduled and executed.

Latest Deadline First (LDF). If the given task set has also precedence constraints, i.e. tasks must be executed before other tasks, then it becomes more difficult to find a feasible schedule. In general the problem becomes also NP-hard. But if we can assume that all tasks arrive at the same time (synchronously), then Lawler [76] found the algorithm called *Latest Deadline First* also to be optimal in minimizing the maximum lateness (Graham's class: $1 \,|\, prec, sync \,|\, L_{max}$) and it can be computed in $O(n^2)$ time. LDF builds the schedule from the tail to the head. Among all tasks with no successors or whose successors have already been all selected, LDF selects the task with the latest deadline to be scheduled last. That means at run-time the task which is inserted last into the schedule will be executed first. The proof for the correctness is done similar to the one for the EDD algorithm.

EDF*. Chetto, Silly and Bouchentouf [27] found a very elegant way to schedule a task set with precedence constraints with a modified EDF algorithm. Thus, the task set must be preemptable (Graham's class: $1 \,|\, prec, preem \,|\, L_{max}$). Their idea is to transform the dependent task set into an independent task set by modifying the timing constraints. In fact, if the timing constraints are met, then it must follow that also the precedence constraints must by obeyed. In detail the minimal start time (or release time) and the deadline for the tasks have to be modified. Assume, that task τ_i must precede task τ_j, then the start time of τ_j must not be before the end time of τ_i: $r_j = \max(r_j, r_i + C_i)$. Additionally, the task τ_i must finish before task τ_j. This can be achieved by modifying the deadline of τ_i: $d_i = \min(d_i, d_j - C_j)$. These modifications can both be computed in $O(n^2)$ time. (The precedence graph must be traversed appropriately and for each task the maximum, respectively minimum must be found.)

For aperiodic task sets various algorithms had been given which are applicable under different terms of assumptions on the task set (synchronous, preemptive, precedence constraints). The algorithms also differ in their computational complexity. A summary of these algorithms can be found in Table 4.1.

	sync. activation	preemptive async. activation	non-preemptive async. activation
independent	**EDD** (Jackson '55) $O(n \log n)$ Optimal	**EDF** (Horn '74) $O(n^2)$ Optimal	**Tree search** (Bratley '71) $O(n\, n!)$ Optimal
precedence constraints	**LDF** (Lawler '73) $O(n^2)$ Optimal	**EDF*** (Chetto et al. '90) $O(n^2)$ Optimal	**Spring** (Stankovic & Ramamritham '87) $O(n^2)$ Heuristic

Tab. 4.1: Scheduling algorithms for aperiodic tasks. (see [25, p. 75])

4.2.2.2 Periodic Task Sets

The basis for the schedulability test in the periodic case is the *processor utilisation factor* U. Since C_i/T_i is the fraction of the processor time spent in executing task τ_i, the utilisation factor for n tasks as originally given by Liu and Layland in 1973 is defined by the equation

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \quad \text{[85].}$$

Obviously, any task set that has an utilisation factor greater than one is infeasible, because the utilisation factor is a measure of the computational load of the processor and that cannot exceed 100%. On the other hand for each task set Γ and for each scheduling algorithm A exists another upper bound U_{ub} ($0 < U_{ub} \leq 1$) for which the task set is schedulable and any increase in the utilisation of any task will make the schedule infeasible. With such an $U_{ub}(\Gamma, A)$ the set Γ is said to *fully utilise* the processor. For a given scheduling algorithm A, the *least upper bound* $U_{lub}(A)$ of the processor utilisation factor is defined as the minimum of all $U_{ub}(\Gamma, A)$ for all task sets Γ, which fully utilise the processor.

That means that a task set Γ is certainly feasibly schedulable if its processor utilisation factor is below the least upper bound [$U(\Gamma, A) \leq U_{lub}(A)$], but it need not to be schedulable if it is greater as the least upper bound [$U_{lub}(A) < U(\Gamma, A) \leq 1$], and it is absolutely not schedulable if it is greater than one [$U(\Gamma, A) > 1$].

Various algorithms had been defined for different assumptions made for the task set. Each algorithm A defines its own least upper bound $U_{lub}(A)$. In the following the most

important and widely used algorithms for aperiodic task scheduling will be shortly presented.

Rate Monotonic (RM). This algorithm assigns priorities to the tasks. Thereby, tasks with shorter periods will be given higher priorities. Since periods are constant, RM assigns the priorities statically before run-time and the priorities will never change. RM is intrinsically preemptive. Thus, an actually running task is preempted by a newly arriving task with shorter period. In 1973, Liu and Layland [85] showed that RM is optimal among all fixed-priority algorithms. That means that no other fixed-priority algorithm can feasibly schedule a task set, which cannot be scheduled by RM. They also derived the least upper bound for the processor utilisation factor. A generic task set of n tasks has

$$U_{lub}(\Gamma_n, RM) \;=\; n(2^{1/n} - 1).$$

For any task set with an unlimited amount of tasks this is often also given as

$$U_{lub}(\Gamma, RM) \;=\; \lim_{n\to\infty} n(2^{1/n} - 1) \;=\; \ln 2.$$

A detailed derivation of this can be –of course– found in [85] and in [25, p. 82]. It should clearly be mentioned here, that this schedulability condition is sufficient to guarantee the feasibility of a task set, but it is not necessary. That is, if the utilisation factor is between U_{lub} and 1, then nothing can be said about the feasibility of the set. However, since RM is optimal an improvement can only be achieved by using dynamic-priority assignment algorithms.

Earliest Deadline First (EDF). The EDF algorithm –as already presented in Section 4.2.2.1$_{69}$– is also applicable for periodic tasks. EDF selects tasks according to their absolute deadline, which means that the task with an earlier deadline will be executed first. Thus, the priority will be assigned dynamically during execution to the tasks. Thereby, EDF is intrinsically preemptive. The optimality criterion also holds for periodic tasks. Moreover, EDF can be used to schedule aperiodic and periodic tasks at the same time. Based on the results achieved by Liu and Layland [85], Spuri, Buttazzo, and Sensini showed in [119] and [25, p. 93] that the least upper bound for the processor utilisation factor under EDF is one:

$$U_{lub}(\Gamma_n, EDF) \;=\; 1.$$

Deadline Monotonic (DM). The Deadline Monotonic priority assignment algorithm weakens the "period equals deadline" constraint. Each task τ_i of the task set may have an arbitrary deadline $D_i \leq T_i$. But D_i stays constant for all instances of τ_i. According to the DM algorithm, each task is assigned a priority inversely proportional to its deadline. Since deadlines are constant, DM is also a fixed-priority assignment algorithm. In 1982, Leung and Whitehead [84] proposed this algorithm as an extension for the RM. They showed in a similar way compared to RM, that DM is also optimal for fixed-priorities and deadlines lower than their periods. The sufficient schedulability test for RM can be applied for DM, if the periods are replaced by their deadlines: $\sum_{i=1}^{n}(C_i/D_i) \;\leq\; n(2^{1/n} - 1)$. However, such a test is not optimal as the workload on the processor is overestimated. A less pessimistic and also sufficient schedulability test calculates the maximum interference I_i of higher priority tasks on the lower priority task τ_i. Assuming that the tasks are ordered by increasing relative deadlines ($i < j \Leftrightarrow D_i < D_j$), then this interference

is $I_i = \sum_{j=1}^{i-1} \lceil D_i/T_j \rceil \; C_j$. The schedulability test has to assure for all tasks that the finishing time is before the deadline: $\forall i : 1 \leq i \leq n \quad R_i = C_i + I_i \;\; \leq \;\; D_i$. This test assures that a higher prioritised task τ_j interferes task τ_i in every of its periods T_j. But this might not be true, if the execution time of task τ_i is so small, that its finishing time or response time $R_i < (\lceil D_i/T_j \rceil - 1)T_j$. That means, if at least in the last period of τ_j, which covers the deadline of τ_i, task τ_j does not interfere (preempt) τ_i, then the interference is overestimated. That means that also this test is only sufficient but not necessary.

In 1992/1993, Audsley et al. [5, 4] proposed a sufficient and necessary schedulability test for DM. They propose to use the response time R_i of task τ_i instead its deadline D_i for the calculation of the number of interferences. Thus the equation, which is to be checked for all tasks is

$$\forall i : 1 \leq i \leq n \quad R_i = C_i + I_i \;=\; C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil \; C_j \;\leq\; D_i.$$

The problem in calculating the response time is that R_i appears on both sides of the equation. Fortunately, R_i can be computed stepwise from the first to the last interference [25, p. 101]:

$$schedulability\ test(DM) := \begin{cases} R_i^0 = C_i,\ I_i^{-1} = 0 \\ \text{do} \\ \quad R_i^{k+1} = I_i^k + C_i \text{ with } I_i^k = \sum_{j=1}^{i-1} \left\lceil \frac{R_i^k}{T_j} \right\rceil \; C_j \\ \text{until } I_i^{k+1} + C_i \geq R_i^{k+1} \\ R_i^{k+1} \leq D_i \Leftrightarrow \text{ feasibility} \end{cases}$$

Moreover, in the calculation of this recurrent formula the task set is *not* feasible, if $\exists\, k$: $R_i^k > D_i$.

EDF*. Also EDF with its dynamic-priority assignment approach is again applicable to periodic task scheduling when deadlines are less to periods. Only the schedulability test has to be adapted. Baruah, Rosier, and Howell [8] proposed in 1990 to use the *processor demand criterion*. The processor demand of any task set during the interval $[0, L]$ is defined as the cumulative processing time $C_P(0, L)$ that is required by tasks with deadlines less or equal to L: $C_P(0, L) = \sum_{i=1}^{n} \lfloor L/T_i \rfloor C_i$. In 1993, Jeffay and Stone [69] stated that a set of periodic tasks with deadlines equal to periods is schedulable under EDF, if and only if

$$\forall L \geq 0 : L \geq \sum_{i=1}^{n} \left\lfloor \frac{L}{T_i} \right\rfloor \; C_i.$$

It should be noticed that it suffices to test the equation only for values of L equal to release times less than the hyperperiod H, because $\forall L \in [r_k, r_{k+1}), \lfloor L/T_i \rfloor = \lfloor r_k/T_i \rfloor$. Moreover, it is enough to test the equation for values of L for release times less or equal to the *busy period* $B_P = \min\{L \,|\, W(L) = L\}$ with $W(L) = \sum_{i=1}^{n} \lceil L/T_i \rceil C_i$. During the busy period the processor is fully utilised. It ends either with an idle time, or with the start of a release of a new periodic instance.

Baruah, Rosier, and Howell [8] showed that for periodic tasks with deadlines less than periods the processor demand can still be used, if considered that the last interval covering L must be taken into account, if the deadline is before L ($D_i \leq L$, see Figure 4.1).

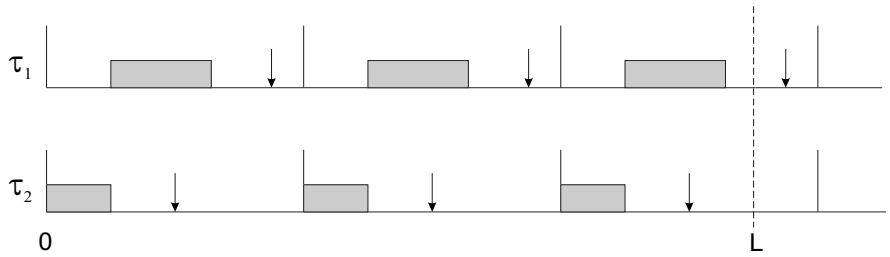

Fig. 4.1: Processor demand when deadlines are less than periods (from [25, p. 106]).

Thus, if $D = \{d_{i,k} | \, d_{i,k} = kT_i + D_i, \; d_{i,k} \leq \min(B_P, H), \; 1 \leq i \leq n, \; k \geq 1\}$, then a set of periodic tasks with deadlines less than periods is schedulable by EDF if and only if

$$\forall L \in D \quad L \geq \sum_{i=1}^{n} \left(\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) C_i.$$

A summary of scheduling algorithms for periodic tasks can be found in Table 4.2. Notice, that the algorithms for deadlines equal to periods can be solved by the processor utilisation approach and therefore, they can be computed in linear time ($O(n)$), whereas the processor demand approach for task sets with deadlines less than periods requires pseudo-polynomial time.

	$D_i = T_i$	$D_i \leq T_i$
Static priority	**RM** Processor utilisation approach $U \leq n\,(2^{1/n} - 1)$	**DM** Response time approach $\forall\, i : R_i = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \leq D_i$
Dynamic priority	**EDF** Processor utilisation approach $U \leq 1$	**EDF*** Processor demand approach $\forall\, L \geq 0 \;\; L \geq \sum_{i=1}^{n} \left(\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) C_i$

Tab. 4.2: Summary of guarantee tests for periodic tasks. (see [25, p. 107])

4.3 Servers for Aperiodic Tasks

In most real-time applications periodic and aperiodic tasks have to be executed in parallel (or quasi-parallel). Typically, the periodic tasks are time-driven and underlie hard real-time constraints, whereas the aperiodic tasks are event-driven and have soft or firm and rarely hard real-time constraints. The main objective of the real-time operating system is

to guarantee the hard real-time tasks and to provide a good average response time and minimal deadline misses for soft and firm real-time tasks.

Often, aperiodic tasks are characterised by a minimum inter-arrival time, which defines a kind of maximum probable period. For this reason they are also called *sporadic tasks*. In the scheduling theory aperiodic tasks are handled in another way than periodic tasks. Whereas in the previous sections some basic techniques for hard real-time scheduling had been discussed, in this section some principles for real-time scheduling with combined periodic and aperiodic task sets will be presented.

The so-called *server technique* is an appropriate approach to handle aperiodic real-time tasks in the case when a periodic scenario is assumed. A specific periodic task is responsible to service all pending aperiodic requests. This special task is called *server*. The server approach tries to optimise the processor utilisation bound and tries to minimize the response time for aperiodic requests in contrast to non-server techniques.

Background Scheduling. In *Background Scheduling* soft aperiodic tasks are handled in the background behind periodic tasks. They will be executed after all hard real-time tasks had run and still some processor time is left. Therefore, aperiodic real-time tasks will get assigned a priority just lower than the priority of any periodic real-time task.

Fixed Priority Servers. A representative for a *Fixed Priority Server* is the so-called *Polling Server* (PS). The server task has (as any periodic task) a period T_s and a maximum computation time C_s. The computation time is called *capacity* of the server. The server is scheduled like any other periodic real-time task, not necessarily at lowest priority. The schedulability test for a combination of PS under RM can be guaranteed if

$$U_p + U_s = \sum_{i=1}^{n} \frac{C_i}{T_i} + \frac{C_s}{T_s} \leq U_{lub}(n+1) = (n+1)(2^{1/(n+1)} - 1) \; .$$

The finishing time f_a of an aperiodic request $\tau_a = (r_a, C_a, D_a)$ can be computed as

$$f_a = \left\lceil \frac{r_a}{T_s} \right\rceil T_s + \left\lfloor \frac{C_a}{C_s} \right\rfloor T_s + (C_a - \left\lfloor \frac{C_a}{C_s} \right\rfloor C_s) \; .$$

This results into a schedulability test for aperiodic tasks: $f_a \leq r_a + D_a$ [25, p. 111 ff.].

Another *Fixed Priority Server* is the *Deferrable Server* (DS) proposed by Lehoczky, Sha, and Strosnider [79, 136]. Unlike the Polling Server, DS preserves its capacity until the end of its period. Only, when aperiodic tasks are served, the capacity is decreased accordingly. With the beginning of each server's period, the capacity is replenished at its full value. Thus, future aperiodic requests can be served until the capacity is exhausted during the whole period of the server. The schedulability test for a combination of DS under RM can be guaranteed if

$$U_p \leq \ln\left(\frac{U_s + 2}{2U_s + 1}\right) \; .$$

Note, that if $U_s < 0.4\ U_p$, then the RM bound ($\ln 2$) gets worser, but for $U_s \geq 0.4$ the presence of DS improves the $U_{lub}(DS) = U_s + \ln(\frac{U_s+2}{2U_s+1})$ (see [25, p. 119 ff.]). The finishing time of an aperiodic request under DS can be computed to

$$f_a = \begin{cases} r_a + C_a & \text{if } C_a \leq c_s(t) \\ \left\lceil \frac{r_a}{T_s} \right\rceil T_s + \left\lfloor \frac{C_a - \Delta_a}{C_s} \right\rfloor T_s + (C_a - \Delta_a - \left\lfloor \frac{C_a - \Delta_a}{C_s} \right\rfloor C_s) & \text{otherwise,} \end{cases}$$

where $\Delta_a = \min[c_s(t),\ (\left\lceil \frac{r_a}{T_s} \right\rceil T_s - r_a)]$ [25, p. 116 ff.].

Priority Exchange Server. The *Priority Exchange Server* (PE) had been introduced in 1987, by Lehoczky, Sha, and Strosnider [79]. The PE algorithm also uses a periodic server task (usually at a high priority) for servicing aperiodic requests. The capacity of the server is not preserved at the server's priority while no aperiodic request are pending. However, the capacity is divided into capacities on all priority levels. At the beginning of the server's period its capacity is replenished at its full value. If no aperiodic requests are pending and a periodic task have to be executed, then the capacity is borrowed or transfered to the capacity of the periodic task's level. Otherwise, (that means an aperiodic request can be executed, due to having capacity on any level, or the processor is idle) the capacity degrades linear from level to level. By this, the server's priority is virtually exchanged with a scheduled periodic task. By this rule, the capacity is preserved at the periodic task's priority level and can be consumed later if required. The capacity is only consumed by idle times or serving aperiodic requests. The least upper bound for the processor utilisation factor, when PE is combined with RM for the periodic tasks, is

$$U_p \leq \ln\left(\frac{2}{U_s + 1}\right) \quad \text{[25, p. 125 ff.].}$$

Other examples for servers are the *Sporadic Server* (SS) [116] and the *Slack Stealing Method* [81]. Appropriate adaptations of all these Fixed Priority Servers to the dynamic case, where the periodic tasks are scheduled using EDF, are made by Buttazzo et al. [117, 118]. The Dynamic Priority Servers are the *Dynamic Priority Exchange Server* (DPE), the *Dynamic Sporadic Server* (DSS), the *Total Bandwidth Server* (TBS) and the *Earliest Deadline Late Server* (EDL).

4.4 Real-Time Analysis for Communication

When we have a distributed system, tasks run on different processors. They often communicate via communications links. Tasks will send and receive data from other tasks. This defines in/out dependencies between the tasks. These in/out dependencies can be modelled by precedence constraints. But, in fact, a receiving task can execute until it requires input from another task. If this input is not present, the task will be blocked. This is the normal case. It is also possible that the receiving task continues execution, although input data are not present. This is called *asynchronous receive*, whereas the normal one is called *synchronous receive*. Also the sending of data can be *asynchronous* or *synchronous*. Here, the normal case is the asynchronous case, where the sender task continues with its execution directly after the data are sent. In the synchronous case the sender task will be blocked until the receiver task has received the data. Therefore, an acknowledgement has to be sent back. The synchronous send and the synchronous receive define a so-called *rendezvous*.

Because the synchronous send and the asynchronous receive play a minor role for communications, often only the synchronous receive is investigated for real-time analysis. The early receive, which means that the data are not present, causes delays for the receiver task due to the blocking. It is the task of the real-time analysis for communication to determine the maximum blocking delay. This maximum blocking delay has then to be

incorporated into the schedulability analysis for the task set. The maximum blocking delay contains the message transfer time. For this reason, the message transfer times must be calculated. Additionally, the message transfer exclusively occupies the communication link during its transfer. Thus, the scheduling of the message transfers onto the links has to be analysed.

In order to define a model for the real-time analysis for communications a lot of concepts for the task scheduling are re-used. The main idea is to replace the task characteristics by communication characteristics of the message transfer.

4.4.1 Definitions

Before we present some analysis concepts for the real-time communication, a few definitions have to be given in order to clarify their meaning.

Communication. Communication means the exchange of information. In the case of *process communication* data will be sent from one sender task to one receiver task. This will be an uni-directional communication. If the sender and receiver have to setup a connection between each other before they communicate, then they establish a so-called *channel*. Communication via channels is indispensable for real-time communication, because resources have to be reserved and the schedulability have to be verified. In contrast to *channel-oriented communication*, in *packet-oriented communication* a so-called *datagram* is sent from the sender to the receiver. The *datagrams* are often delivered only by a best-effort strategy. If the sender sends the data to multiple receivers, we speak from *multi-cast communication*. If the data is sent to all other possible receivers, then this is a *broadcast*. It is often assumed that the task set is periodic and, therefore, also the messages will periodically be sent.

Message. The *message* is the portion of information, which have to be sent. For real-time communication it is essential that a message has a limited size, as tasks have a worst-case execution time.

Packet. The messages have to be enriched by *protocol information* (so-called *headers* and *footers*). A message plus this protocol information can result into one or more packets. If the physical size of the *packet*, which can be delivered at once over a physical connection, is limited, then a message must be split into several packets. This is known as *fragmentation*. The protocol is responsible to fragment the message at the sender's site and to de-fragment the packets accordingly at the receiver's site.

Link. Link refers to the physical connection between to nodes of the communication network. This can be a *point-to-point connection*, like a serial RS-232 connection, a *bus* like an Ethernet line or a backplane, or it can be a *ring*, like a Token Ring as it used in the FDDI. When a packet has to cross several links until it reaches its receiver (or its destination), we speak from a *multi-hop connection*.

Path. The sequence of links that a packet has to take for a (multi-hop) connection is named the *path* or *route* of the packet.

Network. The processor nodes together with their physical communication links form the *communication network*. It can be described by a graph. Here, the nodes represent

the processors and the links. An edge of the graph represents a physical connection of a processor to a link. Therefore, the graph is bi-partide. This graph then defines the *topology* of the network.

Routing. The path that a packet takes is determined by the *routing*. A routing algorithm calculates this path. The *routing table* on each processor node contains information for the routing algorithm in order to determine the route. The routing algorithm often tries to calculate an optimal route according to, for example, the shortest-path, the fastest connection, or an optimal load distribution for every link, etc.

The following properties of an uni-directional communication V_k are of interest for the real-time analysis:

- **Sending task** τ_{sk}: is the task which sends the message
- **Receiving task** τ_{ek}: is the task which receives the message
- **Sending node** S_{sk}: is the node from which the message is sent
- **Receiving node** S_{ek}: is the node on which the message is received
- **Size** M_k: is the maximum size of the message
- **Period** P_k: is the interval between two consecutive message transfers and is also called *minimal message interarrival time*
- **Deadline** D_k: defines the maximum delay of the message, that is allowed
- **Worst-case response time** R_k or **End-to-end delay** E_k: is the maximum time between the send event and when the receiver is unblocked
- **Jitter** J_k: is the maximum difference between any two response times

4.4.2 Approaches

In the following sections two basic and outstanding approaches for applying real-time analysis to communication systems are presented. The first approach uses the utilisation factor, whereas the second approach analyses the response times. As a third approach the Time-triggered Protocol with its reservation mechanism will briefly be described.

4.4.2.1 Utilisation Approach

In 1994, Sathaye and Strosnider [112] proposed a framework about reasoning of the timing correctness of messages in a network. Therefore, they defined a *communication request* V_k as a tuple:

$$V_k = (S_{sk}, S_{ek}, M_k, P_k, E_k, J_k)$$

They applied the generic schedulability test approach, which checks that the utilisation is not exceeding 100%, to a communication network. They directly identified the communication requests to be the tasks. The computation times has been replaced by the message

size. With this assumptions they defined, that a set of network requests is transmission-schedulable (t-schedulability), if and only if the maximum saturation of the network S_{max} is not greater than one:

$$S_{max} \leq 1 \;\Rightarrow\; \text{t-schedulability}.$$

The maximum saturation is defined as the maximum over all saturations S_i, which each communication request produces:

$$S_{max} = \max\{S_i \mid 1 \leq i \leq n\}.$$

The saturation of a communication request is thereby defined as the minimum of all cumulative workloads $W_i(t)$ during the interval $(0, D_i]$ normalised by time:

$$S_i = \min_{0<t\leq D_i} \frac{W_i(t)}{t}$$

The workload is defined as

$$W_i(t) = \sum_{j=1}^{i} (C_j + Ovh_j) \left\lceil \frac{t}{T_j} \right\rceil + Ovh_{sys_i}(t) + B_i \text{ , where}$$

C_j is the time for which the network is occupied due to the message size.

Ovh_j is the time for which the network is occupied due to the transmission of extra headers, trailers, or acknowledgements. Assuming that messages have to be split into several packets and an extra acknowledgement message in one packet has to be transmitted before a new message can be sent, the packet overhead Ovh_j is defined as follows:

$$Ovh_j = (C_h + C_t + C_{ack}) \left\lceil \frac{C_j}{P_{max} - (C_h + C_t)} \right\rceil \text{ where}$$

C_x is the time spent for transmitting the message, header, trailer or acknowledgement, respectively and

P_{max} is the maximum size for a packet on the link.

$Ovh_{sys_i}(t)$ captures communication request independent system overhead, like media access delays and overhead because of unsynchronised clocks (If the destination station's clock is ahead, then the message must be delivered with a shortened deadline):

$$Ovh_{sys_i} = O_{MAC} + O_{clock}$$

B_i is the time the communication request V_i is delayed, because of the actual transmission of a (possibly) lower priority packet due to non-preemption of packets on the network (priority inversion problem).

T_j is the period of consecutive message transmissions.

Sathaye and Strosnider define S_{max} as the "degree of schedulable saturation" metric. It can be observed that S_{max} is monotonically non-decreasing, if either the number of connections increase or each C_i or Ovh_{sys_i} or B_i increases. For this reason $1 - S_{max}$ can be considered to be the amount of high-priority work that can be added to the system per unit time without missing a deadline.

For a network consisting only of a single link, the above presented "schedulable saturation" metric can directly be applied. But additionally, Sathaye and Strosnider applied this uniform schedulability framework to three different network types by modifying the above described parameters. These networks are a dual link network, a token ring network and a multi hop network.

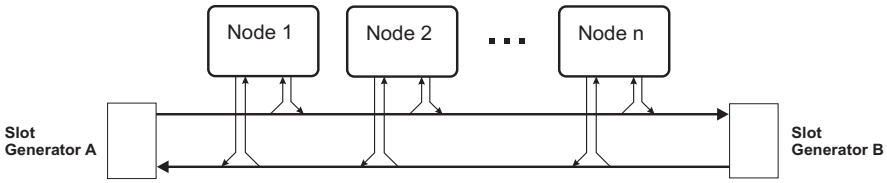

Fig. 4.2: The DQDB network topology.

4.4.2.1.1 IEEE 802.6 DQDB The IEEE 802.6 DQDB is a dual link network, where all stations are arranged in a line with two slot generators at the ends and the links are working in opposite directions (see Figure 4.2). No trailers or acknowledgments are required. But for transmitting a message a slot has to be reserved by requiring a free slot from the appropriate slot generator. A request has to be sent to the slot generator and the slot has to arrive at the station before the message can be sent. This results into a delay of $2d_i$, if d_i is the time for transmitting a slot between station i and the appropriate slot generator. Preemption is originally not defined in the standard, but can be achieved, because each station reads from the incoming link and sends the data forward on the outgoing link. Thereby, the station is able to replace the incoming packet by a new one. Thus, Ovh_j, Ovh_{sys_i} and B_i can be defined as follows:

$$\begin{aligned} Ovh_j &= C_h \left\lceil \frac{C_j}{P_{max} - C_h} \right\rceil \\ Ovh_{sys_i} &= 2d_i \\ B_i &= 0 \end{aligned}$$

4.4.2.1.2 IEEE 802.5 FDDI The IEEE 802.5 FDDI is a Token Ring (see Figure 4.3) where for each station k a certain bandwidth H_k is allocated. The bandwidth is normally calculated as

$$H_k = \frac{U_k}{U_{net}}(TTRT - W_T),$$

where U_k is the network utilisation of station k and $U_{net} = \sum_{i=1}^{n} U_i$ is the network utilisation. $TTRT$ is the target token rotation time and W_T is the walk time (the token rotation time when the network is idle). A trailer and acknowledgment before sending a new message is not required. Assuming that a message cannot be sent while not having the token and while a packet is actually be transmitted (no preemption), the parameters have to be defined as follows:

$$\begin{aligned} Ovh_j^k &= C_h^k \left\lceil \frac{C_j^k}{P_{max}^k - C_h^k} \right\rceil \\ Ovh_{sys_i}^k &= (TTRT - H_k) \left\lceil \frac{t}{TTRT} \right\rceil \end{aligned}$$

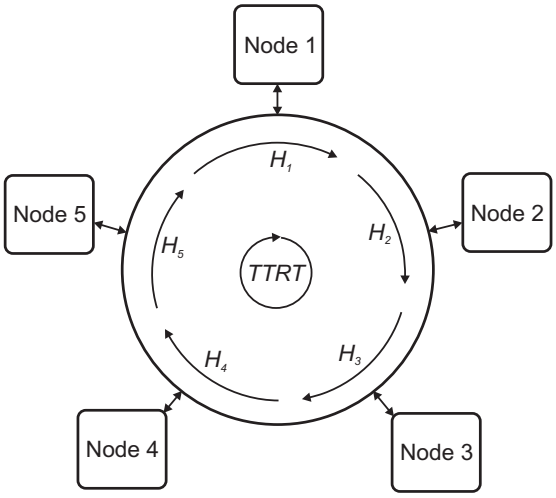

Fig. 4.3: The FDDI token ring topology.

$$B_i^k = P_{max}^k$$

Due to the fact, that each station can have a different H_k a new index k for the station is introduced. Therefore, the maximum saturation S_{max} must be calculated over the maximum saturations S_{max}^k of all stations, which are calculated over the saturations S_i^k of each message transmitted from this station.

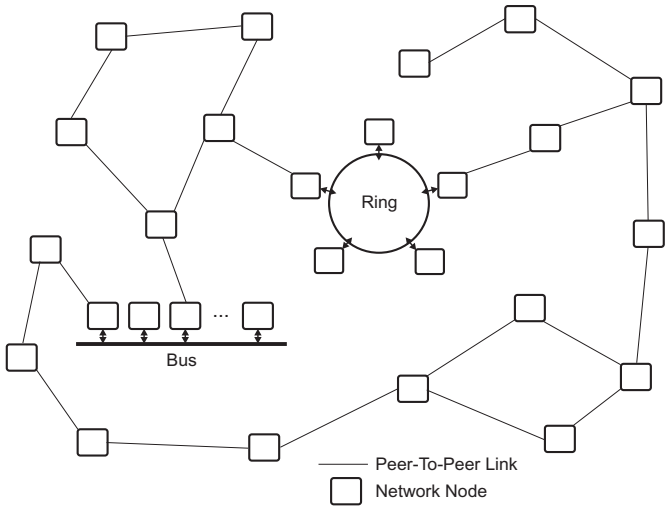

Fig. 4.4: An example for a multi-hop network.

4.4.2.1.3 Multi Hop Network In a multi hop network messages have to be routed through several stations from the source to the destination (see Figure 4.4). Each link that the message has to follow is named hop. The sequence of links (or stations) from the source to the destination is named path. The transmission schedulability approach must be adapted by assuring that for the paths of all messages the saturation of each hop of

the path P is not grater than one:

$$\forall j \in P \quad \forall i = 1, 2, \ldots, n_j \quad \min_{0<t\leq D_{i,j}} \frac{W_{i,j}(t)}{t} \leq 1,$$

where j refers to a link on the path and n_j is the number of connections on that link. There exists only a small problem: When messages are scheduled on multiple links in series, they may arrive at the next link before their deadline on the current link. If the messages are scheduled upon their arrival, they lose their periodic characteristics. This undesirable effect can be prevented, if messages arriving in a given period become eligible for transmission only at the beginning of the next period. This is known as *double buffering* or *stop-and-go queuing* [53]. This is often implemented by having an in-queue and an out-queue. Messages arriving at a station are inserted into the in-queue. Messages that have to be transmitted are taken from the out-queue. Every time a period is over, the appropriate messages in the in-queue are transfered into the out-queue. This procedure is known as *promotion*. The time between two promotions of message i on link j of the path at station k is $E_{i,j} + T_{i,j}\lceil \phi_{prop_j}/T^k_{i,j} \rceil$. The promotion at each station on the path increases the end-to-end delay for the message. The end-to-end delay E_i for the complete message transfer on the path is not only the sum of all promotion delays on each station, but must be calculated as:

$$E_i = h_k E^k_{i,j} + \sum_{j=1}^{h_k} T^k_{i,j} \left\lceil \frac{\phi_{prop_j}}{T^k_{i,j}} \right\rceil,$$

where h_k is the number of hops on the path and ϕ_{prop_j} is the propagation delay on link j. Thus, it should be additionally checked, whether

$$E_i \leq D_i.$$

4.4.2.2 Response Time Approach

In 1995, Tindell, Burns, and Wellings [140] considered m messages, which have to be separated into C_m packets of constant size in each period. They tried to bound the time between the arrival of the sender task and the time at which the destination task is unblocked. This time is said to be the *worst-case response time* R_m. They separated this time into different timing intervals:

$$R_m = J_m + Q_m + \rho_{trans} + \phi_{prop} + \tau_{notif},$$

where

J_m	is the generation delay, which is equal to the WCET of the sending task.
Q_m	is the queuing delay, which is caused by previous transmission of former packets.
ρ_{trans}	is the transmission delay for one packet.
ϕ_{prop}	is the electrical propagation delay on the physical line[1].
τ_{notif}	is the notification delay until the receiving task really is consuming the message.

[1] The electrical propagation delay may be small for a local area network, or may be very large if, say, a satellite link is part of the network.

The transmission, propagation and notification delay are assumed to be constant for each message and are worst-case estimations depending on the operating system and the communication link and protocol.

Because the queuing delay can change in every period, Tindel, Burns, and Wellings defined the queuing delay $Q_{m,q,k}$ and therefore also the worst-case response time $R_{m,k}$ of the k-th packet of message m as the maximum over all q-th multiples of the period T_m:

$$R_{m,k} = \max_{q \in \mathbb{N}_0} (J_m + w_{m,q,k} - qT_m + \rho_{trans} + \phi_{prop} + \tau_{notif})$$

Hereby, $w_{m,q,k}$ is the width of the *level i busy period* [80]. The busy period is defined as

$$w_{m,q,k} = qC_m\rho + (k-1)\rho + B_m + \left(\sum_{\forall h \in hp(m) \cap out(p)} I_h \right) \rho + \left\lceil \frac{w_{m,q,k}}{T_{token}} \right\rceil (T_{token} - H_p)$$

In this equation ρ is the worst-case transmission time for one packet and B_m is the delay caused by priority inversion[2] and I_h is the interference (i.e. total preemption) by messages with higher priority than m ($hp(m)$) and sent from the same processor p ($out(p)$). The interference is thereby defined as

$$I_h = \left\lceil \frac{w_{m,q} + J_h}{T_h} \right\rceil C_h \text{ with } w_{m,q} = w_{m,q,C_m}$$

Obviously, the worst-case response time of the message is then equal to the worst-case response time of the last packet of the message: $R_m = R_{m,C_m}$.

Tindel, Burns, and Wellings state that the calculation of the busy period $w_{m,q,k}$ can be done by the following and simple recurrent formula:

$$\begin{aligned} w^0_{m,q,k} &= 0 \\ w^{n+1}_{m,q,k} &= w_{m,q,k}(w^n_{m,q,k}) \end{aligned}$$

This formula should converge, if the utilisation is not greater than 100%:

$$w_{m,q,k} = \lim_{n \to \infty} w^n_{m,q,k}$$

The above given formulas can be applied not only to token protocols, but also to priority busses (see Figure 4.5), like for example CAN[3]. For priority busses set $T_{token} = H_p$, so that there is no delay caused by not having a token.

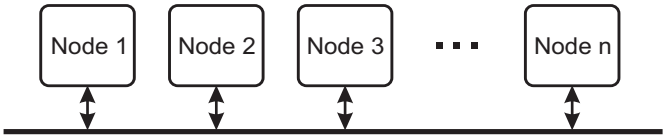

Fig. 4.5: The topology of a communication bus, like CAN. A message that is sent by one station can be received simultaneously by all stations.

For the schedulability test it must be checked, if for all messages the worst-case response time is lower or equal than the maximum end-to-end delay:

$$\forall i = 1, 2, \ldots, n : \; R_i \leq D_i$$

[2] The formula can be simplified if assumed that $B_m = \rho$.

[3] Controller Area Network, see [67]

4.4.2.3 Time-Triggered Approach

One of the main problems that causes delays for the communication is the *media access*. When a packet is already transfered via the link, another packet cannot be transmitted simultaneously. However, if this is still done, the transmission of both packets will be corrupted. For this reason, a variety of media access protocols has been developed. A possible categorisation can be the one that is presented in Table 4.3.

<table>
<tr><td colspan="8">Time-multiplexed</td><td rowspan="4">Frequency-
multiplexed</td></tr>
<tr><td rowspan="3">Isochronous</td><td colspan="7">Asynchronous</td></tr>
<tr><td colspan="4">Controlled</td><td colspan="3">Accidental</td></tr>
<tr><td colspan="2">Centralised/Master</td><td colspan="2">De-centralised/Ring</td><td colspan="2">CSMA CD</td><td>Repetition</td></tr>
<tr><td>Single</td><td>Multiple</td><td>Logical</td><td>Physical</td><td>Priority</td><td></td><td></td></tr>
<tr><td>ATM
TTP</td><td>USB
Bitbus
DQDB</td><td>VME</td><td>Token bus
Profibus</td><td>FDDI
Interbus</td><td>CAN
VAN
J1850</td><td>Ethernet</td><td>ALOA
Wireless</td><td>Wireless</td></tr>
</table>

Tab. 4.3: Possible categorisation of media access protocols and examples for their implementation.

For the real-time communication mainly three media access protocols are used: isochronous *time-multiplexed* protocols, *token-based* protocols, and *priority-based* protocols. The last two ones have been briefly described in the previous sections. The time-multiplexed TDMA (*Time Devision Multiple Access*) protocols grant each communication request or station one or more *time slots*, when they are allowed to send messages. For this reason, the complete available time is divided into intervals (*time slots*). Often they are of equal size. The assignment of time slots to messages will be done during a reservation phase.

Often the slot assignment is done statically before the run-time of the system. Thus, the times when communication can occur is pre-defined. This can be extended to the complete execution model: Also the processes are scheduled by such a fixed assignment. The schedule is fixed and pre-defined. The execution order and the activation times for each task must be calculated in advance and will not change during execution. Thus, the system behaves totally deterministic and always in the same way. A famous representative for this technique is the *Time-triggered Protocol* (TTP), which had been investigated in detail and implemented at the university of Vienna by Prof. Kopetz et al. [72]. Because every action of the system is pre-defined and happens during a certain time interval, these systems are sayed to be *time-driven systems*. On the contrary, systems that re-act on events are known as *event-driven systems*.

MARS The MARS[4] operating system [36, 73, 74] implements the TTP approach. The communication as well as the process scheduling are time-triggered. Thus, priorities are not required. The complete time slot assignment is calculated offline during the design-phase of the system. For the communication is assumed that all stations are attached to one bus (Ethernet or CAN). The communication in MARS is based on state messages. State messages represent a kind of shared and distributed variable. Its value can be overwritten

[4] MAintainable Real-time System

from a remote site during its pre-defined time slot. But only the actual state, which represents the last write, can be read. The update of the states happens periodically. MARS does not offer synchronisation primitives like semaphores or a rendesvouz principle. In MARS synchronisation as well as precedence constraints must be considered during the time slot assignment.

One of the biggest drawbacks of this approach is the request for a global clock synchronisation: Each station must identify the same time slot at the same time. Additionally, those systems are failure-prone during overload situations. Slightly shifts or moves, when specific event handlings are required, are not possible. In this sense token-based systems are more robust.

Often the periodical and static appearance of message transfers is not found in the reality. Here, often only a certain bandwidth is required, which must be mapped onto the TDMA scheme. Cruz and Turner developed a so-called *leaky bucket* model [77], which is used, for instance, in the ATM (*Asynchronous Transfer Mode*) protocol [111] in order to weaken the synchronous TDMA scheme to an asynchronous bandwidth allocation scheme.

4.5 Resource Constraints

All the previously described scheduling algorithms assume that the tasks are independent from each other. But in most cases this is wrong. Often, tasks claim for exclusive access to resources. Semaphores [21, 38, 98] or monitors [137] are used to protect such a critical section of the tasks, when it accesses the shared resource in exclusive mode. For this reason a task must be blocked, if the required resource is already used by another task. But this leads to delays of the task, which extend the finishing time. If it is known in advance, which task claims for which resource, then the blocking times B_i for each task can be computed. But there exists another problem that is more severe:

The Priority Inversion Phenomenon: *Assume that task τ_1 has higher priority than task τ_2. Despite this fact τ_1 may be blocked by task τ_2. If there exists a task τ_3 with intermediate priority, then τ_3 preempts τ_2 while holding the resource, that is also claimed by τ_1. The result is that task τ_3 runs before τ_1! Moreover, the blocking time of τ_1 is unbounded if more intermediate tasks arrive.*

For RM or EDF and related algorithms this means that the execution order of the tasks can be completely corrupted. The fact that a lower priority task blocks a higher priority task, because of requiring the same resource, is the semantic of mutual exclusion. But the problem of the priority inversion and the unbounded delay can be solved.

Priority Inheritance Protocol. In 1990, Sha, Rajkumar, and Lehoczky [115] proposed the *Priority Inheritance Protocol* (PIP). The basic idea of this protocol is, when a task τ_i blocks one or more higher priority tasks, it temporarily inherits the highest priority of the blocked tasks. This prevents medium priority tasks from preempting τ_i and, therefore, they will not prolong the blocking period.

Theorem 4.1 (Sha, Rajkumar, and Lehoczky): Under the Priority Inheritance Protocol, a task τ_i can be blocked for at most the duration of $\min(n, m)$ critical sections, where n is the number of lower-priority tasks that could block τ_i and m is the number of distinct semaphores that can be used to block τ_i.

Out of this theorem the schedulability test for rate monotonic scheduling [remember $\sum_{i=1}^{n}(C_i/T_i) \leq n(2^{1/n}-1)$] can be extended to

Theorem 4.2: A set of n periodic tasks using the Priority Inheritance Protocol can be scheduled by the Rate-Monotonic algorithm if

$$\forall j,\ \ 1 \leq j \leq n,\ \ \sum_{k=1}^{j} \frac{C_k}{T_k} + \frac{B_i}{T_i}\ \leq\ i(2^{1/i}-1).$$

Here, the tasks are ordered by their priority: $j > i \Leftrightarrow P_j < P_i$. Observe, that the equation now must be solved for all tasks. The equation can be interpreted as follows. In order to guarantee a task τ_i, we have to consider the effect of preemptions from all higher-priority tasks ($\sum_{k=1}^{i-1} C_k/T_k$), the execution of the task itself (C_i/T_i), and the effect of blocking due to all lower-priority tasks (B_i/T_i).

What still is to be done, is the computation of the blocking time B_i for each task τ_i. We can define the *ceiling* $C(S_k)$ to be the priority of the highest-priority task that may use the semaphore S_k. With this and the definition of the Priority Inheritance Protocol we can follow:

Lemma 4.3: In the absence of nested critical sections, a critical section of τ_j guarded by S_k can block τ_i only if $P_j < P_i \leq C(S_k)$.

The idea for the proof is the following: If $P_i \leq P_j \Rightarrow \tau_i$ cannot preempt $\tau_j \Rightarrow \tau_i$ cannot be blocked by τ_j directly.

The blocking time B_i of a task τ_i can then be defined as the minimum of two different sums of durations of longest critical sections:

$$B_i = \min(B_i^l, B_i^s),$$

where

$$B_i^l = \sum_{j=i+1}^{n} \max_k [D_{j,k} :\ C(S_k) \geq P_i]$$

is the sum of durations $D_{j,k}$ of the longest critical sections for any task τ_j with priority lower than P_i that can block τ_i, and

$$B_i^s = \sum_{k=1}^{m} \max_{j>i} [D_{j,k} :\ C(S_k) \geq P_i]$$

is the sum of durations $D_{j,k}$ of the longest critical sections for any semaphore S_k that can block τ_i.

An algorithm for the computation of this blocking times is presented in [25, p. 197] and has a complexity of $O(mn^2)$. The upper bounds for the B_i^l and B_i^s are not very tight, because two or more critical sections guarded by the same semaphore may be summed. But, if two critical sections of different tasks are guarded by the same semaphore, then they cannot be both blocking simultaneously. An algorithm based on exhaustive search, where all possible combinations of blocking critical sections are considered, was presented by Rajkumar [103], but it has an exponential complexity.

The Priority Inheritance Protocol is a good solution for the problem of unbounded delays due to exclusive resource access. But it has two drawbacks: It does not prevent the formation of chained blocking and –as a reason of this– it does not prevent the formation of a deadlock.

Priority Ceiling Protocol. The *Priority Ceiling Protocol* (PCP) was also proposed by Sha, Rajkumar, and Lehoczky [115] in 1990 as an improvement of the PIP. It solves additionally the drawback of chained blocking and prevents deadlocks. The idea of the PCP is that it extends PIP by a special granting rule for locking a free semaphore. The rule says: A task is not allowed to enter a critical section, if there exists other locked semaphores which can block the task.

This is achieved by assigning to each semaphore S_k a *priority ceiling* $C(S_k)$ equal to the priority of the highest-priority task that can lock the semaphore. A task τ_i is allowed to enter a critical section only if its priority is greater than the ceiling of all semaphores currently locked by other tasks than τ_i. If the entering of the critical section is denied, then the task τ_i is said to be blocked on semaphore S^* by the task τ_j that holds that semaphore S^*. (Note: S^* is the semaphore with the highest ceiling of all currently locked semaphores by tasks other than τ_i.) When the task τ_i is blocked by task τ_j, task τ_j inherits the priority of task τ_i. In general, a task inherits the highest priority of all tasks it blocks. When a task exits a critical section, then its priority is updated as follows: If no other tasks are blocked by the task τ_j, it is set to its nominal priority P_j; otherwise, it is set to the highest priority of the tasks blocked by τ_j. The priority inheritance is transitive; that is, if a task τ_3 blocks a task τ_2, and τ_2 blocks task τ_1, then τ_3 inherits its priority from τ_1 via τ_2.

A theorem, which was given by Sha, Rajkumar, and Lehoczky, says that under the Priority Ceiling Protocol, a task τ_i can be blocked for at most the duration of one critical section. Additionally, it follows that a critical section $Z_{j,k}$ (belonging to task τ_j and guarded by semaphore S_k) can block a task τ_i only if $P_j < P_i$ and $C(S_k) \geq P_i$. From this observation we can follow, that the maximum blocking time of a task can be computed as follows:

$$B_i = \max_{j,k}\{D_{i,j} \mid P_j < P_i,\ C(S_k) \geq P_i\}.$$

Remark, that the calculation of the blocking times under PCP is much easier than under the PIP. But in order to implement the PCP and to calculate the *ceilings* of the semaphores, all critical sections, which can be entered by any task, must be known in advance. Because this is impossible for a general implementation of an operating system, nearly all operating systems do not implement PCP.

But in TEReCS the requirement specification contains also the information about critical sections, because the system calls on the semaphores have to be modelled in order to configure DREAMS correctly and to insure a more detailed timing analysis. Thus, the critical sections are known in advance and the PCP can be implemented in DREAMS and configured by TEReCS.

Another quite interesting resource management protocol is the *Stack Resource Policy* (SRP), which will not be described here. The interested reader may be referred to the literature (for example [25, p. 208]).

Implementation Considerations

All of these resource management protocols are implemented by different `signal()` and `wait()` methods. Also, the PCP does not require a waiting queue per semaphore as it would be required in the normal case. The reason is that the task can be kept in the ready queue since the other tasks will have higher priority. But the semaphore requires an extra field for storing the ceiling and the *process control block* (PCB) requires an extra field for storing the task's active priority. It would be also convenient to have a field in the PCB for the semaphore on which the task is blocked. Generally, the operating system should maintain for the PCP a list of all currently locked semaphores ordered by their ceilings.

You see that the different protocols require different implementations of the operating system internal data structures and functions, which are not directly related to the synchronisation primitives `signal()` and `wait()`. Thus, it is highly recommended to have a configuration management that takes care for a valid and working implementation of the operating system.

4.6 Contribution of the Chapter

This chapter gave a brief survey about the real-time problems and their solutions in the case of scheduling, communication, and resource access.

All the different implementations of scheduling algorithms, communication protocols, resource management protocols, and their schedulability tests allow a brought variety of combinations and selections. It should be the task of a configuration management to select the appropriate methods and data structures in order to maintain the complexity of their combinations. All the different requirements of the applications and all the possible implementations inside the operating and communication system should be described within a concept that allows to derive a description for the configurator's domain knowledge bases. This task is the motivation for the next chapter.

CHAPTER 5

From Taxonomy Towards Configuration Space

In this chapter the attempt is made to identify characteristics of real-time systems by which they can be categorised. These characteristics are ordered within a taxonomy. This taxonomy is important for the completeness of this thesis. It can give hints, where configuration of a run-time platform for a real-time system makes sense, i.e. where configuration should be done in order to improve the system's efficiency. The dimensions or domains that span the taxonomy can define the items of a run-time platform, which are configurable. It should be clearly mentioned that the taxonomy, which is developed here, is especially designed for this purpose. That means that there may exist other taxonomies, which are designed for other reasons.

The Table 5.1 on the following page is an attempt to identify properties by which a distributed real-time system can be classified. The different properties describe different operating system implementations, whereas for one class of properties all operating system implementations are equal. Formally speaking, the taxonomy defines equivalent classes of real-time systems, where for each class the operating system is fixed and does not change.

Obviously, the taxonomy can describe systems that are impossible to implement and, therefore, an operating system that ensures a temporal execution does not exist. For example, the precedence constraints of the tasks define a deadlock or the cumulative workload of a processor is exceeded. Thus, a system including the final operating system should be analysed for its correct execution. This analysis phase should be done after the configuration phase, because configuration decisions have impact on the execution, i.e. the operating system overhead can differ.

In the next few sections the impact of the taxonomy items on the operating system configuration will be described in more detail.

5.1 Taxonomy for Real-Time Systems

In order to identify configuration items for an operating and communication system the properties and characteristics and even implementation possibilites of distributed real-time applications should be investigated in general. In the following sections some di-

<table>
<tr><th colspan="2">Global System Model and System Behaviour</th></tr>
<tr><td colspan="2">• time driven vs. event driven (polling vs. interrupts)
• mono-programming vs. multi-programming
• static vs. dynamic (creation of new tasks)
• deterministic vs. non-deterministic
(worst-case execution time vs. average execution time,
connection- vs. packet-oriented communication)
• distributed vs. local vs. centralised</td></tr>
<tr><th>Task Characteristics</th><th>Communication Characteristics</th></tr>
<tr><td>• periodic vs. aperiodic vs. sporadic
• preemption vs. no-preemption
• constraint types:
– resource constraints
(critical sections)
– precedence constraints
(communication, priority)
– timing constraints (deadlines)
– criticalness:
⋆ hard, firm, soft</td><td>• periodic vs. aperiodic vs. sporadic
• channel- vs. packet-oriented
• preemption vs. no-preemption
• constraint types:
– resource constraints (bandwidth,
time slots)
– precedence constraints (priority)
– timing constraints (deadlines
end-to-end delays)
– criticalness:
⋆ hard, firm, soft
• peer-to-peer vs. broadcast vs. multi-cast ($1:1$, $1:m$, $n:1$, $n:m$)
• buffered vs. unbuffered (size)
• maximum message size
• no loss of message order
• guaranteed transmission (acknowledgements)
• error detection
• error correction</td></tr>
<tr><th colspan="2">Machine and Hardware Properties</th></tr>
<tr><td colspan="2">• single-processor vs. multi-processor
• homogeneous vs. heterogeneous processors (endianess, speed, memory size)
• shared vs. distributed memory
• interconnection network:
– topology (peer-to-peer, bus, ring, etc.)
– unidirectional vs. bidirectional links
– neighboured communication vs. multi-hops
– homogeneous vs. heterogeneous links
– media access protocol per link
– maximum packet size per link
– delay of packet per link
– transfer time of packet per link</td></tr>
</table>

Tab. 5.1: Dimensions of a taxonomy for real-time systems.

mensions will be presented, for which the implementation differs in order to meet special properties of the application. These application properties can form the dimensions of a taxonomy, by which the implementations can be classified. That means, if the value of a taxonomy's dimension differs for different applications, then the implementations of the operating and communication system will be different, too. Therefore, on the one hand the implementations will belong to different classes of applications. On the other hand, if the values of all dimensions are the same for different applications, then the implementation of the operating and communication system will also be the same. Thus, the applications belong to the same taxonomy class.

The dimensions of the taxonomy can be categorised into four different major domains. These domains group dimensions concerning properties of the system model and the system's behaviour, task characteristics, communication characteristics, and properties of the distributed machine (hardware) on which the system should run. These domains had been identified, because they tackle different aspects of the application's implementation. Moreover, we will see later that these taxonomy domains will result into different parts of a requirements specification.

5.1.1 System Model and System Behaviour

Time driven or event driven: Time driven means that inputs are regarded at fixed absolute times (i.e. the system is polling). Event driven considers events to be taken at their arrival. Thus, the system has to handle interrupts. To have interrupts or not makes a big difference for the operating system, because interrupts require a special treatment like saving the context and restoring it. In contrast, time driven systems require a global clock in order to synchronise.

Mono-programming or multi-programming: This property defines whether the programmer specifies a single task or has multiple tasks running simultaneously. If this property is regarded per node (processor) of the system, then obviously, the question for requiring a task switching mechanism (scheduler) per node can be answered.

If the system behaviour can be regarded to be static (the task set and its properties do not change during execution), then the scheduling algorithm can be simpler and the task set's feasibility test can be done offline. For example, the simplest scheduling scheme applied to a static task set is a fixed activation order. On the other hand, if tasks can arrive dynamically, then the operating system must integrate an acceptance test and possibly, the scheduling algorithm requires preemption, which requires again saving the context and its restoration.

If the system is deterministic, then it is said to be predictable. In this case the conventional management strategies (like RM, DM, LDF, EDD, EDF, etc.) for the task set can be applied. But, if some of the system's tasks are not deterministic, then assumptions or probabilities about their properties are made. In this case the so-called *server algorithms* for aperiodic task scheduling should be considered.

A centralised distributed system requires more powerful hardware for the server and, therefore, other system services than a homogeneously distributed system. A client/server implementation has other requirements than distributed agents. Especially, the bottleneck of a centralised server with applied high performance services or the load balancing

service for a highly dynamic system should be considered.

5.1.2 Task Characteristics

A task set can be regarded periodic, aperiodic, or additionally contains sporadic tasks. This has direct impact on the schedules for a timeliness execution. Obviously, for this reason the scheduling algorithms, which had been introduced in Chapter 4[61], had been developed.

Additionally, all the task properties themselves allow or forbid a specific scheduling algorithm. These are especially properties classifying the algorithms into categories (see Table 4.1[71] and Table 4.2[74]). If a task allows preemption, has precedence, resource, or timing constraints, then these properties demand for a special treatment by the scheduler. Moreover, synchronisation by `signal()` and `wait()` or semaphores has to be considered. Thus, PIP, PCP, or SRP come into the picture.

When tasks run quasi-parallel on a processor, because they can be preempted, then mutual access to shared resources is requested. The internal state of a resource must not be changed by more than one task simultaneously. Otherwise, data integrity or correct behaviour is not guaranteed. Thus, tasks use operating system primitives for synchronisation and protection, like semaphores, monitors, or simple `signal()` and `wait()` primitives. The usage of these primitives requests for a special functionality of the operating system. Additionally, these synchronisation primitives have great impact on the scheduling, because tasks might be delayed (blocked). In order to assure a special behaviour the synchronisation protocols PIP, PCP or SRP had been designed.

Many scheduling algorithms and their analysis assume that the given tasks are independent from each other. But when a single application is split into several tasks in order to exploit inherently possible parallelism, then these tasks work normally on data, which is forwarded and manipulated from task to task in a pipelined manner. Examples may be streaming applications, where filter, conversion, and Fourier transformation algorithms are sequentially ordered. Other examples may be control algorithms, where subsequent controllers read input values that had been produced by other controllers. Controllers can also be arranged hierarchically, where top-level controllers (e.g. the body control of a car) have influence on low-level controllers (e.g. active suspension) by generating appropriate control values. These in/out dependencies between tasks request for communication. Thus, communication demands should very carefully be investigated and can have great impact on the running task set.

If these tasks are local to a processor and, for example, use shared memory communication, then the communication problem can be transformed into a resource and synchronisation problem. But when the task set is distributed over several processor nodes, which are interconnected by communication links, then the calculation of the delays become more complex. This is because of mutual influences of tasks that are not directly communicating but acquiring shared resource access of the links. The situation becomes more severe, when for the message transfer several links have to be traversed, because there is no direct connection between the sending and receiving processor. For this reason, the communication in a distributed real-time system should be investigated in detail.

5.1.3 Communication Characteristics

As seen from the analysis of communication systems the properties of a task set can be transferred to a set of messages. Messages can be periodic, aperiodic, or sporadic. This mainly has impact on the reservation scheme for the bandwidth (which is equal to the processor allocation). Periodic messages often have to be established by channels. Thereby, bandwidth or time slots of the links are reserved. This allocation scheme is named *channel-oriented communication*. This method is also recommended, if the maximum delay of a message transfer should be predictable. So, real-time messages should be handled always channel-oriented. On the other hand, *packet-oriented communication* exists. This method is to be preferred, if no reservation has to be done or the message transfer time cannot have unlimited delays. Unbounded delays occur, when the message traffic in the network is not predictable. Often, packet-oriented communication works only with a best-effort strategy for delivering messages.

Another very interesting property of a message transfer is, that a message can be preempted during the transfer. Here, preemption is relatively more severe, because the message again has to be completely transferred after a preemption occurred. Thus, preemption destroys a message transfer. For this reason, protocols often do not allow preemption.

Precedence constraints also exists for messages. Often for communication, the sending and receiving of messages have to preserve the message order. This means, if a message a was sent before a message b, then the message a should arrive before message b. This can either be assured by the rule, that messages cannot overtake other messages, or the receiver protocol stack delivers unordered received messages only ordered to the application, by considering message numbers and buffering of orderless messages.

Resource constraints of messages have to be considered implicitly, if the channel-oriented communication is used. Here, the route and, therefore, the links that the message has to take for each hop are fixed. Routing has a major impact on the resource constraints of a message. The routing is steered by the origin and destination of the message. If the sender is fixed, we distinguish peer-to-peer $(1:1)$, multi-cast $(1:m, m<n)$, or broadcast $(1:n)$ messages (where n is the total number of all destinations). If we do not have the classical peer-to-peer (P2P) case, then it depends on the topology, if messages have to be duplicated and have to be sent simultaneously on multiple links. Often ignored is the case, when several sources send messages to a single destination $(m:1)$, like clients send messages to a server. The receiver must have the ability to await messages from different sources.

If messages have to take several links in order to reach their destination, then the messages sometimes have to be stored into buffers on intermediate nodes, before they are forwarded (see “stop-and-go queueing” on page 82). For embedded systems with hard resource constraints, the maximum required buffer size must be known in order to save memory and to guarantee no message loss.

Another problem arises and has to be investigated, when the maximum packet size on a link is smaller than the message size. Then, a message has to be split into several packets on the sender side and the packets have to be assembled to the original message on the receiver side. Here, problems like ordering, buffer size, and preemption have again to be considered. Even, if a packet on a link cannot be preempted, the complete message can

be delayed, because the sending of a packet can be delayed by packets of other messages due to media access problems. These media access problems occur, because during the required setup-delay for a new packet, another node can acquire the media. If packets have priorities, then this problem can be much more worse and can result into priority inversion.

Sometimes the transmission of a message cannot be guaranteed by the physical architecture of the link. A message can be completely lost or is partly corrupted during the transfer. But the application demands for correct transmissions and no message loss. Then, the communication protocol has to assure error detection and correction. This sometimes requires the re-transmission of a previously corrupted or lost packet. Thus, positive or negative acknowledgements are sent in the opposite direction than the communication itself. For this reason backwards channels have to be established. These channels acquire additionally bandwidth and have to be handled not only by the protocol, but also by the analysis.

5.1.4 Machine and Hardware Properties

Characteristics of the machine and hardware have also impact on the configuration. The processor architecture itself selects or forbids services, because the services are using a concrete assembler dialect or other hardware specific features. Moreover, processor architectures might use different orderings for storing their high and low bytes of words (big or little endian storage). If packets are transmitted between tasks running on such different processors, the byte ordering of the packet for each number compound, which is greater than a byte, must be changed at the sender's or receiver's side. Thus, one protocol stack has to deal with this conversion and has to be enriched by an appropriate functionality.

The processor architecture is also of great interest for the analysis . The same service can have different worst case execution times due to different processor speeds, caching, or pipelining, etc.

Not only the processor architecture, but also the concrete devices, which have to be managed by the operating system, require specialised device drivers. This problem is already well known for general purpose operating systems like Windows™ or Linux. But it becomes more difficult for embedded and real-time operating systems. Even for the same device the driver implementation should differ due to different use cases. The communication devices and protocol stacks are considered especially in this thesis, because of the distributed aspect of the targeted application scenarios. It is the primary goal of the configuration aspect to support the application with the optimally adapted services with the best efficiency and to remove unnecessary features from the operating and communication system. Additionally, it should be tolerable to specify a lot of services, devices, and/or connected links per processor. The final application requirements should determine, which services, devices, and links are really used and which are not. This implicitly determines whether a driver is required or is not required.

A very important property of the hardware, which has impact on the link usage, is the topology of the interconnected processors and links (media). Required communication is routed through the network by determining a path that the message has to use. Such a path describes the resources, which has to handle the message. These are devices for

the sending and receiving, as well as media for the transport. Additionally, processors are specified for the sender and receiver and for intermediate nodes (hops), at which the message has to be forwarded. The topology greatly determines whether forwarding is required or not. Forwarding of messages often requires dynamic routing and buffering of the messages. If forwarding can be omitted, then the system overhead due to communication can dramatically be reduced.

If forwarding is required, then another problem arises. Each communication media defines its own protocol stack, which is used for the sender and receiver side. If on the path different in- and out-links are used, then on such intermediate nodes two different protocol stacks have to be bridged. Bridging means protocol transformation, which additionally leads to system overhead and have to be coped by the involved protocol stacks.

Each type of hardware link and protocol stack require special communication drivers. Thus, the system overhead is different. Moreover, each link architecture and its media access protocol cause different transmission times and delays for the packets. This has to be considered by the analysis.

5.2 Configuration Items

Until now the different requirements that an application may seek for have been identified. These requests form the dimensions of the previously presented taxonomy. The primary goal is to identify items or entities within an operating and communication system that have other implementations in different taxonomy classes. These changes from one taxonomy class to another and the reason for these changes have to be formally described. This formal description spans the configuration space for the operating and communication system.

But, how do we identify different implementations for other taxonomy classes? Fortunately, this is easy. Researchers have already identified properties of different applications, which in fact define the taxonomy, and, therefore, they "invented" different implementations leading to better results for the concrete case (taxonomy class). For example, there exist different implementations for scheduling algorithms, because special task set properties require special treatment in order to achieve the best performance. There we find the relationship between properties or requirements of the application and exchangeable items inside the operating system.

Implicitly, these relations have been mentioned while the different properties, characteristics, and requirements of the taxonomy were presented. These items have been gathered in Tab. 5.2 on the following page. On a very high level some areas of an operating system can be identified by grouping exchangeable items together, which are influenced by the same characteristics of the taxonomy. For instance, all the scheduling algorithms are grouped into the group of *scheduler services*, because they depend on the properties of the given task set.

Yet another example would be the *protocol services* that depend on the properties, which must be assured during the message transfer. In fact, the protocol does not only depend on the required quality of service (QoS) for the message transfer, but it also depends on the hardware capabilities of the communication link. This is an example where more

than one taxonomy dimension (QoS of message transfer and hardware properties) have impact on the same item (protocol). Two dimensions have to be matched by the configuration, that means by the final implementation. The solution employs protocol stacks, where on each level the protocol can be exchanged in order to achieve the desired QoS. Moreover, the application can access the protocol stack at different levels or the hardware requests for a special low level protocol. Additionally, the protocols rely hierarchically on each other. Each demand for a specific quality for the communication service requests for a specific protocol. Each protocol itself requests for an appropriate lower level protocol, until a protocol is reached, which handles the communication link. This is the lowest level of a protocol stack, which is often named *link layer* (e.g. see OSI model [66]). This is a

Global Services	
• interrupt management (context saving) • synchronisation (*signal and wait*) • scheduling • memory management • communication • repositories/databases/lists/arrays • timer management (task suspension by *timer sleeps*)	
Task Scheduling (*per processor*)	**Communication Scheduling** (*per link*)
• off-line schedulability vs. on-line acceptance test • global policy (*deadline, slack, remaining, execution time, dynamic, server, etc.*) • multi-level hierarchy (*per level*): – algorithm (*EDD, LDF, EDF, DM, RM, PS, DPE, etc.*) – ordering (*deadline, priority, remaining execution time, slack time, FIFO, criticalness, etc.*) • dispatching scheme – context switch time (> 0 ?) • real-time support yes/no • critical section yes/no – algorithm (*semaphore, mutex, monitor, etc.*)	• off-line vs. on-line – bandwidth/time slot reservation – channel establishment • multi-level hierarchy (*per level of protocol stack*): – max. packet size – protocol: ⋆ preserving message order ⋆ error detection/acknowledging ⋆ error correction ⋆ buffering ⋆ forwarding (device usage) ⋆ reception (device usage) ⋆ splitting ⋆ duplicating ⋆ delivering ⋆ routing ⋆ conversion (bridging) – protocol overhead – maximum delay
Hardware Interface	
• resource usage – device interrupts – exclusive device access – device parameters (*address, baud rate, memory usage, etc.*)	

Tab. 5.2: Configuration items.

step-wise refinement of the requests, where the requests that are leading to sub-requests of the sub-layer are transitive.

At a very high level the services of an operating and communication system that are influenced by a certain domain inside the taxonomy can be grouped into:

- interrupt management services
- scheduling services
- synchronisation services
- memory management services
- communication services
- repository services
- time management services

Each of these groups can be divided into sub-groups, which depend on a different domain of the taxonomy's dimension. A domain of the taxonomy is defined as a specific set of properties, characteristics, or requests. The different values or instances of this domain directly lead to different implementations of a specific operating system function. For instance, the domain to implement the system time driven or event driven, requests for timers or interrupt functions (or both, if time driven and event driven is mixed). The domain of tasks that have critical sections requests for synchronisation mechanisms like they are supported by semaphores, mutexes, monitors, etc.

In Table 5.2 on the preceding page, besides the high level service groups, the sub-groups, for example, for task scheduling and for the communication services, have been sketched. At the bottom of all configurable (selectable) items we find the hardware. Here, the hardware resources are enumerated, which can alternatively be used. The demands of the taxonomy for special hardware domains requires specific resources to be present or not.

The resources are not only defined by the required hardware dimensions, but also by the application's dimensions (demands). The application's demand for a broadcast prefers a communication bus in contrast to a peer-to-peer network. Nevertheless, the application demands and the hardware properties define a trade-off, which has to be solved by the configuration process. Besides the resource selection, parameters for the resources must be determined. The resource itself as well as its parameters can be identified as configuration items that have to be assigned to concrete values.

5.3 Creating the Configuration Space

Previously, we have identified a taxonomy, which led directly to the configuration items in an operating and communication system. Now, we need to develop a formal representation for the configuration items and their dependencies. This description should span the design space of all operating system classes that are formed by the taxonomy.

The alternatives of a dimension in the taxonomy identify exchangeable items in the implementation. Such alternatives naturally define an OR-group. For example, the dimension of scheduling services contains all alternative scheduling algorithms. But only one algorithm should be implemented. Another example is the critical section domain, which consists of alternative synchronisation algorithms. On the one hand, either one item of a

set of alternatives is required or none of the items are availed. Thus, the OR-group should allow no selection. On the other hand, sometimes more than one alternative should be selectable. An example can again be the critical section domain. Because tasks differ in their implementation model, the tasks can request different synchronisation algorithms. This implies, that for instance, the operating system has to support semaphores and monitors at the same time. Thus, side constraints can have impact on the domain's value selection.

If a value in a domain is another domain, then this is a transitive request. In the previous section we said this to be a refinement of a request. For example, a scheduler can be a non-real-time scheduler or a real-time scheduler. Both requests or items can be refined as shown in Table 5.3_{103} and form a domain by themselves. This directly defines a transitive dependency between two domains (OR-groups).

Often, the dependencies between values of the taxonomy domains and other domains are manifold. For instance, the Round-Robin scheduler is a value of the scheduler domain and, therefore, it is part of an OR-group of all scheduler services. If the preemptive Round-Robin scheduler will be selected, then it requires a periodic timer interrupt and a queue in order to manage the FIFO list of all the ready tasks. This is a classical AND-group of simultaneously required sub-services. Here, these sub-services are items of other domains. In this case these domains are *time management*, *interrupt management* and *repository*. For this reason, if all AND/OR-dependencies have been found, then not a tree will be the result, but an AND/OR-DAG (*Directed Acyclic Graph*) will describe the design space of all configurable items.

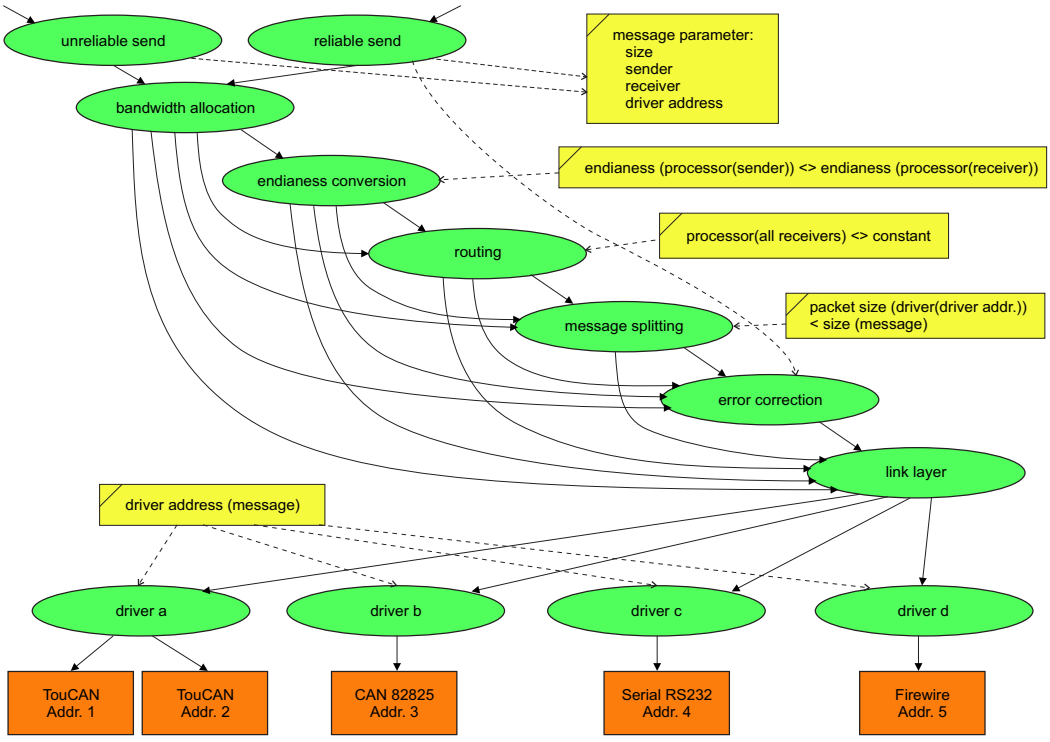

Fig. 5.1: OR-DAG that is representing one possible real-time communication stack.

In Figure 5.1 an example for a customisable real-time communication stack is sketched. Only the sender side of the protocol stack is displayed. Each oval node of the graph

represents a protocol layer, respectively a software service of the operating system. The rectangles at the bottom of the graph represent hardware devices, via which the messages can physically be sent. At the top of the graph the two nodes *unreliable send* and *reliable send* are the interface functions of the API (Application Programming Interface). These functions can be called from the application in order to send a message. Because they serve as root nodes of the graph, they are called *primitives* in this thesis.

The directed edges between the nodes describe the calling dependencies within the protocol hierarchy. That means, if there exists an edge from a protocol A to a protocol B (or to a device b), then protocol A requests for protocol B (or the device b) in order to achieve its function. When more edges are leaving a node, then all sons form an OR-group. All nodes of an OR-group can alternatively be selected in order to serve for the function of the father-node. Therefore, the graph in Figure 5.1 on the facing page is only an OR-DAG. There does not exist any AND-relation.

The OR-group can be an exclusive OR, which means only one successor must be selected, or a normal OR, which means that at least one successor must be selected. All ORs that are depicted in Figure 5.1 on the preceding page are exclusive ORs, except the ORs of the *link layer* and its *drivers*.

It can be observed that the protocol services *endianess conversion*, *routing*, *message splitting* and *error correction* can be omitted from the protocol stack. In Figure 5.2 a simple example for a service that can possibly be omitted is depicted. Assume that service A is always required. The service B can be ommited, because for service A the services B or C can be alternatively be selected. The service C is always requested, either directly by A or indirectly by B. Thus, in a final configuration either all three services must be present or only A and C. If each service produces costs, then selecting only A and C is cheaper. But side constraints can additionally request for service B. For that case, B cannot be omitted.

The protocol services in Figure 5.1 that can be omitted are only requested, if special constraints of the messages, which are all sent via the two sending primitives, hold. The constraints are shown by the rectangles with one marked corner. The text inside the rectangle describes briefly and informally the constraint. Therefore, we assume that messages have certain parameters, like the sender and receiver node identification, and the size of the message. The driver address, which is to be used in order to send the message physically, is determined by the routing protocol.

The configuration problem is to select only the minimal set of required services and devices of the OR-DAG in Figure 5.1. This is to be done in a way, that all messages, which will be sent via the two root primitives, can be delivered to their receivers. For exam-

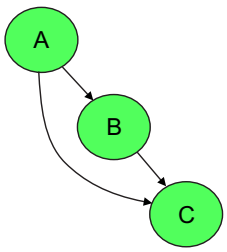

Fig. 5.2: Technique in an OR-DAG for eventually omitting a service B.

ple, the protocol service *error correction* is not required, if no message will be sent via the primitive *reliable send*. Or, the protocol *routing* is not required, if all messages that will be sent, must be sent via the same device. Thus, during configuration for each message a path from the used sending primitive down to the appropriate device have to be found. Additionally, all constraints must be considered. The services and devices of all paths must be selected for the final configuration.

The description of the OR-DAG in Figure 5.1$_{98}$ is done on a very high level of abstraction. For a real usable configuration space description it has to be refined and must be described more formally. One possible way for such a description will be developed in Section 6.3$_{108}$. Additionally, for real message transfers the receiver's protocol stack must be considered similarily. Moreover, there will exist constraints between the services of both protocol stacks, respectively OR-DAGs. For instance, if the protocol service *message splitting* on the sender's side is selected, then on the receiver's side the opposite protocol *combining packets* must be selected. Or none of these protocols at all must be selected on both sides.

5.4 "Puppet Configuration"

As we have seen in the sections before, a taxonomy for distributed real-time systems can define the requirements, properties, and characteristics, which split the implementations of an operating and communication system into different classes. These requirements, properties, and characteristics can be interpreted as the demands of the application of a real-time system. For this reason, these items define a requirements catalogue. This catalogue is the basis for a requirements specification.

We have also seen that the domains of this taxonomy (requirements specification) naturally define an AND/OR-DAG by refinement. The refinement of domains leads to the identification of configuration items (alternative services) inside the operating and communication system. Sometimes, they classify the operating and communication system's implementations into instances, where a special service must be present, or is not required. Often, they define alternatives. Either the alternatives of an implementation are already known (because of research, e.g. in the area of real-time analysis) or they can easily be identified. By definition a property of the taxonomy splits the implementations into (at least) two different instances. Thus, this entails that implementations handle this special aspect of the taxonomy in appropriate ways.

During this refinement process some sub-domains have to be created, which result into additional requirements. They have only to be evaluated, if the top-level-domain is also required. Hence, these sub-domains have to be evaluated during the refinement of the configuration. While questions are answered, additional questions arise, which also have to be answered. This is a sort of depth-search in the AND/OR-DAG until we have reached domains with no sub-domains. In this sense, configuration means to find appropriate paths from each top-level domain to sub-domains, which are not again refined. Thereby, the selection of one or a set of values for a domain leads to a specific sub-domain. Only sub-domains of selected branches will be taken into consideration.

The AND/OR-DAG, which is the result of the refinement process for the demanded properties of all applications, spans the complete design space of the operating and com-

munication system. This design space describes by its alternatives all the different implementation classes. In fact, there can be a lot of classes.

The top-level domains of the taxonomy (*primitive* usages defined in the requirements specification) serve as starting points for the search in the AND/OR-DAG of one operating and communication system's design space (see Section 6.4.3$_{115}$ and Figure 6.5$_{116}$). The fixing of a domain's property to a concrete value is possible by answering the appropriate question of the requirements specification. Thus, by giving answers to those questions, the alternatives inside the AND/OR-DAG can be selected. This is similar to playing a puppet (see Figure 5.3).

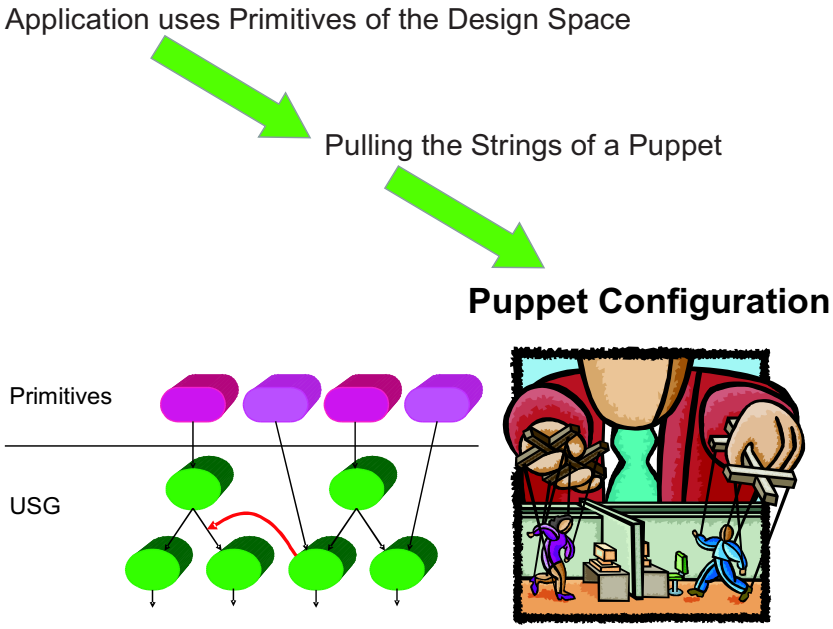

Fig. 5.3: The basic idea of *"Puppet Configuration"*.

The puppet player pulls the strings of the puppet in order to change the spatial arrangement of the puppet's skeleton. If we would take a picture of a puppet, then this picture would show *one* specific configuration of all possible arrangements of the puppet's pieces. We can transfer this example to the taxonomy's AND/OR-DAG. While a special property of the taxonomy is selected and instantiated with a concrete value, we directly select one alternative of an OR-group. This way, we have captured a special aspect, which, for example, means the move of an arm of the puppet. But in order to touch an item with the puppet's hand, we have also to move the hand. This is similar to the request for a sub-domain of the taxonomy to be answered. Moreover, by pulling one string of a puppet, more than one "bone" of the puppet will be moved. This is also true for the AND/OR-DAG and the taxonomy. Inside the DAG hierarchy the selection of one special property to a concrete value can have impact on more than one domain, because the domains are members of a transitive relation. The joints in-between the bones of the puppet are representing this transitive relationships and some side-constraints inside the AND/OR-DAG.

The AND/OR-relationships are not sufficient to describe the design space of an operating and communication system. This we have seen in the example in Figure 5.1$_{98}$.

Additionally, constraints have been defined. The AND- and OR-relations describe only the alternatives of a configuration. The constraints describe, which alternatives have to be taken upon some parameters of primitive usages. Although, the selection of the primitives determines which sub-graphs have to be considered, the constraints determine the selection of the correct branches in this sub-graphs.

The constraints can be categorised into different types. The first type that we have observed in the protocol hierarchy has been the costs per selected service. The overall costs for a complete path down from the primitive to the devices have to be minimised. Other constraints had been defined for parameters of the primitive usages. Thus, certain conditions defined upon properties of the messages that are sent via that primitives force the selction of specific alternatives. Besides forcing a service also the deny of the selection of a service in an OR-group is possible. Additional types of constraints might be useful. They will be described in Section 6.3.4$_{110}$.

5.5 Conclusion

The taxonomy of Table 5.1$_{90}$ is neither complete nor the only possible solution. It had been organised to have a first starting point for the design of a configuration description of a customisable operating and communication system. The domains of the taxonomy precisely describe the questions of the requirements specification that must have impact on configuration decisions. This led to the idea of *"Puppet Configuration"*, which is described in Section 5.4$_{100}$. Moreover, the taxonomy's dimensions give very good abstractions for the configuration decisions.

Other taxonomies contain the configuration items itself. They contain the instantiated scheduling scheme, synchronisation algorithm, driver, or protocol. Here, these items are regarded implicitly on the lowest level. Hence, they are candidates for configuration. Thus, the taxonomy presented here and their configuration items are often mixed in other taxonomies, although they have direct or indirect impact on the operating and communication system. The taxonomy presented here contains only indirect properties which are candidates for a requirements specification, whereas the direct properties are candidates for the description of the configuration space. The dependencies from indirect to direct properties can be regarded as the strings that have to be pulled in order to change from one configuration to another.

Pulling the strings of a puppet can be compared to answering questions of the requirements specification. The string is replaced by a *user primitive* that will be used by the application. In order to change the configuration of a puppet the strings have to be pulled. For the AND/OR-DAG this means to select appropriate *user primitives*. The distance with which the string is pulled can be compared to the parameters of the *user primitive's* usage or the number of a specific *user primitive* call.

In this chapter we presented a motivation for the graph- and structure-based configuration approach in TEReCS. We argued that the refinement of requests from the requirement specification leads naturally to sub-requests. Hereby, the sub-requests can be alternatively be solvable or they must all be solved, like in a problem reduction graph. The natural representation for this in general is an AND/OR-DAG. The sub-requests form again other dimensions of the taxonomy. Mutual influences between different dimen-

interrupts/polling	$\Rightarrow$ context saving for interrupts yes/no
task creation	$\Rightarrow$ online acceptance test $\Longrightarrow$ EDF, EDF*, server
guarantee/best effort	$\Rightarrow$ hard/firm scheduling
multi-/mono-programming	$\Rightarrow$ scheduler yes/no
scheduler	$\Rightarrow$ Non-RT scheduler, RT scheduler
Non-RT scheduler	$\Rightarrow$ FIFO, RR
RT scheduler	$\Rightarrow$ RM, DM, EDD, LDF, EDF, EDF*, server
periodic tasks	$\Rightarrow$ timers and periodic scheduling
timing constraints	$\Rightarrow$ synchronous activation , periodic tasks, aperiodic tasks, mixed periodic/aperiodic, deadline, preemption, precedence constraints
timeouts/sleeps	$\Rightarrow$ timers
periodic scheduling	$\Rightarrow$ RM, DM, EDF, EDF*
deadline = period	$\Rightarrow$ RM, EDF
aperiodic tasks	$\Rightarrow$ EDD, LDF, EDF, EDF*
synchronous activation	$\Rightarrow$ EDD, LDF
preemption	$\Rightarrow$ EDF, EDF*, dynamic server
precedence constraints	$\Rightarrow$ LDF, EDF*
mixed periodic/aperiodic	$\Rightarrow$ EDF, EDF*, server
response time improvement for aperiodic tasks	$\Rightarrow$ server
server	$\Rightarrow$ fixed priority or dynamic server
resource constraints	$\Rightarrow$ PIP, PCP, SRP
periodic messages	$\Rightarrow$ channel-oriented
channel-oriented	$\Rightarrow$ bandwidth allocation and management
aperiodic messages	$\Rightarrow$ packet-oriented, channel-oriented
message preemption	$\Rightarrow$ detection, acknowledgement
order preserving	$\Rightarrow$ sequencing, FIFO
peer-to-peer	$\Rightarrow$ fixed routing
multi-cast	$\Rightarrow$ message duplication or bus
broadcast	$\Rightarrow$ message duplication or bus
client/server model	$\Rightarrow$ alternative receive on multiple input channels
packet size $<$ message size	$\Rightarrow$ splitting
guarantee	$\Rightarrow$ acknowledgement and error correction
graph mapping: communications into topology $\rightarrow$ hopping	$\Rightarrow$ routing
hopping	$\Rightarrow$ forwarding
message buffering	$\Rightarrow$ mailboxes
dynamic channel creation	$\Rightarrow$ online bandwidth management

Tab. 5.3: From taxonomy to configuration items.

sions of the taxonomy exist. This is the reason, why additionally constraints must be integrated into the model. They help to solve alternative requests and to correlate the taxonomy's dimensions.

5.6 Contribution of the Chapter

This chapter presented a taxonomy for real-time systems that leads to a model for the design space description for the configurable options of a real-time operating and communication system. It motivates why the solution, which is used in TEReCS and that is presented in the next chapter, uses a graph- and structure-based configuration approach besides the integration of constraints. Additionally, the idea of the integration of application specific knowledge into the configuration process is presented as the *"Puppet Configuration"* approach. This seems to be very suitable for the configuration of service platforms like operating and communication systems.

CHAPTER 6

TEReCS

The real-time system's designer now has the techniques available to engineer systems rather than just to build them and then see if they meet their requirements during extensive (and expensive) testing.

Alan Burns, 1993 [23]

This is the main chapter of the thesis. Herein, all of the new ideas and concepts of TEReCS (**T**ools for **E**mbedded **Re**al-Time **C**ommunication **S**ystems) are presented. Also, the solution for the problem of Section 2.2_{12} is given.

The main objective of TEReCS is the generation of optimally adapted operating system instances for a parallel service platform, which serves a distributed embedded application. The communication dependencies between the distributed tasks are mainly considered. Moreover, in the focus of TEReCS lies the desire to support an optimal and temporal correct communication inside the distributed parallel machine of the embedded control units.

In the first two sections of this chapter an overview about TEReCS and its concept is given. The model and the method for the description of the knowledge bases and the inputs for the configuration process are presented in Section 6.3. In this model the detailed information about the structural properties of the configurable communication and operating system are hidden from the user. The configuration process, which is described in Section 6.4, is extended by a hierarchy concept in Section 6.5. After the basic concepts have been presented the languages for the inputs to TEReCS are defined (see Section 6.6). Additionally, it will be explained in which way TEReCS is applied to the customisable operating system library DREAMS. Hereafter, the knowledge transfer from the application to TEReCS in order to configure the operating systems and the communication system is explained (see Section 6.7). In Section 6.8 the concept of the real-time analysis in TEReCS is presented. The chapter ends with some examples that show the mutual influence of

the configuration (see Section 6.9) and analysis phase (see Section 6.10) in TEReCS.

6.1 Overview

TEReCS covers two main aspects for the development of embedded real-time systems. First, TEReCS is a software synthesis system for the generation of distributed (multi-processor) real-time operating systems (RTOS) and a real-time communication system (RCOS) from an operating system library kit. Second, it supports and ensures real-time properties of the system. The main goal of the system is the generation of an individual RTOS for each node of the target system. The RTOS should be optimally adapted to the requirements of the application. Additionally, real-time requirements of the RCOS are verified before the final code for the target of the system is generated. This helps to save test runs and shortens dramatically the design cycle of an application.

Based on the formal description of the complete valid design space of the RTOS/RCOS the system tries to enrich the under-specification of this design space by integrating a requirements specification. This is done by creating a valid configuration for all required components of the system. Thus, the generator of the system, in fact, is a configurator. The description of the design space is given in an AND/OR-graph which is based on the idea of AND/OR-trees (modelling paradigm). Nodes of this graph represent *services* of the RTOS/RCOS. The directed edges stand for dependencies between them. By this way the graph models the dependencies between all RTOS/RCOS services with all it valid alternatives. These alternatives are the basis for generating different configurations from different requirements specifications. The configuration process´ main task is to get rid of all OR-dependencies, i.e. alternatives of the graph.

This means, a final configuration is a sub-graph of the design space description without any alternative. The OR-decisions are solved on the basis of costs, priorities, resource accessibility and the total amount of processes and communications, which make use of the services. In addition, special edges in the graph represent constraints like forcing, preferring, or prohibiting the use of a service under special conditions.

The methodology to achieve these configurations is to integrate step-wise knowledge from the application into the under-specification of the RTOS design space. This knowledge is seen as a requirements specification to the RTOS/RCOS and is used as an input for the configurator. The requirements specification describes internal knowledge about the distributed application, which can be exploited in order to optimise the RTOS/RCOS. The requirements specification consists of:

- Description of the hardware components and its topology of the target system (processors, devices, media)
- Process mappings of the application tasks onto processors
- Demands of RTOS *services* (application programming interface [API] accesses) of all processes and the properties of these
- List of all communications between the processes with its properties

All these inputs can also be modelled as graphs having processes, processors and API *user primitives* as nodes and their interdependencies as edges with assigned properties.

The algorithm for the generation of a configuration comprises mainly the following two main goals:

1. Search for a functional correct configuration for the RTOS of each processor
2. Validate the real-time constraints of the application for *services* of the RCOS

If the validation fails, another valid and correct configuration will be searched. Thereby, the correct configurations will be generated with increasing costs. If no valid configuration exists, the system fails, but it gives hints to the user indicating for which reasons it has failed.

In order to find a correct configuration the algorithm combines all graphs to a super-graph (see Figure 6.2). This super-graph should then contain all required information and is optimised. Here, optimisation means to find a minimal and cost optimal sub-graph that fulfils all functional and temporal requirements and contains no alternatives. Such a sub-graph describes a valid configuration.

In order to make a temporal validation of a correct configuration the timing constraints for all communications in the system are checked. Therefore, as a first step, a schedulability test must ensure that the load of all communications devices and media is less than a certain individual limit. If this passes, then for each communication connection its end-to-end communication time is calculated. This end-to-end communication time must be lower than or equal to its specified deadline. Schedules of messages for all communication devices and media are created in order to precisely calculate the end-to-end communication latencies.

6.2 Concept

A communication system like an operating system can be seen as a service platform. But on the other hand it should not be modelled in the classical way of a monolithic client/server architecture but as a very fine granular adaptable system with a lot of dependencies between its service components. In this model, all services, their dependencies and properties must be specified. All this information is stored into a database. A service platform can be built from components in this database.

Besides the model of software components, the hardware also has to be described. Both models, the one for software and the one for hardware components, are not considered in isolation. Instead, the connections and transitions from one into the other description are modelled, too. Thus, one can calculate resource loads and delays caused by services mapped onto the hardware.

The overall synthesis process works as follows: Starting with a specification of the required communication behaviour (*Process and Communication Graph* (PCG)) and the given hardware topology, that is defined in the *Resource Graph* (RG), a generator and configuration tool tries to assemble an execution platform from the components of the database. The description and specification of software and hardware components as well as their relationships form a sort of expert knowledge.

In order to describe the experts knowledge and to define the requirements specification and the hardware topology, a model for the communication, the software, and the hard-

ware has to be developed. In the next few sections the models for hardware, software, and their inter-relationships are presented.

6.3 Model

The model must describe the expert's domain knowledge. The data model, which defines this database, must be able to describe the hardware components, the software (skeleton) services, their dependencies, and the requirements of the application.

6.3.1 Hardware

A real system consists not only of software but also of hardware components. We distinguish three different types of hardware components:

- central processing units (CPUs)
- devices
- media (physical communication links)

Each type of a CPU (*Central Processing Unit*), device, or medium is represented by a node in the *Universal Resource Graph* (URG). The nodes are regarded as *resources*. Edges between resources show that these resources can be connected, meaning they are compatible. Only one restriction exists: CPUs cannot be connected to media. Edges within this graph are directed showing the possible direction of data flow. Each CPU, device or medium type used in a system must have a representation as a node in the URG. Every hardware entity is regarded as a resource with *ports* of different types. Ports are the coupling points for connecting resources with each other.

Programmable resources (CPUs) are hardware entities for which software has to be generated. All entities have special characteristics, which have to be stored and validated by the tool, too. For example, ports of the type CAN (*Controller Area Network*) bus can be connected, provided that they operate with the same bit rate and data protocol. These are parameters of the port-type and have to be checked during configuration.

6.3.2 Software

Each fragment of the communication software is regarded as a *service*. This fragment is normally a function or a method of an object. The methods/functions are cross-linked by method/function calls. This network of dependencies has to be represented so that only permissible combinations can be synthesised.

Like hardware entities, software services are modelled by service objects with two types of interface points. The entry point is the *Service Access Point* (SAP). A service demands other services by its *Service Request Points* (SRP). SRPs are connected to SAPs. They can only be connected if the SAP and SRP are of compatible type, e.g. if the signatures are compatible. A SRP can have multiple connections to several SAPs, representing usable alternatives. The SAP can be seen as the signature of a function and the SRP as a call of the referenced function.

A special kind of software are the *device drivers* and *interrupt handlers*. They are the connections from software to hardware. Device drivers read/write controller registers instead of calling functions. Interrupt handlers are activated by hardware instead of being called. These relationships are modelled by dependencies between software services and hardware resources. Device driver services have a *Hardware Request Point* (HRP) instead of a SRP to represent their dependability to a hardware component. This is modelled by a *usage*-relation between the HRP and a special type of port. On the other hand, hardware can have a special port, connected to an *Hardware Access Point* (HAP) of a service. This is to specify the demand for activating a kind of interrupt handler service.

6.3.3 Inter-Component Model Structure

The graph of the *services* together with the SAPs, SRPs, HAPs and HRPs is named *Universal Service Graph* (USG). Within this graph all interdependencies between all software components are described by special edges; including:

- calling dependencies: a service calls another service,
- activating sequence dependencies: a service must run before/after another service,
- excluding/including features: the use of a special service excludes/includes the use of another service (without calling it).

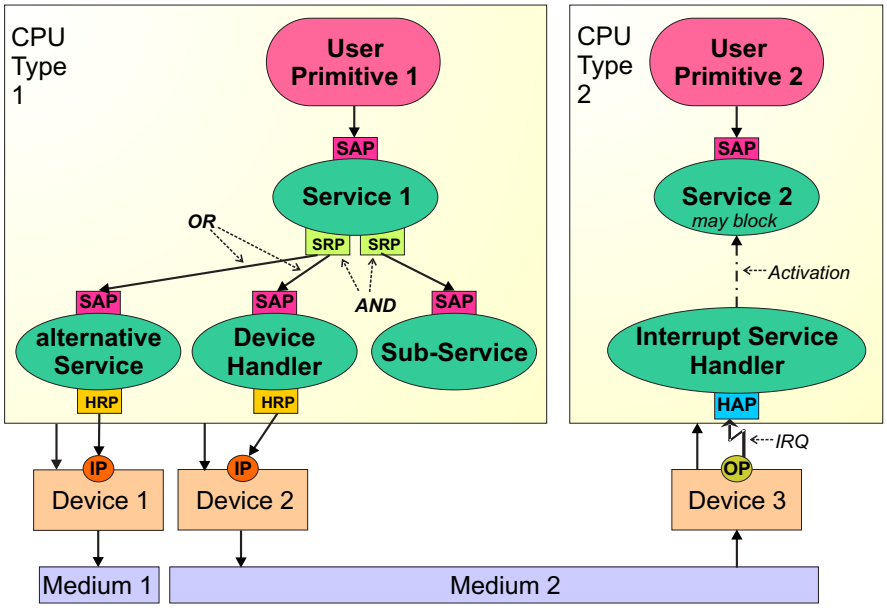

Fig. 6.1: Example for an Universal Resource Service Graph.

The USG is a directed graph of all permissible calls and alternatives of services and calling paths including connections to the hardware (see Fig. 6.1). The USG can even be seen as a demand graph: a *service* needs other services or hardware resources. This graph represents the overall expert knowledge about the services, their properties, and their inter-relationships.

Additionally, we have defined *user primitives* that have only a *User Access Point* (UAP) as the root and leaf nodes of the USG. These primitives are specified for sending and receiving, and represent well defined API (*Application Programming Interface*) functions. Each of those primitives is assigned to a special type of SAP. The root nodes of the graph are the send primitives, whereas the leaves are the receive primitives. Both form the communication API.

The USG and the URG are the basic data structures which describe all components out of which a system can be build. The combination of the USG and the URG leads to the *Universal Resource Service Graph* (URSG) (see Figure 6.2). Within this Graph the attributes HAP and HRP of a *service* have directed edges to ports of devices. The graph represents a network of software services and hardware entities. The OSI reference model [66] for communication software with one sender and one receiver can be unfolded in the same way. The application calls a service of the API, the message passes down the communication stack to the physical layer and goes up again through the layers to the receive-API.

6.3.4 Constraints

Another very powerful mechanism in TEReCS is the constraint definition. Besides the service dependency description of the AND/OR-Graph and the solving of the OR-groups based on the cost function also constraints can influence the decision about the selection of appropriate alternatives of an OR-group.

Technically, a constraint in TEReCS defines a dependency between an item of the URSG and the selection of an alternative service in an OR-group. In TEReCS the source of a constraint can be a service, an SAP, or the port of any device. The OR-group is represented by an SRP with multiple outgoing connections to SAPs. Such SRP is then also identified to be a *choice object*. Each of the SRP to SAP connections is identified to be a *choice element*. For this reason, in TEReCS constraints are visualised as directed edges between the source item and the edge of an SRP to SAP connection. The SRP must define an OR-group. Therefore, edges from the SRP to multiple SAP must exist. Each of these edges (choice elements) can be the destination of a constraint edge.

Semantically, a constraint in TEReCS is guarded by its source item. This means that the constraints is activated if, and only if, the source item has a special status. When the constraint is not active, then it has no impact on the selection of the alternatives. The status of the source item can be that

- it is selected to be used in the configuration
- it requires a special architecture, which have to be provided by the CPU for which it has to be selected
- it (a service) requires different I/O ports
- it (an in-port) requires only the same out-port

The constraint can also be activated, when the above described guard is negated. For example, when the source item is *not* selected to be used in a configuration. Especially, the last two states of the source item are used to configure the communication system. For instance, a routing service is not required, when all messages using a specific service or in-port also use the same out-port.

If the constraint is activated, then it can influence the selection of the specific SRP to SAP alternative in four different ways:

Prefer: Increments the primary priority of the choice element by one.

Favour: Sets the primary priority of the choice element to the maximum priority of the service, which belongs to the SAP. The maximum priority is defined a priori in the URSG and per service.

Force: Sets the primary priority of the choice element to the maximum priority plus one of all elements of the choice object. Thus, this choice element will be selected in any case.

Prohibit: The choice element is marked "erased", which means that it cannot be selected.

If the constraint is deactivated, then the priority of the choice element is restored to its original value and it is again marked eligible for selection. After a constraint is activated or deactivated, the appropriate choice object is again solved. In the Subsection "Configuring Means Choosing of Alternatives" on page 118 we describe in more detail, in which way a choice object is solved, i.e. the appropriate alternatives of the OR-group are chosen.

By using these constraint types the selection of one alternative of an OR-group can be preferred, favoured, forced or prohibited. This can depend on the use of another specific service, the access of an used service by a specific SAP, or the request of a service or in-port for the same or different out-ports.

The constraint solving algorithm in TEReCS is very simple. After all services are selected and all choice objects are solved, the activated constraints are handled and the choice element, which is influenced by the constraint, is again solved. Because other services are selected due to the constraint activation, all successors of deselected services must be deselected and successive services of selected ones must also be selected. This implies that complete sub-graphs must be de-selected or re-selected. This again implies that other constraints may be activated or deactivated. Thus, the activation or deactivation of the constraints and the de-/re-selection of services must be repeated in a loop until no constraint changes. Fortunately, the amount of iterations for this loop is limited by the number of services of the URSG. If more iterations would be executed than services are present, then the definition of the constraints in the URSG contains at least two constraints, which mutually influence each other. For instance, the activation of constraint a implies the deactivation of constraint b. The deactivation of constraint b implies the deactivation of constraint a. The deactivation of constraint a implies the activation of constraint b, that again implies the activation of constraint a. Thus, we have a cyclic dependency and at least two services that are controlled by the constraints a and b are periodically selected and deselected. However, such a miss-specification, which eventually spans over several constraints and services, will be detected and an error will be printed.

6.3.5 Specification of System Requirements

The *Process and Communication Graph* (PCG) describes the distribution and the communication behaviour of the application. Each node of the graph represents a process of the application. An attributed directed edge within the graph shows a possible *communication connection* between these processes: The process at the beginning of the edge acts as a sender and the process at the end of the edge acts as a receiver. The edge is attributed

by two *user primitives* (the sending and receiving one), the amount (and eventually the type and layout) of the data and the period or maximal allowed latency for the communication.

6.3.6 Example

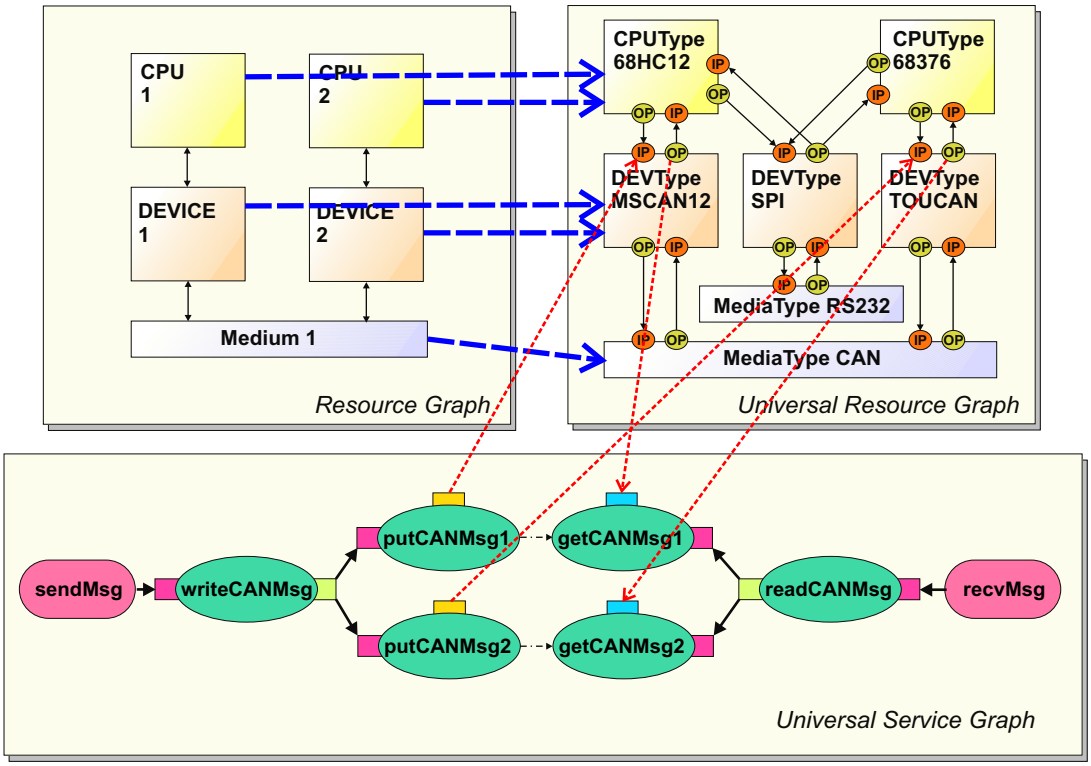

Fig. 6.2: Example for the integration of Universal Service Graph, Universal Resource Graph and Resource Graph.

In Fig. 6.2 a simple example for an USG, an URG, RG, and their interconnections is given. The system hardware consists of two CPUs connected via a CAN medium. The URG describes not only the types of the used resources, but contains also other CPU and device types. For this reason, the USG contains alternative calls of device drivers for writing messages to and reading messages from different device types. That way, appropriate services (using existing devices) have to be selected during the search of feasible calls. Furthermore, in combination with the RG the routing problem in the resource topology can be solved.

6.4 Design Process

This section is divided into three subsections. In the first subsection the methodology is presented, which is the basis for the design process in TEReCS. In the second subsection the synthesis process is explained. The synthesis of code for the operating and commu-

nication system of each node of the system is implemented as a configuration problem. Here, configuration will be described as a graph problem (with a structure-based concept hierarchy, see Section 3.3.1$_{29}$ and page 39), since dependencies of software parts can often be represented by dependency graphs (like used in UML specifications [109]). The second subsection contains a quite abstract description of the configuration. In the third subsection the configuration algorithm is described in more detail.

6.4.1 Methodology

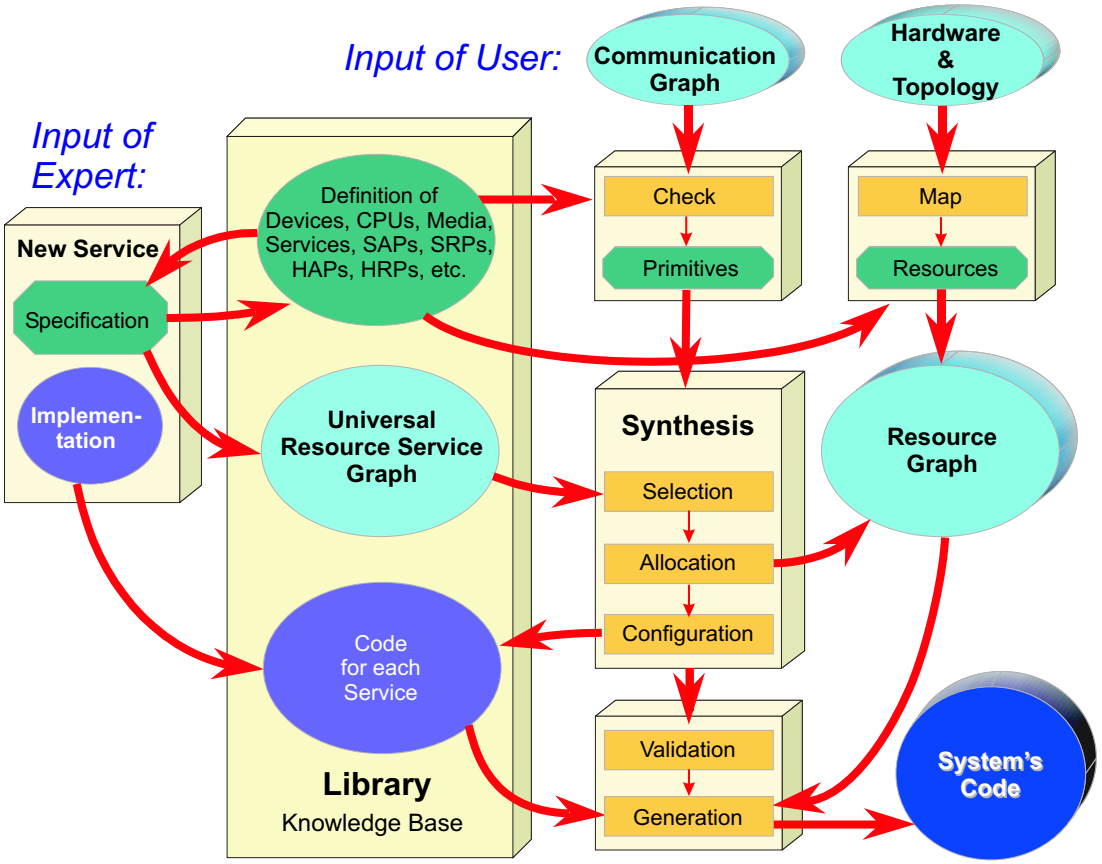

Fig. 6.3: Methodology in TEReCS for creating a configuration and assembling the code for a run-time system.

Central to our methodology (see Fig. 6.3) is the library representing a knowledge base. The library consists of three sections: The description of objects, the URSG and the implementations for each *service* (maintained in source or object code format).

Experts are responsible for creating new services. They describe them with basic descriptions taken from the library, insert them into the URSG and define their properties, relations and implementation. Supported by the database management tool TEDIT, experts are enabled to manipulate and edit the library.

The library with its URG (which is a subgraph of the URSG) serves for users of the system as a basis for modelling the RG and as a source for the synthesis process. In order to start

the synthesis process they have to define the communication behaviour of the distributed application and the hardware topology. From the hardware description a concrete RG is derived. The communication demands (PCG), the RG, and the URSG, as well as the implementations of the *services* are the inputs for the synthesis.

6.4.2 Synthesis

After presenting the basic data structures used for the description of the SW-components the synthesis process (see Fig. 6.4) is briefly introduced.

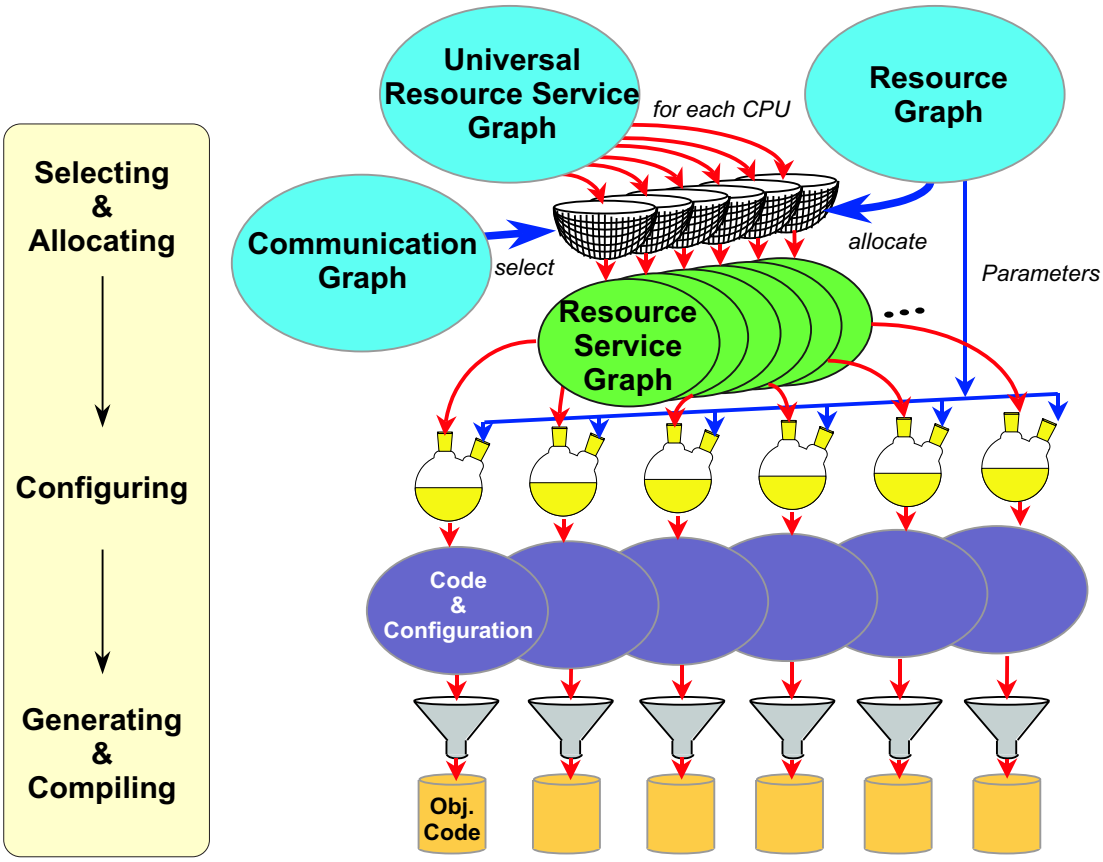

Fig. 6.4: Synthesis process in TEReCS.

The first step (**graph composition**) of the process combines all these three graphs (URSG, RG and PCG) to one hybrid *Resource Service Graph* (RSG) for the required system. This is done in the following manner:

- All resources in the RG are connected to their corresponding types in the URG, resp. URSG.
- Each *user primitive* in the PCG is assigned to its corresponding *user primitive* in the USG, resp. URSG.
- All paths in this graph starting in a *user primitive* that is not used are deleted.

- All paths in this graph with services leading to *resource types* that are not connected to *resource instances* of the RG are also deleted.

This strategy results in a graph where all non-accessible services are erased. The resulting RSG consists only of probably required services and resources for the demanded system.

The second step of the synthesis process contains the main algorithm for finding the really needed services and resources. The main idea is that this step works like hardware synthesis. For this reason, the synthesis is divided into the following sub-processes: selection and allocation of services, configuration, validation, and code generation. Here scheduling means selecting hardware alternatives or planning communication slots onto the media, which allow a more detailed analysis of latencies and delays.

The first sub-step (**Selecting & Allocating**) determines the really needed *services* and resources. This is done by finding an optimal path in the graph from the sending *user primitive* to the receiving *user primitive* along each edge of the PCG. User primitives lead to services, services lead to other services or device types, device types lead to concrete devices on this CPU, devices lead to media, media again to devices, devices to device types, device types to services and services again to user primitives (see Fig. 6.1). That way, by searching a path through this graph, also the routing-problem is solved. Because of possibly existing alternatives in the USG and in the RG more than one path can be found. The optimal path is chosen by using a cost function or because services in a path are excluded by already selected services. Thus, the order of the communication edges that are visited has an impact on the solution. By finding a path the selected services can be allocated onto the corresponding CPUs. Furthermore, it is known which services and resources are used by which communication connection. Thus, the resource loads of all devices and media (and even CPUs, if we know the generated load by each service) can be determined. This information has to be validated in order to ensure the demanded functionality, to guarantee that all resource-loads are lower than a given bound (e.g. 100%), and to meet all communication deadlines.

The second sub-step (**Configuring**) generates data which are needed for the *services* (e.g. a routing table for a routing service) and assigns concrete values to free parameters of the services (e.g. the baud rate used for a medium).

The third and last sub-step (**Generating & Compiling**) assembles the code for each CPU. This is accomplished by generating Makefiles, which compile the needed services and generated data into a library, for each CPU.

6.4.3 Configuration Algorithm

In this section the configuration algorithm is described in more detail. Especially, the sub-steps “Selecting & Allocating” and “Configuring”, that have been presented in the previous section, are explained. This section describes in which way the *Universal Resource Service Graph* (URSG) is traversed in order to select all required services for the configuration of the operating and communication system of each processor node in the system. We also define the cost function that is used in order to solve the selection of the alternatives of all OR-dependencies in the URSG.

The configuration algorithm works in four phases:

1. Initialisation and input of the service database, the hardware topology, and the requirement specification
2. Selection of all possibly required services for each processor node
3. Make decisions for the OR-groups and the constraints for each processor node
4. Generate the configuration description for each processor node

In phase 1, which is the initialisation, the service database with its *Universal Resource Service Graph* (URSG) is read from a text file. Also in phase 1, the hardware and topology description within the *Resource Graph* (RG) is read from a text file, and, last but not least, the requirement specification within the *Process and Communication Graph* (PCG) is also read from a text file.

In phase 2, which is repeated for each processor node (CPU), the possibly required services are marked for selection ("Selecting & Allocating"). They are selected for being implemented on a specific processor node. This phase is named "selecting phase". This is mainly done by traversing each path from a primitive, which is used in the requirements specification, down to every reachable leaf (which is a service that requests for no sub-service, see Figure 6.5). Thereby, all alternatives of one OR-group in the URSG are stored in a *choice* object for the specific processor node. Additionally, each defined constraint, which possibly must be solved, is stored as an *inhibitor* object. (For the definition of the constraints see Section 6.3.4$_{110}$.) The result of this selection is a specific sub-tree of the complete URSG for each processor node. Note, that this sub-tree contains still all possible alternatives of OR-groups.

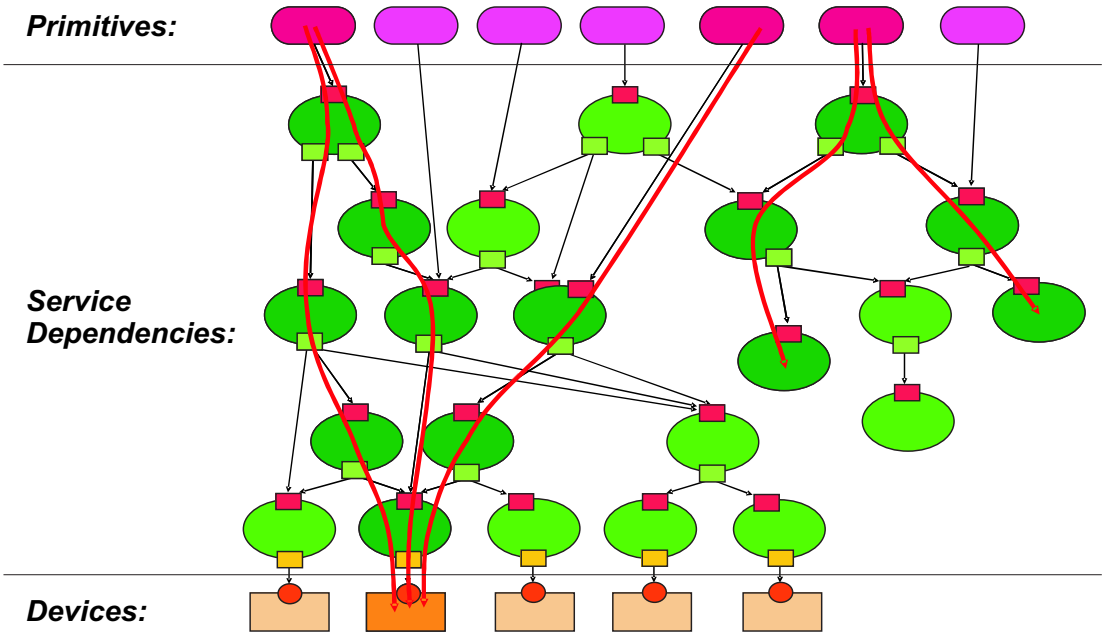

Fig. 6.5: *"Puppet Configuration"* finds a solution by traversing all paths from selected primitives down to all terminal nodes, whereby all OR alternatives are resolved.

In the 3^{rd} phase all OR-alternatives of the *choices* are determined. This means, that only as many services remain in each OR-group as required (often only one). This phase is the "configuration phase". For each service A of an OR-group, which has not been selected, all successive sub-services, which can only be reached via that service A, must also be

deselected. Additionally, all constraints are handled in this phase. Only activated constraints must be solved. A constraint is activated, if its source item, which controls the constraint as a guard (and which can be any service, a *Service Access Point* (SAP), or an in-port of a device), is selected. A constraint is not considered, which means it is deactivated, when its source item is not selected. Constraints influence alternatives of OR-groups (*choices*) to be selected or not. The constraint solving has already been described in Section $6.3.4_{110}$.

In the last and 4^{th} phase the configurator writes the configuration description for each processor node, the makefile for each processor node and the net description for the routing of the communications onto the links into text files ("generating phase").

The reason for splitting the main configuration algorithm into the two phases for "selecting" and "configuring" is the following. In order to decide for an OR-group (*choice object*), which alternatives have to be chosen, a cost function is required. This cost function determines the alternatives of an OR-group that must be selected. It is the main part of the "selecting" phase to determine the costs of each selected service and SAP. Additionally, it determines the number of request for each service. This is the number of tasks and communication requests, which require that service.

Cost Function

The cost function is hierarchically defined. Each service and each SAP of the URSG have assigned some initial costs. They produce this initial cost, when they are possibly selected for a configuration. The total costs of a service or an SAP are defined as the sum of the initial costs of all it succeeding services and their used SAPs of its sub-tree of the URSG. This means that for each path from a service down to every reachable leaf service the initial costs of each SAP and each service on that path are added to its own initial costs. This results into a cost labeling of each possibly selected service. A service may only be *possibly* selected, because in this phase *all* alternative services of a OR-group are selected. If a service refers to an OR-group, then it inherits the maximum costs of the alternatives of this OR-group (see Figure 6.6). The services may be deselected during the solving of the choice objects ("configuration phase").

While the maximum of the total costs of each service are determined, also the depth of each service is calculated. The depth is defined by the maximum path length of a path from this service down to any leaf service. The path length is defined as the number of services that are visited on the path. An OR-group is modelled as a *Service Request Point* (SRP), which refers to more than one SAP. The depth of the corresponding choice object, which represents this SRP, is defined as the minimal depth of a service to which it belongs.

After the "selecting phase" all choice objects are handled during the "configuration phase". The choice objects are handled according to their depth, starting with the highest depth. This is done, because the costs of the service A, which owns the SRP that defined the choice object, are re-calculated upon the actual selection of the choice. That means, that the costs of the deselected and succeeding services are removed from the total costs of the service A. This assures that the cost labeling contains only costs of the selected services after each choice object is handled.

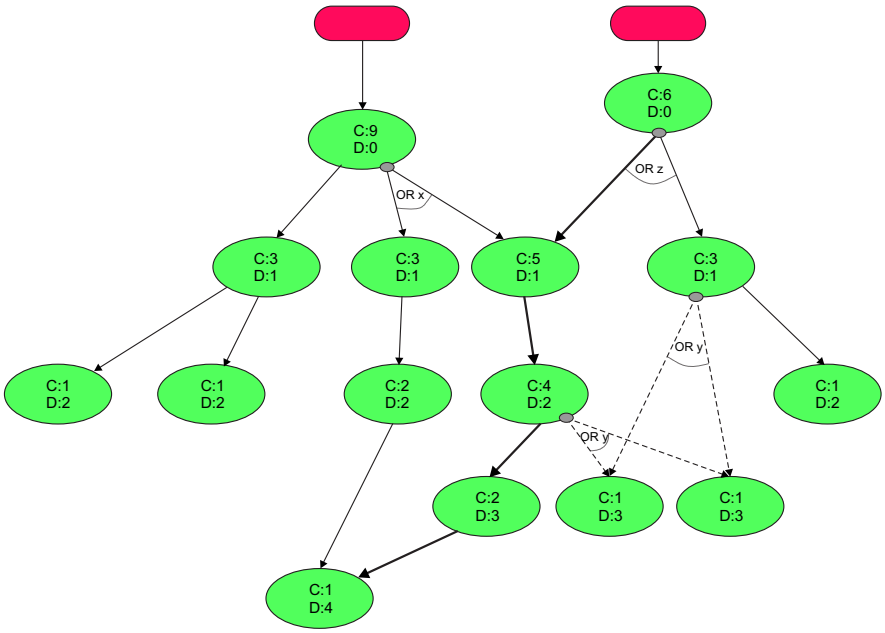

Fig. 6.6: Example for the calculation of the costs (C) and the depth (D) of each service assuming each service has the initial (self) costs of 1 and each SAP costs 0. The bold lines refer to the longest path. The dashed lines refer to the same OR-group.

Configuring Means Choosing of Alternatives

The "handling" a choice object means to determine the minimal set of required services (often this is only one) from the complete OR-group. The decision about which services should be chosen is made on different levels. On each level some services are removed (deselected) from the OR-group. If there are still too much services selected, then the next level is applied. Actually, TEReCS defines six levels for the deselection of services from a choice object. Each service is assigned an initial primary and an initial secondary priority. The first level deselects all services, which do not have the highest primary priority of all the services in this OR-group. At the second level only that services remain selected, which have the highest number of requests by communications that are defined in the requirement specification. At the third level all services are deselected, which do not have minimal total costs of the cost labeling. At the fourth level all services are deselected, which do not have minimal costs that are defined locally by their choice object. This are the minimal total costs of all services that use the SRP that defines this choice object. At the fifth level only that services remain selected, which have the highest number of requests from tasks. If there are still too much services selected, then at the sixth level all services are deselected, which do not have the highest secondary priority of all the services in this OR-group.

After a level is applied, too less services might remain in that OR-group. In this case all services with minimal total costs are again selected. After this procedure it can still occur that the number of selected services is either too less than the defined minimal number or too high than the defined maximum number. In this case TEReCS does not succeed to solve the choice. A warning will be given. But the practice showed that the URSG can be defined in such a way, so that always the choices can be solved.

Considering Communications

The communications, which are defined in the requirement specification, request for two primitives that are used for the sending and receiving of the message, respectively. Both primitives can be required on two different processor nodes. Thus, it is the task of the configurator in TEReCS to select all required services on the sending node, the receiving node, and eventually on all intermediate nodes (hops). For this reason, the configurator of TEReCS determines the routing of that communication, which determines all used nodes. Additionally, the devices are determined that have to be used for the local sending and receiving of the message on each node. The configurator selects the services for a path through the URSG from the sending primitive to the first sending device (out-port). On every intermediate node it selects the services of the path from the incoming device (in-port) to the next outgoing device. On the receiver's node the configurator selects a path from the receiving primitive to the receiving device (in-port). Remember, that the ports of the devices are connected to a *Hardware Access Point* (HAP) or a *Hardware Request Point* (HRP) of a service in the URSG.

The path selection on the last hop is made from the receiving primitive down to the in-port. This is opposite to the message flow and can arise a problem. An incoming message is read by the device driver, which has been activated by an interrupt of the in-port. The message is then stored into a kind of mailbox. Therefore, the device driver requests for some sub-services until a service of the mailbox stores the message into the memory. At the other end of the communication, the receiving primitive requests for some services until it reaches the read service of the mailbox in order to retrieve the message. Thus, we have to follow the service dependencies on the receiving node from both directions. This case is also handled by TEReCS. The store service of the mailbox have to be marked as a *server service* for the read service of the mailbox. The read service have to be marked as a *client service* of the store service. In this way, TEReCS can select the services of a path from the receiving primitive down to a client service, and the services of a path from the in-port to the appropriate server service.

In some environments the communication is handled by some dedicated communication co-processors. In this scenario the message is stored on the co-processor's memory and has to be explicitly requested from there. TEReCS is also able to handle this. The TEReCS's specification model allows to place the server service for a receiving primitive to any directly neighboured node. Then, TEReCS routes a path from the client service over the communication link to the server service on the neighboured node and back to the client service.

The modelling of a real client/server-communication between two not directly neighboured nodes is yet not possible in TEReCS. However, it is intended by TEReCS to model each directed communication separately. The above presented functionality is only intended to be used for a scenario with a communication co-processor.

6.5 Hierarchical and Dynamic Configuration

Up to now the configurator requires the complete design space description of the system, which should be customised. This is provided by the *Universal Service Graph* (USG). The USG is flat and can be very huge. Thus, the process, respectively the scanning and

traversing of the graph can use a lot of time. For example, think of a graph, which does not only describe one operating system but a lot of different operating systems and even some middleware software like CORBA services, protocol stacks, database systems, application specific domain services (video/audio codecs, controllers), etc. Therefore, a methodology was developed to prompt how the configuration process can be accelerated.

Hierarchy is a very powerful structuring paradigm for information hiding. If the USG is extended by hierarchy, then this can be exploited during configuration. When a well-defined hierarchical clustering mechanism for the object graph joins the process preceding the configuration step, the advantages are manifold. The rules of clustering aid the configuration process by not only encapsulating the objects as clusters and sub-clusters but also streamlines the configuration process. The configurator chooses optimised paths by taking advantage of having clustered objects, for a given set of reasonably complex requirements. Also, when a change in the set of requirements takes place, it directly attacks the cluster in question with the aid of rules instead of doing the global search for identification of the involved objects. Added advantages result from keeping the rules of clustering inline with the configuration process itself. In this case the entire process is able to support different clusters for different applications.

Hence, a need for a specific hierarchy or clustering mechanism, which is *based on the connectivities and types of relationships of the USG* is felt, which tries to improve the process of configuration in totality. The relationships, *inheritance*, *membership*, *using*, and the *constraints* form the basis for the connectivity, form clusters and further form the configuration. The rules governing the cluster formation, which are mentioned later in this section, weight each relationship against others, resolve conflicts and provide a connected cluster. The basic idea behind the novel tool-oriented clustering paradigm is the following: Any small change in the requirements specifications will affect in visiting the particular cluster(s) alone instead of the whole USG.

While formulating the rules for clustering, which support the configuration process, care has been taken not to encroach into the process of configuration. That means, it is manifest that both the process of configuration as well as the hierarchical clustering process have distinct objectives and functions.

6.5.1 Hierarchical Clustering

Services of the USG can be grouped to many clusters based on their connectivity. The graph can be transformed in that way, that all services of a cluster are represented only by one higher-level service. This service of a higher level is called *super-service* or *cluster*. All relations connected to a service of the cluster and any service outside of the cluster are redirected to the *super-service*. All services and relations inside the cluster are completely hidden. In order to support unlimited levels a cluster on level n can consist of normal services (at lowest level 0) and *super-services* of level n and is represented by a *super-service* at level $n + 1$ (see Figure 6.7).

At the lowest level the graph represents the original flat graph and consists of all *user primitives* described there. Because *super-services* are handled by the configurator equally to normal services, all customisable items of this level (visible alternatives) will be configured, when this graph is used as input for the configurator. Additionally, the configurator

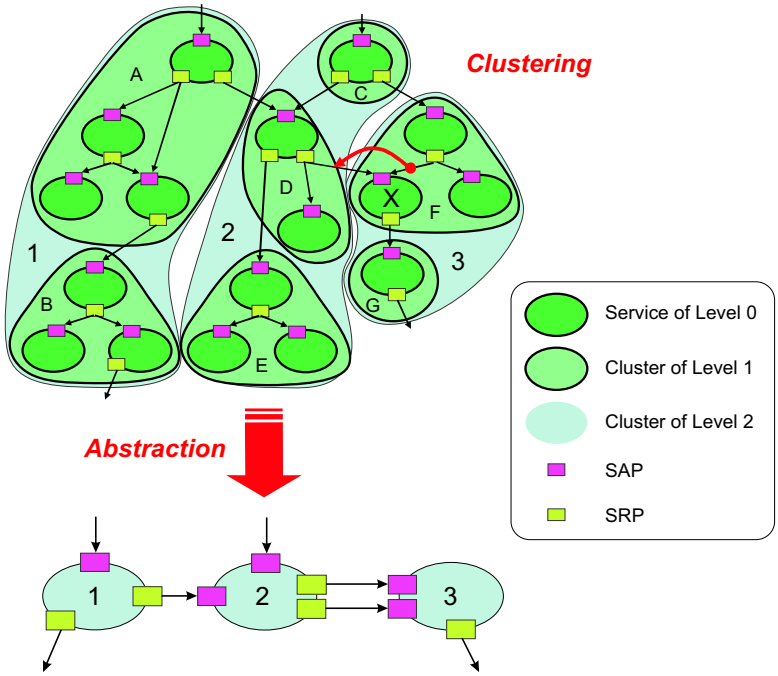

Fig. 6.7: Using hierarchy for hiding cluster graphs for *super-services*. The *super-services* are abstractions of cluster graphs.

gives the output, which cluster is required and which is not required. This means that only the required clusters need to be configured at their higher level. This works as follows: The SRPs of all incoming relations into this cluster are replaced by *user primitives*. All outgoing relations can nearly be ignored, because they are already configured at the higher levels. The newly and temporarily created *user primitives* and the sub-graph inside the cluster are used as input for the normal configurator. A new requirements specification is created from the configuration at the higher levels, by the rule: The primitive inherits its usages from the SRP. After the configuration for this cluster is ready, it can be integrated into the higher-level configuration by simple insertion (see Figure 6.8).

This methodology dramatically decreases the time for configuration, if a lot of clusters need not to be considered. Therefore, it is inherently important to have a good clustering of the original flat graph. But this strongly depends upon the requirements specification and the graph itself. Thus, rules are required that describe the way to define cluster borders or whether (super-) services should be joined into the same cluster or not. It has to be decided whether a dependency should cross such a border or not. This can be decided upon the properties of this dependency and the impact of the requirements specification onto this dependency and vice versa.

The rules resulting from our investigation are given as follows:

R1 Do not separate services, which are tied to another service with OR relationship. Because OR relationships are resolved by the configurator.

R2 Join services by looking for inheritance relationship first, if not membership relationship next. Consider using relationship only when no other relation exists. (This rule is purely heuristic.)

R3 If R1 and R2 are conflicting and both inheritance and membership carry equal weight (that means, the cost of both the relationships is same), then resolve the conflict by referring to rule R4 and R5.

R4 In conflict, if the involved relationships are of the same type, the small segment joins the bigger one. While partitioning, when a service in question faces a conflict, whether to join this cluster or that cluster when the costs involved are same, this rule becomes active.

R5 In conflict, a constraint edge or simply constraint, which prohibits or prefers or forces prioritised relationships, is used to decide the place of services under consideration. This rule gets priority over rule R4 when constraints are involved.

The *configurator-supporting-clustering* is dynamic in the sense that a service attached to one cluster for one set of a requirements specification would be placed in another cluster, which corresponds to another set of requirements specification.

```
Procedure Cluster-Configuration (graph USG, requirement
specification RS):
 1. // Clustering phase:
 2. Create set of clusters SC for graph USG taking RS
    into account;
 3. Remove all not required clusters (not touched by
    requirement specification RS) from SC;
 4. // Configuration phase:
 5. If SC has only 1 cluster then
    (a) G := Graph in Cluster C from SC;
    (b) USG.Configuration := Configure(G,RS);
 6. Else
    (a) Foreach graph G in cluster C from SC do
        (1) R := all requirements from RS touched by
            primitives in G;
        (2) RS := RS - R;
        (3) C.Configuration := Cluster-Configuration(G,R);
        (4) USG.Configuration += C.Configuration;
    (b) Enddo
    (c) G := Graph describing cluster connectivity in SC
        according USG;
    (d) USG.Configuration += Configure(G,RS);
 7. Endif
 8. Return USG.Configuration;
```

Fig. 6.8: Algorithm of the hierarchical (recursive) configuration process.

6.5.2 Dynamic Aspect

The hierarchical configuration process can be divided into two general steps: (1) Generate a good clustering and hierarchy for a flat graph. (2) Configure the system by following the hierarchy and exploiting the clustering. Even these two phases of configuring can

alternate in a hierarchical (recursive) algorithm. It means, after clustering and configuration have taken place at a certain level, both phases will be applied to the clusters of the next level, and the process continues up to the highest level (see Figure 6.8).

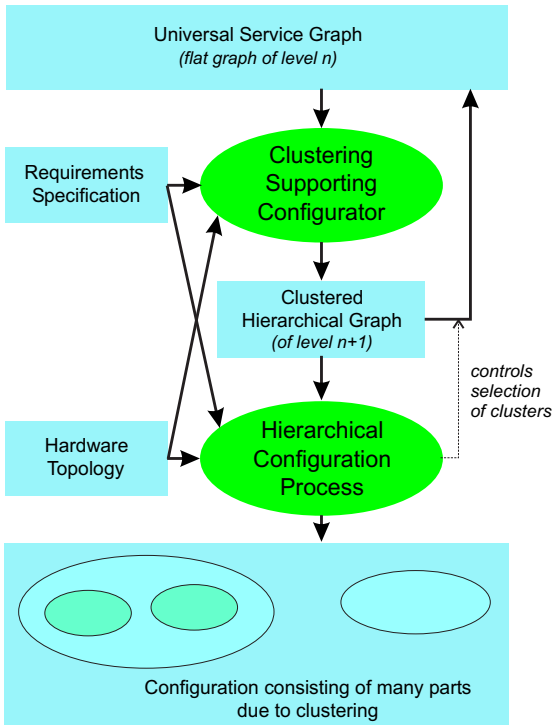

Fig. 6.9: Design flow for hierarchical clustering and configuring

The hierarchy and clustering of a given service graph has been designed not to be static for all given application scenarios. If it would be static, then the information covered by the hierarchy and clustering structure would just be redundant. This means that the hierarchy and clustering can be obtained before configuration by applying the partitioning algorithm. In contrary to this, in our approach clustering also depends on the requirements specification, because constraints are strongly influenced by the requirements specification. Thus, different requirements specifications result into different clustered hierarchical graphs providing additional information for the control of the configuration process.

This can also be achieved by giving a clustering which is not complete, respectively, which contains non-determinism in form of design freedom. For example, a service can be selected to be a member of more than one cluster (dynamic or fluidic cluster border). But for configuration it must be uniquely assigned to only one cluster. So, there exists an open problem, which still must be solved before the basic configuration can start. This incomplete or non-deterministic dynamic hierarchy and clustering must be transformed into a static and well defined hierarchy and clustering. This can be done by applying also application specific knowledge from the requirements specification, like rule R5.

6.5.3 Results of Clustering

The purpose of the clustering phase is to support the configuration phase for faster execution and to support for the reuse of already made sub-configurations. This is a sort of case-based configuration, where similarities to previously made configurations are exploited. Any modification in the requirements specification would not necessitate the configurator to re-run the whole process, instead it simply localize the change in a cluster or a sub-cluster with the help of the configurator-aided-clustering.

A first version of the clustering algorithm has been applied to the USG of DREAMS. It resulted in a hierarchy of 3 levels where the clusters per tier are ranging from 2 to 8.

You can observe that heuristic knowledge did play a role in clusterings for supporting the configurator. The first-level clusters that have been obtained from the clustering supporting the configurator are very similar to those clusters, which would have been built by the operating system experts for documentation purposes. The clusters naturally embody all services that belong to the same general operating system function family, like *board & devices*, *core* DREAMS *functions*, *memory management*, *communication*, *synchronisation & scheduling*, *exception handling* and *data structures* (see Figure 6.10).

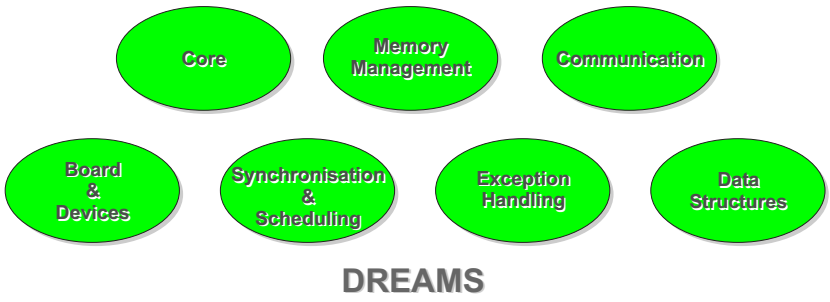

Fig. 6.10: Top-level clusters of DREAMS.

Another major impact of having hierarchical clusters in the process of configuration, when a reasonably complex system is under consideration, is the expected reduction in the configuration time. It paves a new way to re-configuration. Another advantage includes customising one cluster at a time instead of the whole flat graph, which not only reduces the required memory size, but also makes the configurator to manage bigger systems.

6.6 Description Languages

In this section we will describe the languages that are used in order to describe the general domain knowledge, the hardware and the topology, and the application specific domain knowledge (requirements specification). A detailed description of the languages is given in Appendix C. There, we will explain the semantical meaning of the statements.

The general domain knowledge describes the *Universal Resource Service Graph* (URSG), which is one input to the configurator TGEN. The software model (USG) and the hardware model (URG) are both integrated in the URSG. Another input for TGEN is the *Resource Graph* (RG) that defines the hardware and topology description, for which the

operating systems and the communication system should be generated. Last but not least, the application specific knowledge defines the requirements specification and is the *Process and Communication Graph* (PCG), which must also be an input to the configurator TGEN.

The grammars of the languages were designed to be context-free in order to build simple parsers (in contrast to context-sensitive grammars). Each description of an item of the databases, respectively graphs, is introduced by a type identifier keyword, like "`SERVICE`", which is followed by the name of that item in brackets (<`service-name`>) and several property statements. The description of an item of the databases is ended by the same type identifier keyword, which has the prefix "`END_`". A property statement is defined by a property identifier keyword, like "`VISIBILITY`", which is only followed by a parameter list in brackets. An example can be found in Figure 6.11.

```
SERVICE(<service-type-name>)
  VISIBILITY(<num>,<num>,<num>)
  POSITION(<x>,<y>)
  COSTS(<int>)
  REQUIRES(<cpu-type-name>)
  PRIORITY(<num>)
  PRIORITY2(<num>)
  MAXPRIORITY(<num>)
  ORDER(<num>)
  PREFIX("<string>")
  SUFFIX("<string>")
  PATH(<path-name>)
  FILE(<file-name>)
  [ IN(<sap-type-name>) [ FIXED_AT(<x>,<y>) ] ]*
  [ OUT(<srp-type-name>) [ FIXED_AT(<x>,<y>) ] ]*
  [ NEEDS(<hap-type-name>) [ FIXED_AT(<x>,<y>) ] ]*
  [ TRIGGEREDBY(<hrp-type-name>) [ FIXED_AT(<x>,<y>) ] ]*
  [ ASYN_CLIENT_OF(<service-type-name>) ]*
  [ SYNC_CLIENT_OF(<service-type-name>) ]*
  [ INHIBITS(<srp-type-name>,<sap-type-name>) ]*
  [ FAVOURS(<srp-type-name>,<sap-type-name>) ]*
  [ FORCES(<srp-type-name>,<sap-type-name>) ]*
  [ FORCEHWUSE(<srp-type-name>,<sap-type-name>) ]*
  [ INHIBITS_AT_MULTI_NEED(<srp-type-name>,<sap-type-name>) ]*
  [ SAY("<string>") ]*
  [ NEGSAY("<string>") ]*
END_SERVICE
```

Fig. 6.11: Description of a declaration for a *service* in the specification file for the USG.

For a better readability each property statement should be placed into a single line; but this is not a must. All of the property statements are optional. If one is missing, then a default value is assumed. Some property statements can be multiply defined with different parameter lists per database item.

It would have also been possible to define the input languages to be compliant with the XML (*eXtensible Markup Language*) specification. But XML would introduce some non-

negligible overhead, which makes the reading of the declarations difficult for humans. An advantage of using XML would have been that a lot of parsers for the development of the software tools could have been used and also other tools could read the languages.

General Domain Knowledge

The general domain knowledge for the configuration process is defined in a database, in which all configuration items are stored. It defines a kind of knowledge base, because all properties of the configuration items, like their dependencies, the alternative selections and the constraints, are described herein. The configurator TGEN reads this database from a text file. The definition of the database starts with the keyword `DATABASE` and ends with the keyword `END_DATABASE`. Every line before and after this keywords are ignored. Inside the database definition all `IPORTs`, `OPORTs`, `IOPORTs`, `SAPs`, `SRPs`, `HAPs`, `HRPs`, `PRIMITIVEs`, `SERVICEs`, `EXECUTION_BLOCKs`, `CPUs`, `DEVICEs`, and `MEDIAs` have to be declared. Thereby, exactly this order have to be used, because, for instance, before an SAP can be referenced in a *service*, it must have been declared.

The central definitions of the general domain knowledge are the declarations of *services*, *execution blocks*, and *user primitives* (see Section 6.3$_{108}$ and Section 6.8.1$_{135}$).

Hardware and Topology

In the hardware and topology description, which is the *Resource Graph* (RG), only two statements are allowed. This is the `HARDWARE` statement and the `TOPOLOGY` statement. They also have to be placed in the given order.

Inside the `HARDWARE` statement concrete instances of `CPUs`, `DEVICEs` and `MEDIAs` are created. Therefore, Each instance have an unique identifier as its name and is of a specific type, which must have been declared in the general domain knowledge of the *Universal Resource Service Graph* (URSG).

The `TOPOLOGY` statement mainly defines connections between the CPUs and the devices and between the devices and the media. For each connection a port type name must be given. A port of exactly this name must be present in both connected resources. That means that only those ports can be connected, which have the same name! Remark, that the direction of the connection is defined by the ports and not by this statement.

Requirements Specification

The requirements specification allows only five different statements. With these statements the tasks, their system call events, their communications, and their placement onto the CPUs can be described.

6.6.1 Describing the Design Space of DREAMS with the TEReCS Model

This section shows in which way very fine granular customisation of a run-time platform can be exploited. The configuration tool of TEReCS is applied to a customisable library-

based construction kit for embedded operating systems. We report on the use of the TEReCS configuration approach to the object-oriented and at source code level customisable application management system DREAMS (see page 51). We demonstrate that the universal configuration approach of TEReCS is applicable to object-oriented paradigms and that this leads to fine-grained and tunable systems that avoid overhead.

First, a brief introduction to the customisation facilities of DREAMS is given. Some of the internal concepts of DREAMS are presented. Second, some major aspects of how the configuration model of TEReCS is applied to DREAMS are considered.

6.6.1.1 Customisation Features of DREAMS

DREAMS is a construction kit for embedded real-time operating systems. It supports for memory management, device access, multi-threading (scheduling), resource allocation, mutual exclusion, synchronisation and communication for embedded real-time applications. Its advantages are the flexibility of customisation, the availability of its source code, the extensibility, and the hardware support of different target architectures. DREAMS is an object-oriented system written entirely in C++ (except some very few lines in assembler for context switching). Thus, the class structure of DREAMS is the basis for its customisation. Classes of DREAMS objects can be configured. Such configurable classes are named *skeletons*. For providing configuration facilities C++ was extended by a *Skeleton Customisation Language* (SCL) [42].

Customisation within DREAMS is applied during compile-time. Therefore, SCL is translated into pre-processor commands, which are finally handled in the compilation process. A customisation description in SCL and an appropriate makefile is required in order to generate an execution platform (kernel-like virtual machine) from all skeletons of DREAMS. The task of TEReCS' configuration tool TGEN is to support these files.

As mentioned before, customisation within DREAMS is applied at class level. Mainly, there are four class properties, which can be configured:

Inheritance: The base classes
Aggregation: The components which are part of a class identified with their type
Polymorphism: Choice of a specific method during run-time (virtual concept)
Construction: Allocation method for objects and when they are created

The first and second property are the most important ones. This means that they are used more often than the other ones. This is not surprising, because with these design decisions the main properties of an object get defined.

Now, we shall give a brief introduction to some meta customisation concepts in DREAMS. This basically helps to sketch the general possibilities of customisation in DREAMS, and to show the complexity of the design decisions and their results in a more or less conceptual way. Mainly, we present the two categories of customisable objects, which belong to different levels of abstraction in DREAMS: *Skeletons* and *Application Programming Interfaces* (APIs). Further, we introduce the concept of synchronisation for mutual exclusion. Finally, we give an idea of how overhead is avoided by customisation.

APIs and Skeletons. Customisation in DREAMS is not only applied at the class or object level, but also at the interface level. Not only concrete properties of an object are configurable, but also the access interface is adaptable. The interface level of the inheritance hierarchy (class), on which an object is accessed or used by others, can be changed. This is done by declaring stub-classes, named APIs. For example, if class A is derived from class B, then the API of an instance of class A may be the interface of A or B. If B is chosen, A may have virtual functions, which need not to be accessed. The API of class A defines the access interface to concrete instances of the object.

Resource Protection. If an object is shared by multiple processes, the access to the object normally must be protected using synchronisation by locking. This is done in DREAMS by deriving an object from the base class `Resource` instead of the base class `Object`. It is left to the configuration to determine if an object is a `Resource` or a simple `Object`, which must be protected or not.

Avoiding overhead. Avoiding overhead means normally to avoid each source code line that produces some "never used" code, i.e. code which is useless and code that has no effect but occupies valuable resources. In an object-oriented specification paradigm this means to eliminate classes from the class hierarchy, and also to eliminate aggregated members which are not demanded. Replacing a simpler function by a more complex one often means to exchange the base class, which owns this function. These concepts request for an extension to the object-oriented paradigm that allows to specify inheritance and aggregation at a meta-level. Exactly this and a few more things are done by the SCL.

The classes of DREAMS can be divided into eleven families: classes that represent special operating system levels, classes that reflect the hardware architecture of the board, processor and its devices, classes for memory management, object representation, database management, device drivers, process management, exception handling, scheduling, multiplexing, synchronisation and communication.

The class diagrams for each of these groups in DREAMS were extended by additional information about the configuration options of these classes. That means that different alternatives for inheritance and aggregation etc. are modelled by the SCL. This was necessary to model the complete design knowledge and to get the overall view of DREAMS. Valid alternatives were distinguished from invalid ones by examining various customisation examples of predefined execution platforms constructed from DREAMS (and by a lot of discussions with the DREAMS developer Carsten Ditze). Invalid configuration options had been eliminated by the definition of constraints between mutual dependent OR-groups. This led to a description of the valid design space of DREAMS.

6.6.1.2 From DREAMS' Skeletons to TEReCS' Service Dependencies

The procedure of applying the TEReCS approach to the DREAMS system can be divided into the following steps:

1. Investigate configuration options of DREAMS.

2. Analyse the DREAMS class structure.
3. Examine predefined customisation examples of DREAMS for special run-time platforms.
4. Search for customisation rules.
5. Define transformations from DREAMS' customisation facilities into the TEReCS model.
6. Apply transfer rules to all DREAMS classes considering all valid customisation options.

We have already had a look at the DREAMS system and the TEReCS concepts. Now, we want to present the main ideas of integrating DREAMS into TEReCS. In order to describe the customisation facilities of DREAMS with the TEReCS model a set of transfer rules (from one model to the other) have to be developed. The basic rules are the following:

- Each *skeleton* and API in DREAMS is viewed as a *service*.
- For each functionality of DREAMS a "virtual service" is being introduced.
- The inheritance relation of a *skeleton* or an API is modelled by an SRP that references all possible SAPs of valid base classes.
- For each configurable aggregated member of a *skeleton* an SRP is made available. This SRP contains all possible SAPs of valid types for this member. This includes eventually an SAP of a particular empty service indicating that the aggregated member is not required.
- The SAPs being used in inheritance SRPs and the SAPs being used in aggregation SRPs are different ones.
- The SAP used for an aggregation relation additionally determines the construction method and its polymorphism.

The major transfer rules have been illustrated in Figure 6.12 on the following page. They show in what way the inheritance and aggregation dependencies are modelled in the deterministic as well as in the non-deterministic case. The non-deterministic case is exactly the case, where the alternatives for the customisation are described within the SCL.

Now, let us take a detailed look at some concrete examples that are taken from the extended DREAMS class hierarchy.

Example 1

In DREAMS the class `Task` can be derived from the class `Thread` or from the class `MultiThread`. The class `MultiThread` is required, when multiple threads are created for the task. For this reason, the access interface `API_Thread` for the thread structure in DREAMS can either be of type `Thread` or of type `MultiThread`. Multiple threads can only be created, if the task calls the primitive `CreateThread()`. These dependencies are modelled and shown in Figure 6.13$_{131}$. The configuration procedure works as follows on this model: The SAP of service `Thread` is limited by one entry. Therefore, either `API_Thread` or `MultiThread` can use the service `Thread`. Normally, the service `Thread` is chosen, because of its lower costs (`MultiThread` inherits costs from `Thread` plus its own costs). But, if `MultiThread` is also used by the virtual service `MultiThreading` (requested by the use of the primitive `CreateThread()`), then `MultiThread` must be chosen, because of more requests. For a detailed description of the selection mechanism of alternatives please refer to Section 6.4$_{112}$.

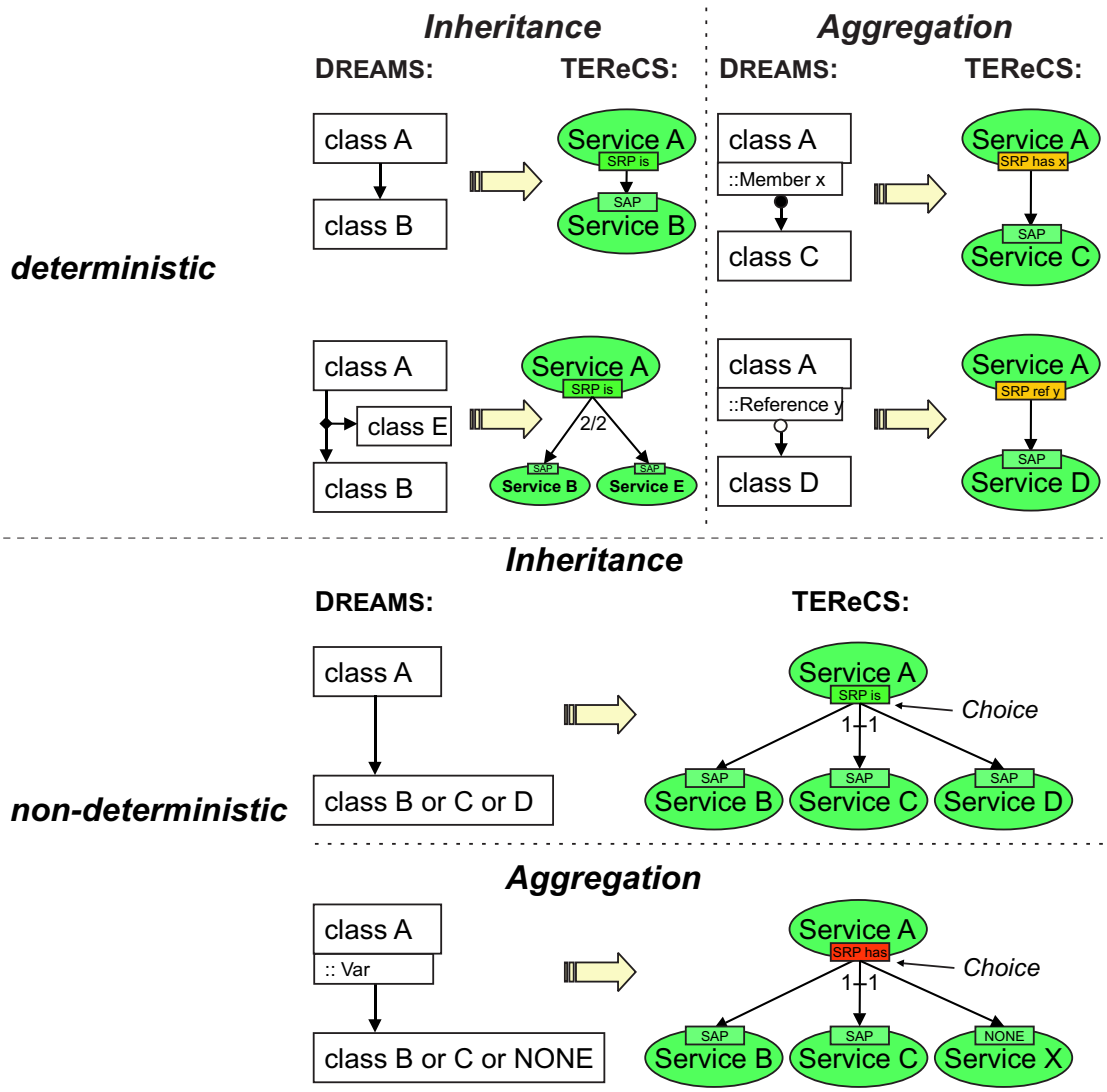

Fig. 6.12: Illustration of the major transfer rules for the modelling of DREAMS' object-oriented customisation features within the TEReCS' model.

Example 2

The class `Processor` in DREAMS has an optional timer component. This represents the concrete device driver for accessing the hardware timer of the processor. Depending on which processor is used, the correct driver must be set in an appropriate manner. This includes, that the timer can be unused and, therefore, the driver can be excluded in order to save space for the representation of the class `Processor`. The model for these service dependencies is given in Figure 6.14 on the next page. In the normal case the component is chosen to be "empty". But if the service `TimerInterrupt` is also selected, it prohibits the aggregation SRP of the class `Processor` to choose "empty". Two possibilities exist for using a service as the device driver for the timer: Either the class `PowerPCTimerDevice` or the class `CToolsTimerDevice`. The selection will be made according to the present hardware. The service `PowerPCTimerDevice` can only be chosen, if the processor is a `PowerPC`. Likewise, service `CToolsTimerDevice` can only be chosen, if the processor is a `Transputer`. Thus, the choice is unique.

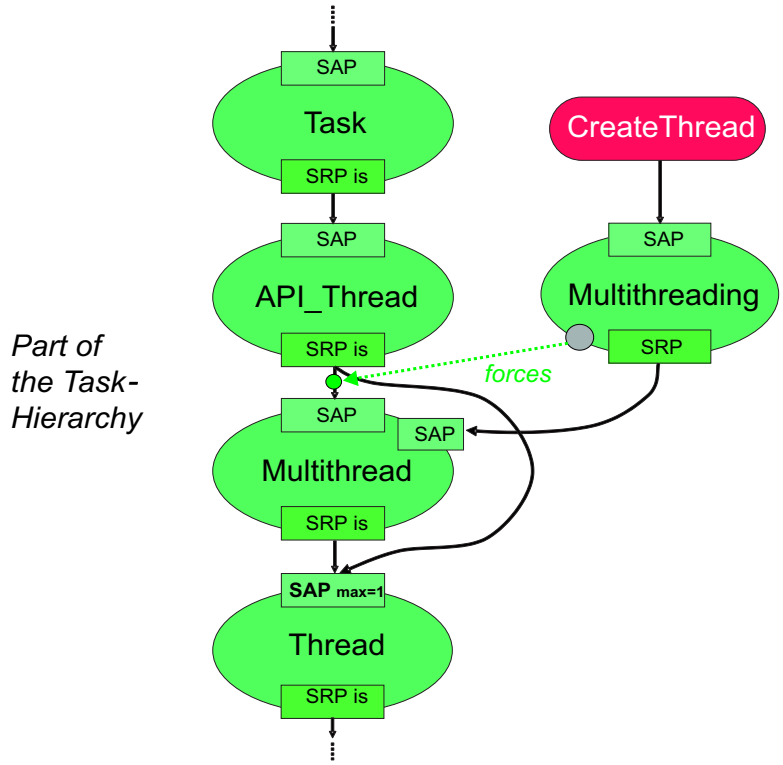

Fig. 6.13: TEReCS model for the class hierarchy of DREAMS' task hierarchy.

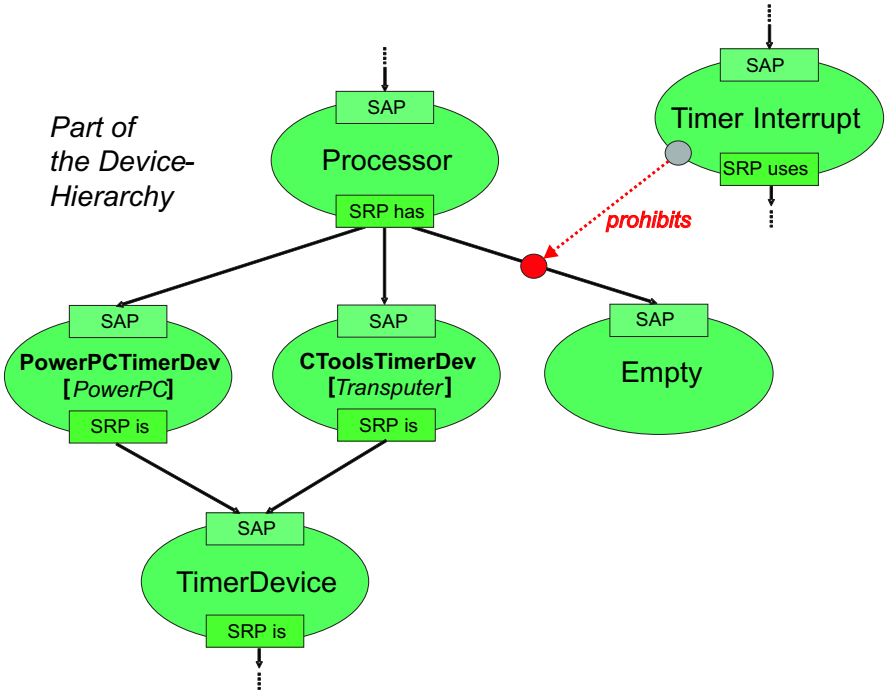

Fig. 6.14: TEReCS model for the integration of timers into DREAMS.

These two examples illustrate major underlying concepts of coding the configuration options of DREAMS into the TEReCS' model.

6.7 Knowledge Transfer from the Application to the Configurator

This section tries to explain briefly in which way knowledge about the application can be transferred to the configurator and in which way it can there be exploited for the operating and communication system's configuration. The requirements specification (as described in Section 6.6$_{124}$) defines the behaviour of the application. When this behaviour is known a priori, then the operating and communication system can be implemented appropriately. This means that specific services are implemented in such a way, so that they cope only with the required use case. They do not produce any overhead, because they need not to handle excluded use cases of the application. The code of a service can often be simplified, if assumed that special restrictions hold. These restrictions are defined by the application programmer and are often directly intended. The restrictions define a special behaviour of the application or they define that the application must behave in a special way.

An example is a specific scheduling scheme, which depends on the task set's properties. Another example is the synchronisation of a resource access. The synchronisation must only be implemented, if the resource is really shared simultaneously between different processes. If the resource is not shared or the resource is never accessed during the same time slot (implicit synchronisation), then the explicit synchronisation by using a semaphore, for example, is not required. Additionally, if a resource is accessed only by two processes, then the synchronisation code can be simpler, as if it would be, when it is accessed by an unknown number of processes.

Specific knowledge about the application is used during the configuration of the operating systems in order to simplify the implementations of certain operating and communication system services. Because of this additional knowledge about the behaviour of the application, the operating and communication system functionality can be adapted to the special requirements of the application. Or, expressed from another point of view: The operating system specification is enriched by specifications about the application. Thus, this additional knowledge can be exploited for optimising the implementations of operating and communication system services. This process of integrating additional knowledge into the operating and communication system's specification is a knowledge transfer from the application's domain into the operating and communication system's domain.

The configurator is responsible to integrate this additional knowledge correctly. In TEReCS the functional correctness of the generated operating systems is assured manually. The operating system designer, who integrates new services into the knowledge base for operating system services (USG), must assure that these services correspond to the other services and that they will only be selected, if the correct use case, for which the service had been developed, is applied. This can be done by the use of constraints. Often, *virtual services*, that represent those use cases, are defined and must explicitly be activated by the use of primitives in the requirements specification. The activation of those "virtual" services control the correct selection of the other services by constraints. However, these constraints must be correctly defined by the operating system engineer.

In this way the knowledge about special use cases is integrated into the operating and communication system: The application demands for specific primitives that request for (virtual) services that again activate certain constraints, which control the correct selec-

tion of services from their alternatives. This is exactly the idea of *puppet configuration* (see Section 5.4$_{100}$).

Besides information about the application's behaviour also information about the expected temporal correct execution of the application is provided to the configuration process. This information comprises worst-case execution times of the processes, their expected maximum finishing and response times (*deadlines*) and the relative event times, when a process calls special operating system services. These relative event times are measured from the start of the process. During the configuration the implementation of those operating system calls (represented by the use of primitives) is determined. The overhead for each implementation of an operating system service is known, because it had been also specified by the operating system engineer while he integrated that service into the USG. To determine the overheads is not part of this thesis. Already existing tools, like CHaRy [1], can do that. These already known overheads are forwarded together with the timing information of the application to a real-time analysis of the application's behaviour in co-operation with the operating and communication system's behaviour (see Section 6.8).

When the timing analysis detects failures of the timeliness behaviour (deadline misses), then mainly the configuration of some operating systems or the configuration of the communication system must change. This is mainly done by selecting other communication paths (change in routing) or other task mappings onto processors. In this way, the creation of a functional correct configuration and the timing analysis is executed alternately unless both succeed or the configuration does not find any more valid configuration.

The process of the timing analysis is described in the next Section 6.8. Section 6.9$_{142}$ and Section 6.10$_{147}$ explain, in which way the configuration and the timing analysis interact.

6.8 Real-Time Analysis

For real-time analysis mainly two approaches exist. The first approach is the workload analysis, in which the cumulative workload of the processors, devices, and communication links must not exceed 100% or even a lower limit. The second approach is the response time analysis, in which the overall sum of all delays of an action (including its execution) between the arrival of a request and the time to finish servicing it must not be greater than the deadline. In this case all the possible orders of the delays (runs of a schedule) must be considered. Therefore, an analysis has to take care of all possible interruptions and disturbances in an execution path. For complex systems, and even more for heterogeneous systems, this becomes very hard to solve. For example, in a communication system, which consists of several *different* communication links, the transmission and queuing delays differ. Moreover, if the protocols are different on the paths, then the blocking delays have to be calculated appropriately. Thus, the analysis has to cope with a lot of different formulas according to the path of the message.

Other problems for the analysis of distributed communicating processes are the mutual influences of the process scheduling and the communication delays. The process scheduling determines the relative and absolute times of sending and receiving events. This forms the basis of the calculation of the end-to-end communication delay. But the communication delays have again impact on the blocking delays of the process schedules.

Thus we have a cyclic dependency. Real-time analysis often assumes that this will result into a stable state, where the mutual influences are broken. This means that a slightly different process schedule does not change the blocking delays or vice versa. Often, real-time analysis assumes that the blocking times are independent of the schedule and, therefore, they can be determined a priori. This results into worst-case timings for the blocking delays, which are often too pessimistic (see PIP, PCP, SRP). In real-time analysis also the precedence constraints between tasks are ignored or require a complex analysis scheme (see EDF* or Spring). But in this thesis precedence constraints due to communications are assumed to be absolutely indispensable for distributed embedded systems.

For this reason, in TEReCS another approach is followed. Assuming, that the execution delays of all atomic actions can be calculated very precisely, and their order can also be determined precisely, then an execution model of the final application can be created. From this execution model all deadline misses can be observed. Additionally, if a deadline miss occurs, then the actions responsible for delays can be identified. Consequently, if we have an execution model of all atomic actions, then their execution can be visualised in form of a Gantt diagram. This can give a deeper insight into the application for the engineers. They can realise, what happens when on the target system. This approach is similar to a simulation of the application.

In contrast to a simulation the code will not be executed or analysed. Control structures of the code, like condition statements or loops, will not be considered. *Execution blocks* are defined instead. An execution block represents the execution time of a code sequence. All loops in the code are assumed to be unrolled. For a condition statement only one execution path is considered.

In order to achieve the schedule of the final execution, certain execution blocks must be arranged in the form of a schedule. This means that the execution blocks are placed chronologically onto the time line. An execution block can be interrupted by an *event*. To each event a certain sequence of other execution blocks is assigned. The occurrence of an event means that the assigned sequence of execution blocks must be planned immediately. Thus, an execution block can be divided into interrupted sub-blocks, which we call *runs*.

A process (task or thread) of the system is represented by one execution block. Each call of a primitive (system call) of the operating system or a real interrupt of a processor will be interpreted as an event. Therefore, each execution block contains a time-ordered list of events. These events have to be planned relatively to the starting time of the execution block. Each execution block can again invoke other events at its end. In this way a *Time-triggered Event Scheduling* of the execution blocks is realised. This is similar to the TTP approach of Kopetz (see Section 4.4.2.3$_{84}$). The main difference is that the realisation (see Section 6.8.4$_{139}$) is not seen as strict as the TTP does.

In order to manage a distributed system consisting of parallel executing processors, for each processor (CPU) one time line exists. Thus, for each processor a schedule of runs will be planned. For controlling the event flow, like operating systems do, an event will be inserted into a time-ordered *global event list*, when it is invoked. But the occurrence of the event can be delayed into the future. Additionally, execution blocks can wait for certain events to be present in that event list. An execution block, which waits for events, can only be planned for a schedule, when the events are already in the event list and the time, when the events will occur, are reached. The event can only be taken (planned/executed),

when it is enabled and when there is an execution block, which waits for that event, in order to be placed onto the schedule. For this reason the subsequent planning of execution blocks can be interrupted, when events are missing. Such waiting execution blocks are inserted into a *waiting list* per processor. Besides the waiting list a *ready list* exists. Execution blocks that have to be planned are inserted into the ready list or waiting list of each processor accordingly.

The complete timing analysis is implemented by the tool TANA of TEReCS. In the following sections the execution model and the planning, respectively scheduling algorithm of the execution engine will be presented in detail.

6.8.1 Scheduling Model

First, we will define some basic items, which are used for the simple scheduling model.

Resource

Normally, each `Resource` contains a waiting queue, whose items (processes that wants to access that resource) are ordered by a priority and by a FIFO ordering. For this scheduling engine the local waiting queues of all the `Resources` are integrated into one single global waiting list (WL).

A `Resource` defines several `Events` that can be executed on this resource. For example, this can be `up()` or `down()` for the corresponding semaphore.

Whenever a `Resource` is defined, we also define the number of units available. This shall be referred to as the amount of resources available. These units can be produced and consumed by `signals()` and `waits()`. A `Resource` can be disabled, which means that no units can be consumed thereafter.

All `Resources` must be placed onto a CPU or onto GLOBAL. A `Resource` can be placed simultaneously on different CPUs and on GLOBAL. GLOBAL is an implicitely defined "virtual" CPU.

The following items of an execution model are derived from `Resource`:

- CPU – is representing a processor node of the network
- Task – is representing a process or thread of the system
- Semaphore – used for synchronisation of tasks
- ComLine – used for communication between tasks
- Timer – used for time management of tasks, like sleeping

Thereby, the scheduler of an operating system and also interrupts have to be modelled as tasks.

Event

An `Event` can be called asynchronously by a `Task`. Each `Event` defines a sequence of `Execution Blocks` that must be planned subsequently in the defined order when the `Event` occurs. Thus, an `Event` delays the execution of a `task`.

An Event is always part of a Task and is always executed on a Resource. The Event normally represents the execution of operating system services.

Execution Block

An Execution Block is an instance, which can be planned for a schedule. Nevertheless, an Execution Block can be preempted by another ready Execution Block with higher priority. Execution Blocks can define their own priority or inherit it from the Task that is calling the Event to which the Execution Block belongs.

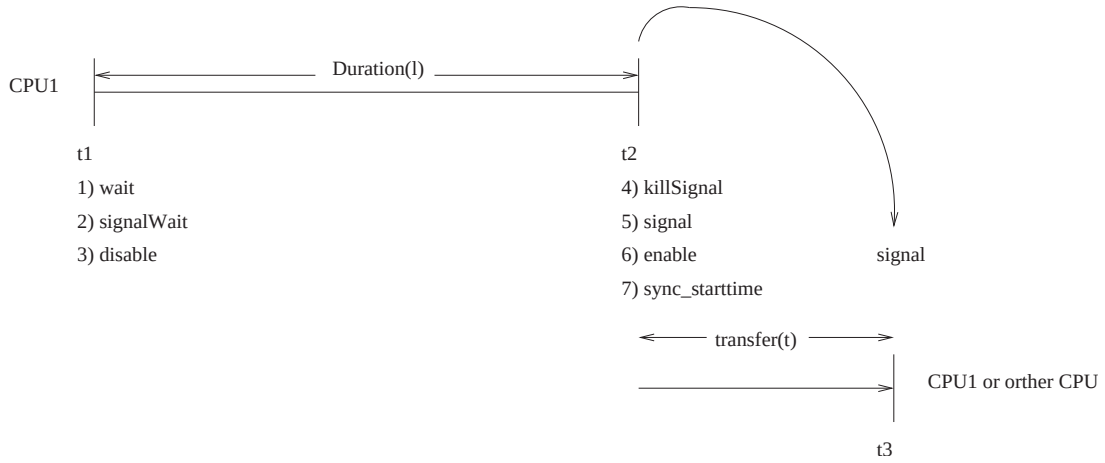

Fig. 6.15: Illustration of timings and execution points of the signal handling for an execution block

At the end of each Execution Block several signals() can be emitted. This means that they are inserted into a global pending signals queue. These signals have to be activated immediately, or in the future. If a signal() is activated, then the items of the assigned Resource are increased by the specified amount.

At the beginning of each Execution Block several waits() can be defined to be considered. A wait() consumes the specified amount of items of a Resource, which is assigned to that wait(). But items can only be consumed, if enough items are present (see Section 6.8.2 on the next page).

Additionally, at the beginning of each Execution Block some Resources can be disabled. At the end of an Execution Block some Resources can again be enabled.

Execution Blocks can be preempted by other Execution Blocks with higher priority because of the occurrence of Events or signals(). Therefore, Execution Blocks are divided into Runs. An Execution Block is always part of an Event or Task.

Run

A Run is the smallest instance that will be planned and mapped into a schedule. It is always a part of an Execution Block.

The duration and, therefore, its ending time is explicitly not known, but must be calculated by the scheduling engine (see Section 6.8.2 on the facing page).

A Run is terminated if its Execution Block ends or if another Execution Block with higher priority can consume all of its Waits and, therefore, preempts the actual Run. The preemption can only occur when a signal is emitted.

Task

A Task is derived from a Resource. By this the Task can wait for itself. This is required in order to simulate a sleep by waiting for itself.

The complete run-time of a Task is defined as one master Execution Block. The complete master Execution Block of the task can be split into several Runs. In-between the Runs the Events of the Task occur. Before and after each Run of the Task only one Event of this Task is allowed to occur at a time. That means that there must not exist two Events of the same Task, which have to occur at the same time.

All Tasks are modelled as periodic tasks. At the end of their Execution Block, their execution starts all over from the beginning. This helps in scheduling periodic tasks or tasks that loop forever. If a Task wants to terminate, then it has to wait forever for a signal.

6.8.2 Time-triggered Event Scheduling Engine

All ready Execution Blocks are maintained by the ready queue (RQ). If an Execution Block has to be executed, then this Execution Block is marked *ready* and is inserted into the ready queue of the appropriate CPU.

Before an Execution Block can be executed, all its waits() must match the available units of the Resources, which are assigned to that wait(). All waits() must be fulfilled. That means that *all* units of the Resources of *all* waits() for the Execution Block are consumed or *none* of them. If they are consumed, then the Execution Block is marked ready. If they cannot be consumed, then the Execution Block is marked *waiting* and inserted into the waiting queue. Additionally, the Task, to which the Execution Block belongs, is marked *blocked*. A wait() matches, if and only if, there are enough free units in the referred Resources *and* if the Resources are enabled.

If an Execution Block is completely executed, then all signals() defined for this Execution Block must be emitted. That means that they are inserted into the global queue for pending signals (PS). If a signal() from this PS list has to be activated, then all waiting Execution Blocks of the waiting list WL (and blocked Tasks), which are waiting for this signal() and that can receive enough units now, will be set to *ready*.

If there exists no active Event of a Task, then the master Execution Block of the Task has to be executed until the next Event occurs or the task terminates.

When an Event occurs, the Event is marked *active*. Then the next Execution Block of this Event has to be executed and, therefore, that Execution Block must also be marked *ready*. If an Event is marked *active* and none of its Execution Blocks can be marked *ready*, then the Event is finished and deactivated.

An Execution Block is executed until it either terminates or the occurrence of a signal() of the PS queue. If the Execution Block is the master Execution Block of a Task, then it possibly has to be executed until the occurrence of the next Event of

that `Task`. The execution is performed by planning a `Run` of that `Execution Block` into the final schedule.

After a `Run` is planned into the schedule, the next ready `Task` with highest priority is selected from the ready queue. If this `Task` has an active `Event`, then the next `Run` of the ready `Execution Block` of that `Event` will be executed. If there exists no active `Event` of the `Task`, then the master `Execution Block` of the `Task` will be executed. To execute means here that a new `Run` of the `Execution Block` is planned until the end of the `Execution Block` or until the occurrence of the next `signal()` in the PS queue, or the next `Event` of that `Task`. If no `Run` can be executed, then no output will be given. There exists implicitly no "idle" task.

6.8.3 Example for a Time-triggered Event Schedule

In Figure 6.16 an example for the output of a *time-triggered event scheduling* is given. Three schedules are illustrated: Two schedules for two processor nodes (CPUs) and one schedule for the communication link between these two CPUs. The two tasks *task 1* and *task 2* are modelled. *Task 1* runs on CPU 1 and *task 2* runs on CPU 2. Both tasks have a period of 340 time units. *Task 1* has an execution time of 80 time units and a priority of 1. *Task 2* has an execution time of 90 time units and also a priority of 1. The formal specification of this example, which is written in TEReCS's requirements language, can be found in Appendix B.

Because the scheduling model assumes that a task starts over its execution immediately after the task's execution block terminates, both tasks execute a `start()` event at their beginning (relative starting time 0) and an `end()` event at their end. The event `start()` executes the execution block *start* and the event `end()` executes the execution block *end*. Whereas *start* waits for a signal on the task's resource, *end* emits this signal at its end for x time units delay into the future. Here, x is the result of the task's period minus the already executed time of the task since its last activation. Thus, both tasks will be activated really periodically within their defined period.

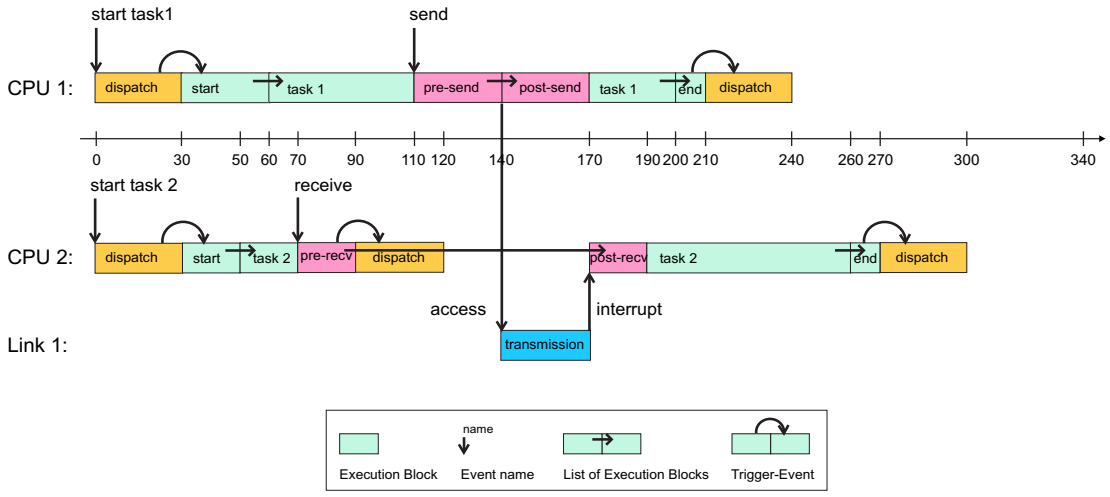

Fig. 6.16: Example for a time-triggered scheduling.

In order to model a communication the resource *Comline1* is defined. *Comline1* declares two events, which can be executed: `send()` and `receive()`. The event `send()` executes sequentially the two execution blocks *pre-send* and *post-send*. The event `receive()` executes sequentially the two execution blocks *pre-recv* and *post-recv*. The execution block *pre-send* emits at its end one signal on the resource *Comline1*. The execution block *post-recv* awaits this signal at its beginning. The signal will be delayed on the resource *Comline1* by 30 time units. This delay represents the media access time plus the transmission time plus the response time of the receiving CPU.

Additionally, in this example it is specified that *task 1* executes the `send()` event relatively after 50 time units. Also *task 2* executes the `receive()` event after 20 time units. With this we have defined a communication between *task 1* and *task 2*.

In order to model a Round-Robin scheduler of an operating system for both CPUs a second task *scheduler* with the highest priority 0 and a period of 500 time units is defined. It specifies the execution block *dispatch*, which waits at the beginning for one signal on its own resource and emits this signal at its end in order to generate its periodical execution. The duration of the execution block *dispatch* is 30 time units.

When the execution block *post-recv* has to be scheduled, but when no signal on the resource *ComLine1* is present, then *post-recv* has to wait and the execution block and its parent task will be blocked. Additionally, a signal without any delay will be sent to the task *scheduler* of the same CPU. Therefore, we have defined a so-called `wait-and-signal()`, which emits a signal, if and only if, the wait will result into a blocking. By this way, the task *scheduler* can proceed immediately. The pending signal for the *scheduler*, which was sent by itself for its periodic activation, will be explicitly be consumed by the execution block *dispatch*. With these definitions we attain that directly after the execution block *pre-recv*, the execution block *dispatch* is scheduled. This is because the *post-recv* is too early (the *pre-send* was not already done) and, therefore, *post-recv* must be blocked.

6.8.4 Schedulability Analysis

If all the durations of the execution blocks, the relative occurrences of the events, and the delays of the signals, as well as the model of all tasks and their dependencies correspond to the reality, then the schedules will represent the original execution on the CPUs and links. But when only one timing is wrong, the complete schedule can be different (see Section 6.10_{147}). Thus, the finishing time of a task can be extremely delayed or shortened.

Taking the worst-case timings does not solve the problem, because the finishing time of a task is not calculated upon worst-case assumptions. Under worst-case assumptions, like traditional real-time analysis does, the finishing time is calculated by considering *all* possible worst-case delays. Here, only the worst-case timing of one, or a limited set of specific delays, is considered.

Nevertheless, in order to achieve good results for the timing analysis in TEReCS the model is extended in the following way: Instead of one time for a duration, an event occurrence, or a signal delay, two times can be specified. These two times define an interval, in which the real event can happen. For this reason, the time-triggered scheduling must be calculated for all times of this interval (because for each time the schedule can possibly change). Moreover, if two intervals are given, then the schedules for all pairs of

all time combinations must be calculated.

A schedule is considered to be feasible, if for all tasks, their finishing time is equal or lower than their deadline. This is the classical definition. A time interval is valid, if the schedules for all times of the interval are feasible. If more intervals are given, then all schedules for all combinations of times from these intervals must be feasible. This results into the calculation of more than 2^n schedules for n given intervals. (In fact, it would be $\Pi_{i=1}^{n} m_i$ schedules, where m_i is the number of times that have to be considered per interval i.) This would lead to an exponential run-time for the real-time validation.

To minimize the run-time for the timing validation, in TEReCS not all schedules will be considered. Only the schedules for the minimum and maximum times of the interval (the two specified limits) will be considered. The schedulability test will be done upon an heuristic, which estimates, whether another time in-between the interval can possibly change the schedule or not. In this way, the run-time is still exponential, but the number of considered schedules is really reduced to 2^n schedules. If the heuristic schedulability analysis with time intervals is used, then its result can be wrong. For this reason, the time-triggered scheduling, which is used in TEReCS for the real-time validation, is – for the first view – only an estimation for the timing validation, but it is not a proof for its correctness.

Nevertheless, it is a very practical approach. If the heuristic detects always a deadline miss, if and only if, such a deadline miss exists, then TEReCS' timing analysis works correctly (see Section 6.8.4.1). So, the value of TEReCS's timing validation depends on the quality of this heuristic function. In practice it could be shown that the defined function achieves good results. But the real advantage of this approach is that the engineer, who plans the system, can see the bottlenecks and timeliness arrangement of all modelled events of the system. In this way he can gain additional information about the effects of his specifications onto the scheduling.

But TEReCS offers another advantage. The configurator of TEReCS can assure that the real execution will be exactly that of the time-triggered scheduling. This will be achieved by integrating *guards* into the code for each event (system call). The *guard* is additional code, which assures that the event is taken at its expected absolute time. Eventually, the guard will delay the execution until the specified time is reached. Otherwise, if the estimated and expected time of the event was missed, the guard will complain and a fall-back function can be activated. Thus, possible deadline misses can be detected much earlier.

The execution time for the additional code of the guard, which is produced in order to verify the event times, is considered during the time-triggered scheduling. But in TEReCS the guard for an event must explicitly be activated in the requirement specification. It is the engineer's task to select appropriate events that should be guarded.

6.8.4.1 Heuristic Selection of Schedules

As already mentioned, whenever an interval is specified for the appearance of an event or a delay, then all schedules have to be created. Each schedule implements the use of one set of discrete times of the intervals. The purpose of the heuristic function is to select one discrete time of the interval, for which the schedule has to be checked. Hereby, it is

essential to select a candidate, if possible, for which the schedule is not feasible. Then, a negative answer can immediately be issued and no further investigations are required. All schedules for an interval must be feasible, in order to allow that interval.

Before we describe the heuristic selection of schedules that have to be tested for feasibility, we define the *direct* and *indirect* influence of a time for a message transfer. If the time belongs to any execution block that is executed by a sending task and that is executed before the real send of the message, then the change of this time can immediately change the transfer time of the message, which is issued by that task. Then, this time *directly* influences that message transfer of that task. The message transfer and the time always belong to the same sending task. If they belong to different tasks and additionally, the change of the time changes the message transfer time, then the time *indirectly* influences that message transfer.

The heuristic function will make use of the following observations:

1. The end time of an interval will produce the maximum delay for the executing task and the directly influenced receiving task.
2. The start time will produce the minimum delay for the executing task and the directly influenced receiving task.
3. Any time of the interval can have impact on the delay of another task. Whether this delay will be extended or reduced for an earlier or later time, cannot be answered.
4. If for any two times the order of the messages of two communications on a media changes, then any time in-between these times will cause an interference between those two communications.
5. For two times the order of a message between a directly influenced message and an indirectly influenced message can only change once. Here, during an hyperperiod corresponding message transfers have to be considered.

From these observations we can derive the following: If for the schedules of the start and end time of an interval an interference between a directly and indirectly influenced message transfer exists, then the response time of the indirectly influenced message transfer have to be increased by the transfer time of the directly influenced message transfer. If the finishing time of the receiving task, which receives that indirectly influenced message, is still equal or lower than its deadline, then all of the schedules for all times of the interval are regarded to be feasible. Otherwise a negative answer can be issued.

By this selection, only two times (the start and the end time) of a given interval must be considered. Of course, the run-time will be still exponential for the combinations of all start and end times. But, (1) only two times per interval are considered, and (2) this is only done for a positive answer.

Two rules help to prevent a lot of overhead:

1. Where possible, intervals should not be specified $\Rightarrow$ less intervals reduce the amount of start and end time combinations.
2. An interval should be specified as small as possible $\Rightarrow$ the probability of interferences is less.

These rules should be followed, when a requirement specification is created.

An advantage of this approach, which allows the definition of timing intervals, is that between the schedules jitters can be observed.

6.9 Impact of the Configuration on the Timing Analysis

In this section we will demonstrate that the results of the estimated time-triggered scheduling are often more accurate than those of a traditional worst-case analysis. Additionally, we identify the influences of the configuration on the timing analysis by inspecting a few examples.

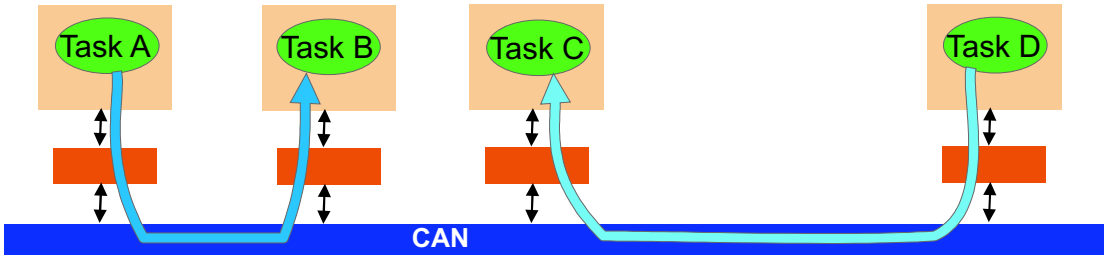

Fig. 6.17: Hardware architecture and task mapping, where four tasks on four processor nodes communicate via one single CAN bus.

We assume the following application scenario: The hardware of the target system is assembled of four homogeneous processor nodes, which are connected via four devices to one single CAN (Controller Area Network) [67] bus (see Figure 6.17). One general feature of a CAN bus is that the maximum size of a packet, which is sent over the bus, cannot exceed eight bytes. Therefore, messages often have to be split into several packets. On each of the four processor nodes one task runs. Task A sends a message to task B. This message must be divided into four packets of maximum size. Task D sends a message to task C by transmitting three packets. The run-time of task A without any operating system overhead takes 32 time units. The run-time of task B takes 16 time units, that of task C takes also 16 time units, and that of task D takes 24 time units. The operating system call for sending a message (send event) is assumed to occur 13 time units after task A has started its execution. For task D this send event is assumed to be after 17 time units. The receive events for the tasks B and C are both assumed to be after 8 time units. The requirement specification looks like the following:

$$
\begin{aligned}
\text{Task A} &: C_A = 32, \quad E_{send}(A) = 13 \\
\text{Task B} &: C_B = 16, \quad E_{recv}(B) = 8 \\
\text{Task C} &: C_C = 16, \quad E_{recv}(C) = 8 \\
\text{Task D} &: C_D = 24, \quad E_{send}(D) = 17 \\
\text{Communication 1} &: \text{Task } A \longrightarrow \text{Task } B \text{ with } n_{A,B} = 4 \text{ packets} \\
\text{Communication 2} &: \text{Task } D \longrightarrow \text{Task } C \text{ with } n_{D,C} = 3 \text{ packets}
\end{aligned}
$$

During the configuration of the operating systems for all four processor nodes the configurator determines the overhead of the operating system for the sending and receiving

of these messages. The execution time for the system call *send* (send primitive) has to be split into four phases. During the first phase the message is split and prepared for the transmission. This phase has a duration of 7 time units. After this, each of the packets is sent in the second phase by a device driver call, which takes 2 time units. After the complete message is sent an interrupt of the device driver, which lasts for 5 time units, informs the operating system in the third phase to proceed the sending task. In the fourth phase the operating system call needs for this 8 time units before the sending task is reactivated. Thus, the complete operating system overhead for sending n packets takes $7 + (n * 2) + 5 + 8$ time units. The system call for receiving of a message (receiving primitive) is similarly split into such four phases. The complete operating system overhead for receiving n packets takes $13 + (n * 2) + 5 + 8$ time units. The configurator gives the following operating system overheads, which had been assigned by the operating system expert to the appropriate services:

$$
\begin{aligned}
Ovh_{Sys}[\text{pre-send}] &= 7 \\
Ovh_{Sys}[\text{dd-send}](n) &= 2n \\
Ovh_{Sys}[\text{irq-send}] &= 5 \\
Ovh_{Sys}[\text{post-send}] &= 8 \\
Ovh_{Sys}[\text{pre-recv}] &= 13 \\
Ovh_{Sys}[\text{dd-recv}](n) &= 2n \\
Ovh_{Sys}[\text{irq-recv}] &= 5 \\
Ovh_{Sys}[\text{post-recv}] &= 8
\end{aligned}
$$

$$
\begin{aligned}
Ovh_{Sys}[\text{send}](n) &= Ovh_{Sys}[\text{pre-send}] \\
&\quad + Ovh_{Sys}[\text{dd-send}](n) \\
&\quad + Ovh_{Sys}[\text{irq-send}] \\
&\quad + Ovh_{Sys}[\text{post-send}] \\
Ovh_{Sys}[\text{recv}](n) &= Ovh_{Sys}[\text{pre-recv}] \\
&\quad + Ovh_{Sys}[\text{dd-recv}](n) \\
&\quad + Ovh_{Sys}[\text{irq-recv}] \\
&\quad + Ovh_{Sys}[\text{post-recv}]
\end{aligned}
$$

The physical transmission time φ_{packet} for a packet of 8 bytes on the CAN bus is calculated upon the bit rate and takes 6 time units. The overall transmission delay φ^{send}_{trans} for n packets for a sending task lasts for $(n-1) * 2\,\varphi_{packet}$ time units. The sending task has to wait only for $(n-1)$ packets to be transmitted. This is defined by the final configuration architecture of the operating system's implementation. For the receiving task the overall transmission delay φ^{recv}_{trans} for n packets takes $n * 2\,\varphi_{packet}$ time units, because the driver has to wait for the reception of all packets. The transmission delays are worst-case values, because the transmission delay for each packet is assumed to be twice of the physical transmission delay. This is, because potentially the transmission of a packet must be delayed by the media access, while actually another packet is on the bus.

$$
\begin{aligned}
\varphi_{packet} &= 6 \\
\varphi^{send}_{trans}(n) &= (n-1) * 2\,\varphi_{packet} \\
\varphi^{recv}_{trans}(n) &= n * 2\,\varphi_{packet}
\end{aligned}
$$

A traditional worst-case analysis normally does not consider the times of the appearances of the system calls. Simply the worst-case delays and execution times are summed. For the classical worst-case approach we obtain the following results:

$$\begin{aligned} R_A &= C_A + Ovh_{Sys}[\text{send}](n_{A,B}) + \varphi^{send}_{trans}(n_{A,B}) = 96 \\ R_B &= C_B + Ovh_{Sys}[\text{recv}](n_{A,B}) + \varphi^{recv}_{trans}(n_{A,B}) + R_A = 194 \\ R_C &= C_C + Ovh_{Sys}[\text{recv}](n_{D,C}) + \varphi^{recv}_{trans}(n_{D,C}) + R_D = 158 \\ R_D &= C_D + Ovh_{Sys}[\text{send}](n_{D,C}) + \varphi^{send}_{trans}(n_{D,C}) = 74 \end{aligned}$$

The finishing times for the receiving tasks B and C contain the finishing times for their sending tasks. This has been done, because of the in-out dependency between a receiving and a sending task. For the classic worst-case analysis (e.g. see EDF* on page 70) the sending task is assumed to finish, before the message is transferred to the receiving task, which then can start its execution.

These results of the classical analysis can dramatically be improved, if the times, when the send and receive events occur, are considered. Especially, the response times of the receiving processes can be calculated more accurate. In fact, the destination process can continue its execution, when the process received the message. This time also depends on the time when the source process sends the last packet. For this reason, the time until the data are received have to be calculated as the maximum of either the destination process' execution time til the receive event plus n times of the time that is spent for the device driver's receive interrupt. Or it has to be calculated as the time of the source process' execution time til the send event plus the overall transmission delay φ^{send}_{trans} for a sending task plus the time that is spent for the device driver's receive interrupt:

$$\begin{aligned} R'_B &= \max(E_{recv}(B) + Ovh_{Sys}[\text{pre-recv}] + Ovh_{Sys}[\text{dd-recv}](n_{A,B}), \\ &\qquad E_{send}(A) + Ovh_{Sys}[\text{pre-send}] + \varphi^{recv}_{trans}(n_{A,B}) \\ &\qquad + Ovh_{Sys}[\text{dd-send}](n_{A,B}) + Ovh_{Sys}[\text{dd-recv}](1)\ ) \\ &\quad + Ovh_{Sys}[\text{irq-recv}] + Ovh_{Sys}[\text{post-recv}] + (C_B - E_{recv}(B)) \\ &= 99 \end{aligned}$$

$$\begin{aligned} R'_C &= \max(E_{recv}(C) + Ovh_{Sys}[\text{pre-recv}] + Ovh_{Sys}[\text{dd-recv}](n_{D,C}), \\ &\qquad E_{send}(D) + Ovh_{Sys}[\text{pre-send}] + \varphi^{recv}_{trans}(n_{D,C}) \\ &\qquad + Ovh_{Sys}[\text{dd-send}](n_{D,C}) + Ovh_{Sys}[\text{dd-recv}](1)\ ) \\ &\quad + Ovh_{Sys}[\text{irq-recv}] + Ovh_{Sys}[\text{post-recv}] + (C_C - E_{recv}(C)) \\ &= 89 \end{aligned}$$

These results can again be improved, if considered that the device driver's sending interrupt will be executed parallelly to the media access delay:

$$\varphi^{recv}_{trans}{}''(n) = \begin{cases} n * 2\,\varphi_{packet} & : Ovh_{Sys}[\text{dd-send}](1) \le \varphi_{packet} \\ n * \left(\varphi_{packet} + \left\lceil \frac{Ovh_{Sys}[\text{dd-send}](1)}{\varphi_{packet}} \right\rceil \varphi_{packet}\right) & : Ovh_{Sys}[\text{dd-send}](1) > \varphi_{packet} \end{cases}$$

$$\begin{aligned} R''_B &= \max(E_{recv}(B) + Ovh_{Sys}[\text{pre-recv}] + Ovh_{Sys}[\text{dd-recv}](n_{A,B}), \\ &\qquad E_{send}(A) + Ovh_{Sys}[\text{pre-send}] + \varphi^{recv}_{trans}{}''(n_{A,B}) \\ &\qquad + Ovh_{Sys}[\text{dd-recv}](1)\ ) \end{aligned}$$

$$
\begin{aligned}
& + Ovh_{Sys}[\text{irq-recv}] + Ovh_{Sys}[\text{post-recv}] + (C_B - E_{recv}(B)) && (6.1)\\
= \;& 91 \\
R''_C \;=\; & \max(E_{recv}(C) + Ovh_{Sys}[\text{pre-recv}] + Ovh_{Sys}[\text{dd-recv}](n_{D,C}) \,, \\
& \quad E_{send}(D) + Ovh_{Sys}[\text{pre-send}] + \varphi^{recv}_{trans}{}''(n_{D,C}) \\
& \quad + Ovh_{Sys}[\text{dd-recv}](1) \;) \\
& + Ovh_{Sys}[\text{irq-recv}] + Ovh_{Sys}[\text{post-recv}] + (C_C - E_{recv}(C)) && (6.2)\\
= \;& 83
\end{aligned}
$$

A time-triggered scheduling of the events and execution blocks from the model above is depicted in Figure 6.18. The results of the conservative worst-case approach could again be improved remarkably. This could be achieved, because for the receiver tasks the worst-case assumption of the media access delay for the first packet could be decreased. Task A needs no blocking for the media access for the first packet (6 time units). But, the device driver overhead for the first packet must be incorporated (2 time units). For this reason, the overall improvement consists of 4 time units compared to R''_B. This is similar for Task C and D. The blocking time for the media access for the first packet of task D are only 4 time units (instead of the worst-case assumption of 6 time units). But, again the device driver overhead have to be included. Thus, we achieve an improvement of $4 - 2 = 2$ time units compared to R''_C.

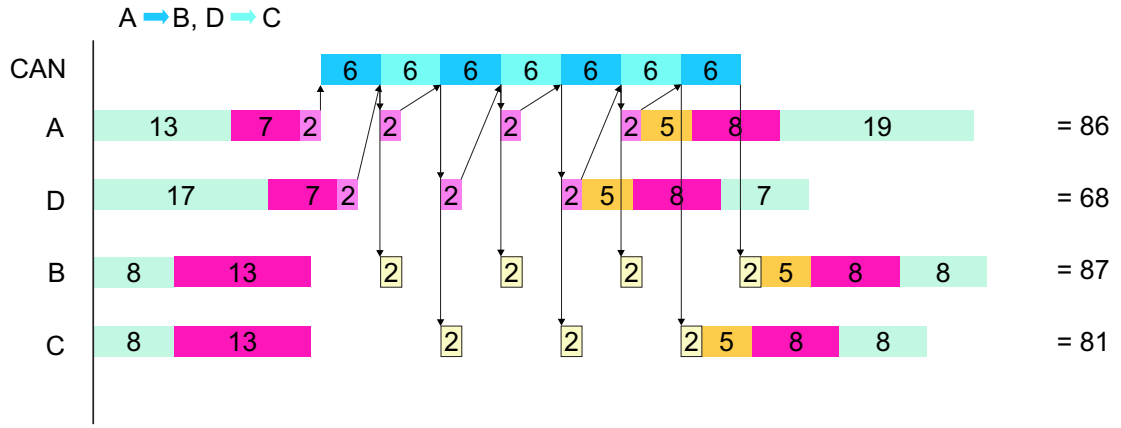

Fig. 6.18: Example 1 for the time-triggered scheduling where four tasks communicate via one single CAN bus.

But the real advantage of these time-triggered schedules is that the engineer sees what happens on each CPU and link. He can realise that the messages of both senders are really disturbing each other. A worst-case analysis only assumes this behaviour. By knowing this behaviour the engineer can re-programme the application. The re-programming can be required, if task C misses its deadline, while task B has enough time until its deadline is reached. He may choose the following solution: On the one hand he lets task D send its message much earlier. On the other hand, task A will send its message much later. All the execution times, the communication protocols, the routing and the placement will be the same. Thus, he specifies the following requirement specification:

$$\text{Task A} \quad : \quad C_A = 32, \;\; E_{send}(A) = 29$$

$$
\begin{aligned}
\text{Task B} &: C_B = 16, \quad E_{recv}(B) = 8 \\
\text{Task C} &: C_C = 16, \quad E_{recv}(C) = 8 \\
\text{Task D} &: C_D = 24, \quad E_{send}(D) = 1 \\
\text{Communication 1} &: \text{Task } A \longrightarrow \text{Task } B \text{ with } n_{A,B} = 4 \text{ packets} \\
\text{Communication 2} &: \text{Task } D \longrightarrow \text{Task } C \text{ with } n_{D,C} = 3 \text{ packets}
\end{aligned}
$$

Because the same tasks will send the same messages, the configurator will give the same operating system overheads. With this inputs the worst-case analysis of Formula 6.1 and 6.2 will yield to:

$$
\begin{aligned}
R''_B &= 107 \\
R''_C &= 67
\end{aligned}
$$

The time-triggered scheduling achieves much better results ($R'''_B = 91, R'''_C = 55$), which are taken from Figure 6.19. The calculated finishing times of the tasks are much better, because the worst-case assumption that a media access delay is as long as a complete message transfer time (6 time units) need not to be considered for this example. We can easily observe that no packet will be delayed, because the transmissions take place at different time intervals.

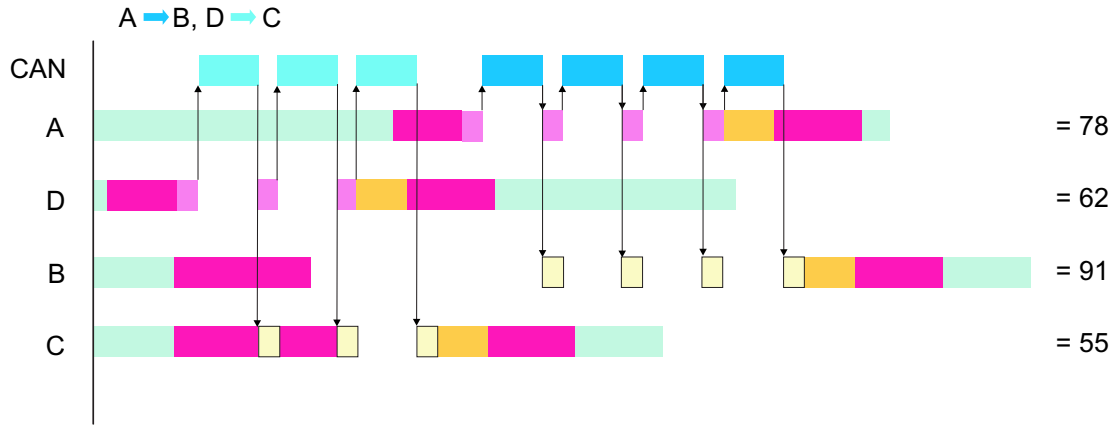

Fig. 6.19: Example 2 for the time-triggered scheduling where four tasks communicate via one single CAN bus, but the event of the second communication starts much earlier.

We can summarise that the configuration determines the following inputs for the timing analysis:

- Implementation of system calls (primitives)
  - Protocol and/or device driver that is used
  - Sequence of execution blocks to be executed per primitive
  - Duration of primitive (operating system overhead)
- Routing
  - Resource usage

It should clearly be stated that TEReCS, in fact, determines the primitive implementation from a set of a priori defined ones. These primitive implementations including the sequence and duration of their execution blocks for the timing analysis had been defined by operating system experts for different use cases. They had been integrated into the USG. TEReCS determines the correct use case upon the overall primitive usage of the complete application (see "Puppet Configuration" in Section 5.4$_{100}$) and some side constraints. Each primitive implementation is assigned a sequence of pre-defined execution blocks. Thus, TEReCS need neither to invent new execution blocks nor their sequence. TEReCS only selects the appropriate pre-defined set. This means that the sequence and duration of the execution blocks have to be defined in advance by the operating system experts (see Section 6.4.1$_{113}$).

6.10 Impact of the Timing Analysis on the Configuration

The results that are received from the previously presented *time-triggered event scheduling* of execution blocks have –of course– impact on the generation of a final configuration for the operating and communication system.

In the last example of the previous section (see Figure 6.19) the finishing time of the receiver task B was extremely prolonged, because the send event of task A was nearly moved to its end. For this reason, task B can miss its deadline.

Let us now additionally specify that a RS-232 peer-to-peer link exists between the processor nodes on which task C and D are placed (see Figure 6.20). Assume also that the requirement specification is the same as in the first example on page 142.

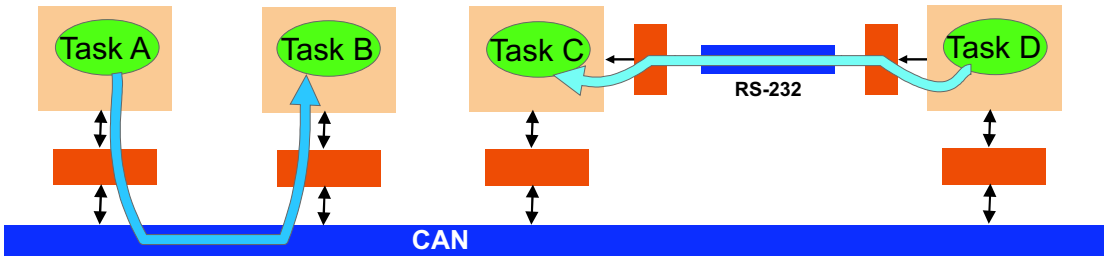

Fig. 6.20: Hardware architecture and task mapping, where four tasks on four processor nodes communicate via one CAN bus and one serial RS-232 peer-to-peer connection.

First, the router module of the TEReCS' configurator will map all communications of the example onto the CAN bus (because it is the fastest link). Thus, the timing analysis based on the time-triggered scheduling will receive the results from Figure 6.19. Let us assume again a deadline miss of task B. For this reason, the configuration has to be changed. This is primarily done by mapping other communications, which delay task B, onto other communication links if possible. In that example task B will be delayed by the communication between task C and D. But for exactly this communication the router can choose the other path via the RS-232 link. Thus, the configuration is again started, where it is prohibited to map the communication between task C and D onto the CAN bus. As a consequence the routing will change. Also the operating system overhead

and the protocol will change for the communication between task C and D. The time-triggered schedules for the new configuration are depicted in Figure 6.21. The finishing times of all tasks are decreased; but unfortunately, more resources have to be spent for the configuration.

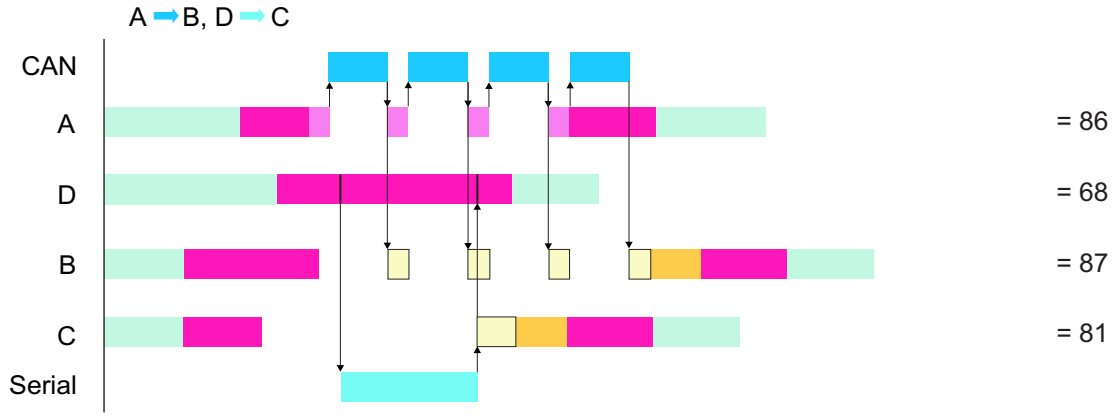

Fig. 6.21: Example 3 for the time-triggered scheduling where four tasks communicate via one CAN bus and one serial RS-232 connection.

This example shows in which way the timing analysis influences the configuration process. In order to exploit this feature, a lot of communication links have to be specified for the target hardware. The configuration process tries to use only a minimal set of these links for the application's communications. Which link has to be chosen first is determined by giving costs to each link. These can be direct or indirect costs. Direct costs can be, for instance, the price of the hardware. Indirect costs are produced by the software services that have to be selected in order to manage the link.

As we have seen, an additional link is used, if a communication that produces a deadline miss must be newly routed. In TEReCS communications are distinguished that directly or indirectly produce a deadline miss. A communication directly causes the deadline miss, if, for instance, the communication is between tasks A and B and task B misses its deadline. The communication indirectly causes a deadline miss, if the communication is between task A and B, but it delays another communication between tasks C and D, so that task D misses its deadline. When a deadline miss occurs, first a communication that directly causes the miss is tried to be re-mapped. If this is not possible or does not achieve success, then the communications that indirectly cause the miss are tried to be re-mapped.

In this way, TEReCS can determine the minimal set of required communication links between the processor nodes. TEReCS implements and routes all communications in such a way that they do not cause a deadline miss of any task. But obviously, this can be impossible, if there are not enough resources available.

The routing of all communications is one main part of the final configuration. Whereas this is nearly done automatically in TEReCS, the mapping of the tasks onto the processor nodes must manually be done by the engineer. But when no solution can be found, the engineer can identify tasks from the time-triggered schedules, which can be re-mapped onto other processors in order to eliminate a resource overload condition.

Besides the automatic communication link selection, the routing, and the manual task mapping, also parameters, properties, or modes of the communication links can be determined. For example, the baud-rate should be selected for a RS-232 serial peer-to-peer link. This can be done in TEReCS by specifying two RS-232 links between the same processor nodes, where one link transmits the bits with, for instance, 9600 Baud and the second link uses 38400 Baud. If a lower baud-rate is preferred (because the link is then less susceptible to electro-magnetic disturbances), then the link with the lower baud-rate should cause lower direct costs.

In the same sense as TEReCS selects the appropriate communication links, TEReCS selects appropriate devices in order to access the links from the processors. For instance, the devices for a CAN bus access vary in their performance and, therefore, also in their costs.

Besides these more or less direct influences of the timing analysis onto the final configuration, there also exists indirect influences. Indirect influences have impact on the operating system of a processor node. This is obvious, since the change of a communication link and/or a device means also to change the device driver inside the operating system. But, additionally, the access to the device can change. That means, for instance, the access must be protected, if more than one task uses simultaneously the driver. Or the buffering scheme for the messages must change, because the messages have the same or different priorities. Also the buffer size depends on the number of messages that are mapped onto a certain connection. All these indirect impacts on the software architecture change the operating system's overhead. This can also help to find a feasible schedule over all processors.

We can summarise that the timing analysis has the following influences on the final configuration:

- Direct impacts:
  - Prohibiting certain communications to be mapped onto specific links and/or devices ($\longrightarrow$ automatic re-routing)
  - Manual re-mapping of tasks onto processors
- Indirect impacts:
  - Parameter or mode changes of the links and devices
  - Change of the software architecture of the operating system
  - Change of the overhead of the operating system services

These impacts represent some of the main advantages of TEReCS. TEReCS determines a valid communication system, which means that all communications and, therefore, also the tasks can feasibly be scheduled. For this reason, the appropriate links, devices, and operating system services per processor are selected, respectively configured.

6.11 Contribution of the Chapter

In this chapter we have presented a new methodology for the automatic configuration of operating and communication systems. Atomic services of the operating systems can

be included or excluded from the final implementations. This is primarily done to save memory. Also, there exists multiple implementations for the same service. Each service implementation is only applicable to a specific use case. Thus, the lack of a service A can result to force service B to be implemented in a special way. Or, it is known that the application behaves in a certain way, then a service C can ignore a special use case and may be implemented more efficiently. By this way, the integration of application specific knowledge about its behaviour into the operating system can save memory for the services' implementation as well as execution overhead. The approach that is used here is named *"Puppet Configuration"*.

Similar benefits can be achieved for the communication system. It is the task of the configuration process to determine the routing of the messages through the network and to integrate or exclude link device drivers into the appropriate operating system of each node. The routing tables have to generated and the existing device drivers must be parameterised (address, port, protocol, access protection, etc.). It is also the task of the configuration process to determine the minimal required links of the network, so that the timeliness transfer of all messages is just possible. Because we assume to have a distributed real-time application, the check for deadline misses must be done. Therefore, TEReCS has also integrated a verification tool for the timeliness execution of all communication tasks. The maximal communication end-to-end delays are determined as well as the schedules of all the messages on the media and of all the processes on the nodes. This is done by a so-called *"Time-triggered Event Scheduling"*.

The configuration algorithm of TEReCS follows a structure-based approach. In fact the knowledge base representation is modelled as an AND/OR-DAG with a top-down concept. It supports for constraints, which are in TEReCS a powerful mechanism to determine the correct choice for alternative implementations upon a specific use case. The model also supports the specification of parameters, which are used for a concrete selection. The *Puppet Configuration* approach assumes that (due to the constraints) only valid and functionally correct configurations can be generated. Thus, only the real-time aspect must be verified. The configuration works without user interaction and, therefore, is automatic. The user only has to provide a requirements specification of the application. Operating system details are completely hidden from the user. The configuration algorithm implements a design space exploration of the operating system under consideration. The modelled graph is traversed and the selection of alternative paths are made upon a cost function and the constraints. Here, the graph is traversed in a top-down manner for each required primitive to the required hardware. For the communication services the routing is considered. Here, also hops for messages in a multi-hop network may occur. The graphs of each node of a message path is traversed accordingly from the send primitive to a device, from a device to another device, or from the device to the receive primitive.

The configuration model and also the configuration algorithm allow hierarchy. Therefore, a graph can be clustered. A cluster represents a sub-graph and appears to be a simple service node in the graph of the next hierarchy level. By this way, the complexity of a graph can dramatically be reduced. Additionally, this supports a faster re-configuration. If the requirements specification is only slightly changed, then only those cluster-graphs have to be re-configured that are involved into those changes. This is especially true for the configuration loop of the configuration and analysis phase. The hierarchy also helps to minimize the memory that is used during the configuration. A cluster-graph, which

is not selected, must not be loaded for the configuration into the memory (information hiding). Another advantage of the clustered hierarchy is that it can help to steer the configuration algorithm. When the clustering and, therefore, also the hierarchy is calculated just before the configuration. Then also the requirements specification can be integrated. Thus, different requirements specifications can lead to different clusterings, which may result into different configuration runs.

The primary goal of the *Time-triggered Event Scheduling* for the timing analysis is to check, whether all tasks meet their deadlines. The aspects of considering distributed communication systems and the configuration itself arise some problems that are often ignored in the classical and theoretical timing analysis. Precedence constraints have to be considered due to the complex in/out-dependencies. Also the resource access constraints on the links (media access) result into unknown blocking delays. Their calculation is very complex and is often very pessimistic. These reasons make the analysis quite complex and lead often to very pessimistic assumptions. Additionally, the classical timing analysis ignores the system overhead (task switching, etc.). But in TEReCS it is essential to incorporate this, because it can change due to different configurations. That means that the configuration has impact on the timing of the system. A solution for this problem was presented in form of the optimistic *Time-triggered Event Scheduling* scheme. Although the results of that analysis may not exactly reflect the real task execution, they can give hints for the detection of bottlenecks in the system. However, by the integration of *event guards* the execution can follow the time-triggered approach and will match the analysed schedules. Additionally, the possibility to define event intervals, the heuristic evaluation, and the very exhaustive search (exponential) on all possible schedules achieves in practice good results. They are seldomly too optimistic.

CHAPTER 7

Results

Currently, DREAMS consists of about 280 classes out of which about 100 classes are customisable *class skeletons*. These led to approximately 200 *services* that are modelled with about 220 SAPs and 300 SRPs. The reason for having twice more services than skeletons is based on the fact that every Skeleton has an API (Application Programming Interface) *symbol*, which defines the access interface to that type of skeleton. This complexity makes it quite difficult to create manually a good and working configuration. TEReCS can manage this very easy. A user of DREAMS need not to be an expert for all of the configuration items of DREAMS.

After modelling DREAMS several application scenarios with different demands to the operating system were defined. These scenarios include different memory management strategies and use or exclude interrupt facilities, multi-threading, multiplexing of shared resources, local or external communication, debugging facilities, terminal output, etc., on different hardware architectures. The resulting configurations for each scenario were valid and no mal-configurations could be detected by an "expert" for DREAMS. On the contrary, some configurations were better than manually made ones. There were some surprising decisions of which the expert had not thought of, but led to better implementations.

Moreover, the configuration was developed nearly 120 times faster than doing it manually (in the worst-case the manual configuration needs approx. 60 min; automatic approx. 30 sec). Additionally, it could be verified that the target code size of the systems varied very much. The minimal system reached a size of about 2.4 KBytes whereas the biggest one (using all features of the system) required about 150 KBytes of memory for code and data. Both systems were made for a 32 Bit processor (PowerPC). This shows impressively the great impact of configuration on the size of the system. Likewise, it is expected that the execution time also differs much [43]. Therefore, a demonstration scenario had been developed. The communication system had been configured differently (mainly by using other protocols). The end-to-end delay of a communication and the service intervals had been measured by software monitors.

7.1 Demonstrator

In order to measure quantitative benefits of the operating and communication system's configuration, DREAMS was extended by special customisable options. For the test scenario the rapid-prototyping hardware USAP from the ETAS GmbH, Germany, was used. This hardware comprises a 19 inch rack with power-supply and a proprietary backplane and up to 6 extension slots. It is extended by 4 processor boards that are equipped with a 20 MHz PowerPC 604 with 8 MBytes DRAM memory in combination with a 10 MHz Transputer T805 with an extra 1 MByte SRAM. The 8 MBytes DRAM are implemented as shared memory between the two processors. The backplane connects the Transputers of each slot in a sequential manner. The first slot is equipped with a single Transputer T405 and only 256 KBytes of memory. One of the links of this Transputer can be reached from outside, for example, from a host system. The hardware topology of this system is described in TEReCS' notation in Figure A.1_{166} in Appendix A.

DREAMS can actually be customised to run on Transputers and on PowerPC 60x, PowerPC 509, and PowerPC 555. The communication system supports for Transputer links, for serial RS-232 interfaces, like the TL16C552A chip (used in the PowerPC 509) or the SCI (*Serial Communication Interface*) of the PowerPC 555, for CAN devices, like the one which is used in the PowerPC 555 chip, and for a shared memory interface in order to exchange messages. Device drivers for these devices can be integrated accordingly to the usage and present hardware. The device driver can be enabled or disabled to listen for incoming messages. The listening for each device is implemented as a virtual interrupt service routine. An optional routing service can be integrated into the operating system, which selects the appropriate outgoing device upon a routing table. The routing service is implemented as a single task. Incoming messages will be redirected to the internal mailboxes, if addressed, or to the routing service, if present. Each communication device, each mailbox and the routing service are implemented as ports, which provide a unified *message exchange interface*. This provides for universally connected ports. Thus, the in-port of a communication device can be connected directly to the out-port of an internal mailbox, to another communication device, or to the routing service. Each port can be implemented as a resource or not. When the port is implemented as a resource, each access (of `send()` and `receive()`) is protected by a semaphore (*mutual exclusion*).

For the serial links (Transputer and RS-232) three different protocols are alternatively selectable:

Generic. The first protocol variant is a generic protocol. It transfers a *header* before the user data (*message body*) are transmitted. The header contains information about the destination node, the destination mailbox, and the size of the user data. The header's size and structure is fixed. The size of the user data can vary, but must be a multiple of one byte. The header can be received asynchronously to the message body. But each user data must be preceded by one header. Thus, after receiving a header, the memory for the user data (which can be of different size) are allocated in order to store and forward the message.

Maximum Size. The second protocol variant transmits the header and body at once as one packet. Both must be received in one stream. The size of the packet and, therefore, also of the message (user data) is constant per link. The header also contains

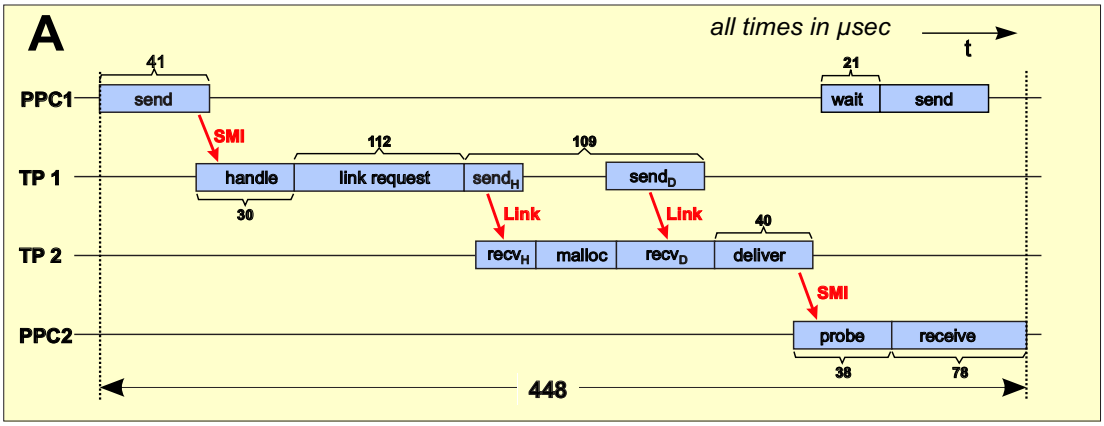

Fig. 7.1: Measured schedule for the transfer of 64 Bytes user data over four processor nodes. The Transputer link must be requested and the *Generic* protocol is used.

a size field, because the user data of the message can be less than the size of the packet's body. The advantage is, that the memory for a message can be allocated asynchronously before a packet is received. Additionally, the sender must not access the media twice: for the header and for the body. The send delay will dramatically be reduced.

Constant. The third protocol variant does not transmit any header before the user data are sent. This assumes that each received packet has the same destination, which is statically known, and always has the same size.

Other protocols (like mixtures of these protocol variants) can possibly be requested; but are actually not implemented. The configuration for each node of the target system must determine the following items:

- Processor type
- List of the required communication devices that defines per device:
  - Identifier
  - Type
  - Base address
  - Boolean flag for the resource protection
  - The protocol variant that is to be used for the `send()` and `receive()` methods
  - The local out-port to which the in-port of this device is connected
- Routing table which contains for each destination (node or channel identifier) the local outgoing device identifier

These things are automatically determined by the configurator tool TGEN of TEReCS for DREAMS. This comprises especially, whether all messages that are sent via one specific link have a maximum or fixed size and, additionally, have the same destination. Thus, one of the three protocol variants can be selected. Whether the communication device is used exclusively by one task or whether it is used simultaneously by multiple tasks is

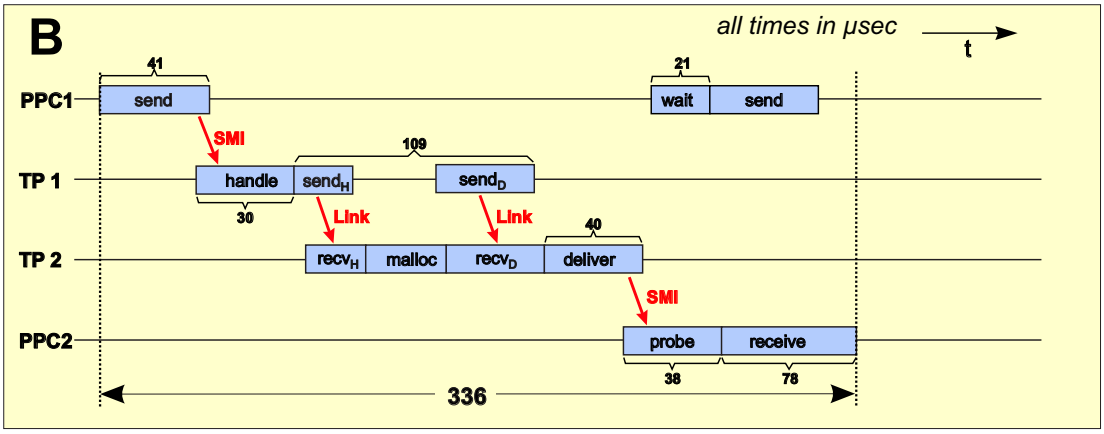

Fig. 7.2: Measured schedule for the transfer of 64 Bytes user data over four processor nodes. The Transputer link is not requested and the *Generic* protocol is used.

noticed (interrupt service routines are also seen as tasks). Thus, the resource protection can be selected or removed. All these things are described within alternatives of the URSG.

It is also recognised that all messages that are received from the same in-port must possibly be redirected to the same local out-port. This means that the in-port will be connected to that out-port or, if they do not have the same destination, it will be connected to the out-port of the routing service. It is also recognised that all messages that have to be sent via one out-port have a maximum or fixed message size. For these reasons, the protocol variant can be determined to be chosen.

For the example a task A on the PowerPC 5 wants to transmit 64 Bytes of user data to the task B on the PowerPC 3 (see Figure A.1$_{166}$ and Figure A.3$_{175}$). Therefore, the data must be routed via the *shared memory interface* (SMI) to the board's communication processor, which is the Transputer 4. This Transputer has to forward the data to its neighboring Transputer 2, which then stores the message. The message is stored here, because it has to be delivered via the SMI to its assigned PowerPC 3. When the task B wants to receive the

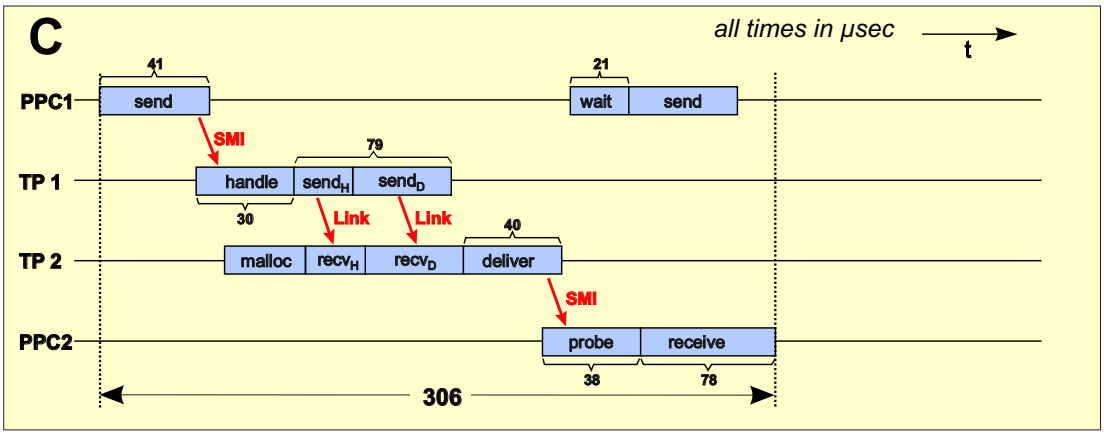

Fig. 7.3: Measured schedule for the transfer of 64 Bytes user data over four processor nodes. The Transputer link is not requested and the *Maximum Size* protocol is used.

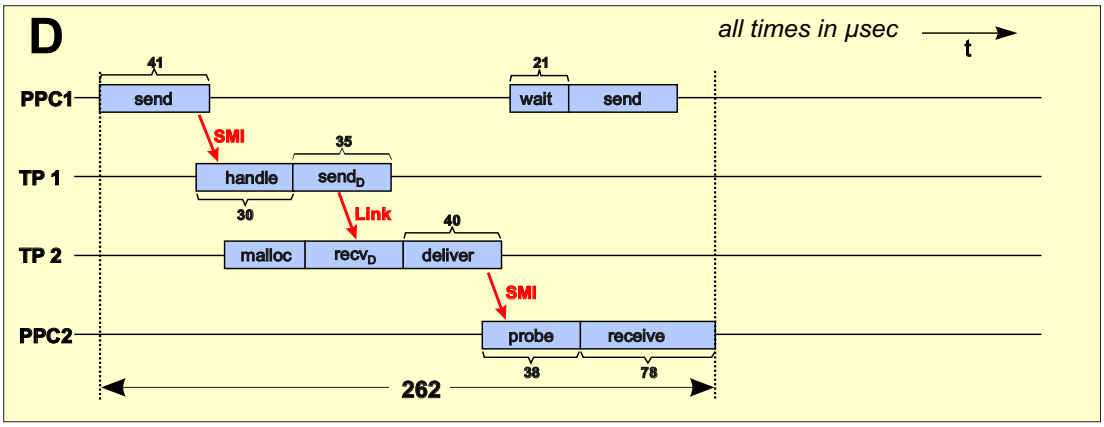

Fig. 7.4: Measured schedule for the transfer of 64 Bytes user data over four processor nodes. The Transputer link is not requested and the *Constant* protocol is used.

message, the client service `receive()` has to contact the server service `deliver()` on its assigned Transputer 2. Then the message is transfered over the SMI (for the definition of neighbouring client/server communication see paragraph "Considering Communications" on page 119).

No other communications occur in the system. For this reason, the `handleTP()` service of the SMI of the first Transputer can be directly connected to the device driver `put()` for the appropriate out-link. A routing service need not to be used, because there is only one message, which always has the same destination. Similar, the `get()` service of the in-link on the second Transputer is connected directly to the `deliver()` service. The `deliver()` service forwards the message on demand as a server via the SMI to its assigned PowerPC. The *Universal Service Graph* (USG) and the final configuration result are described in Appendix A.

The `put()` and `get()` services of the device driver for the Transputer link can also be configured to implement one of the three previously described protocol variants. We have measured on each processor the start and end times of certain communication services.

Four different scenarios had been considered:

A The send link on the first Transputer is protected by a semaphore and, therefore, must be requested. Also, the first protocol *"Generic"* is used. The overall end-to-end delay from the start of the `send()` on the first PowerPC until the end of the `receive()` on the second PowerPC is 448 μsec (see Figure 7.1).

B The link of the first Transputer is not protected. The protocol variant *"Generic"* is still used. Because of omitting the link request the overall end-to-end delay can be reduced to 336 μsec (see Figure 7.2).

C The link of the first Transputer is not protected and the second protocol variant *"Maximum Size"* is used. The memory allocation service on the second Transputer is re-arranged. This saves 30 μsec (see Figure 7.3).

D The link of the first Transputer is still not protected and the third protocol variant *"Constant"* is used. Here, the sending of a header is eliminated. In this case the

overall end-to-end delay takes only 262 μsec (see Figure 7.4).

These examples show impressively that the communication delay can dramatically be influenced by different configurations due to different application scenarios. It is the task of the configuration process to determine the best implementation by considering the application's behaviour.

These examples only consider the serial link communication of the Transputers. However, this example can directly be applied to a serial RS-232 communication and it can be adapted to other communication devices with other protocol variants. For instance, when a CAN bus is considered, protocol variants with and without message fragmentation can be assumed, etc.

CHAPTER 8

Conclusion

The tool suite that has been developed for TEReCS consists of the configurator TGEN, the timing analyser TANA, the creator TCLUSTER for the creation of a requirement specific cluster hierarchy and its configuration with TGEN, the editor TEDIT for the maintenance of the URSG and RG, and the design tool TDESIGN, which implements TEReCS' design cycle. TDESIGN calls TGEN and TANA until a valid configuration could be found. TDESIGN also edits appropriately the routing constraints of the requirement specification due to the timing problems detected by TANA.

TGEN implements a very flexible and powerful method for the application specific design of operating and communication systems for distributed embedded systems. The configuration approach helps to reuse software services and to maintain different possible implementation alternatives. Configuration with highly flexible service dependencies and an appropriate cost function can optimise the operating system overhead and its memory footprint.

During the exploration of this approach it had been revealed that configuration of software components increase dramatically their reuse. Contradictory goals, respectively trade-offs, for example, between performance and flexibility, become highly adjustable. Additionally, the use of design principles for the automated configuration process of runtime platforms, which are well known for the high-level hardware design, showed that operating systems need not to be considered as statically fixed monolithic systems. Instead, they can be seen as appropriate mediators between the hardware and the application. The operating and communication system can be individually adapted to the concrete demands of the application. Hereby, the overall performance of the operating and communication system can be optimised.

A very contemporary approach is the configuration of very fine-grained tunable operating systems. Only the customisation at object-level in the source code makes it possible to handle the trade-off between optimality and flexibility. However, the huge amount of customisable options increases the complexity of a system extremely. It could be verified

that TEReCS with its configuration approach is able to handle this complexity. Moreover, the automatic configuration process requires significant less knowledge from the application programmer about the customisable operating system. Operating system internals can completely be hidden from the user.

Additionally, the description of the valid design space of the operating system within TEReCS' (hierarchical) URSG can serve as a fine-grained documentation for the operating system. Thus, the integration and development of extensions to the system is more easier. The operating system experts can use TEDIT also as a tool for the exploration of the operating system's internal structures, capabilities, and restrictions. For this reason, TEReCS also supports the operating system experts to maintain the operating systems immanent complexity due to the configuration aspect.

The analyser TANA inspects the final configuration before it is targeted. By using atomic executions and their arrangement into a schedule sophisticated statements about the system behaviour can be made. The *Time-triggered Event Scheduling* of execution blocks creates a lot of information that can help to re-configure the system more optimal. Bottlenecks, overload conditions, bursts, and also idle phases are detected and influence the configurator. The mutual influences between the configurator TGEN and the analyser TANA are managed by TDESIGN. Moreover, the load distribution or routing of messages through the (static and fixed) network can be automated in the way that all timing constraints are just met.

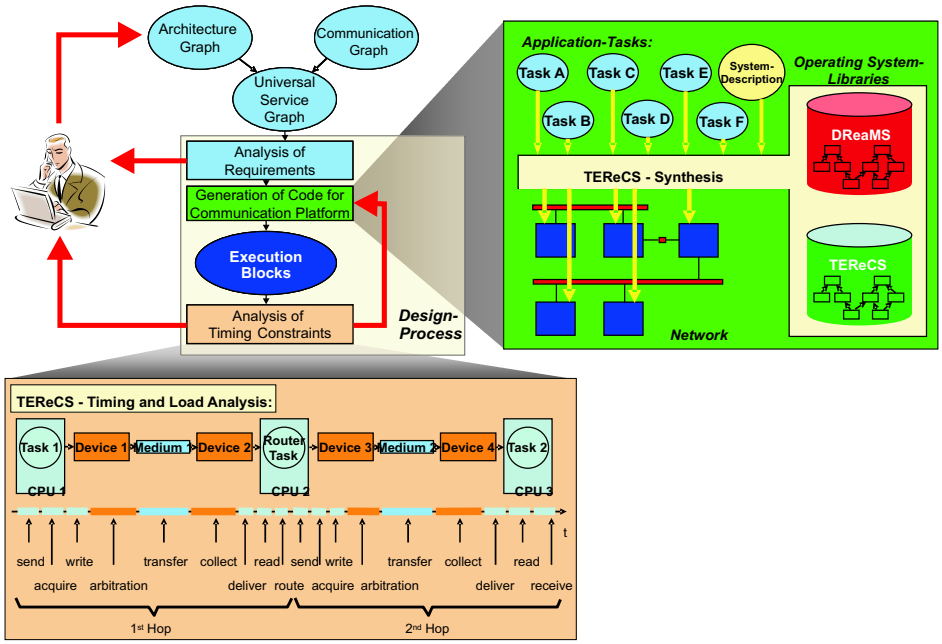

Fig. 8.1: The design cycle of specification, configuration and analysis in TEReCS.

This innovative design cycle (see Figure 8.1) for the targeting of distributed real-time applications with its software reuse and its pre-implementation analysis dramatically decreases the development time of embedded applications and their operating and communication systems. This is an important factor to speed-up the time-to-market of a product.

TEReCS can serve as a front-end for any customisable real-time operating system. Especially, it is very adequate for object-oriented modelled systems. For the timing analysis it is desirable that the atomic operating system services assure a deterministic worst-case execution time that is very similar to the average case. When enough operating system services for different hardware architectures exist as alternatives, TEReCS can help to develop hardware independent embedded applications and their execution platforms, which are comparable to hand-written code.

TEReCS is conceptually integrated into a more general design framework (see Figure 8.2) for the development of distributed embedded applications that is called PARADISE [2, 59]. PARADISE supports also for the development of hardware implementations in ASICs or FPGAs. An Internet-based repository for *Intellectual Property* (IP) cores and software services supports the protection of the IP of third-party companies that offer operating system or application extensions as software or hardware services.

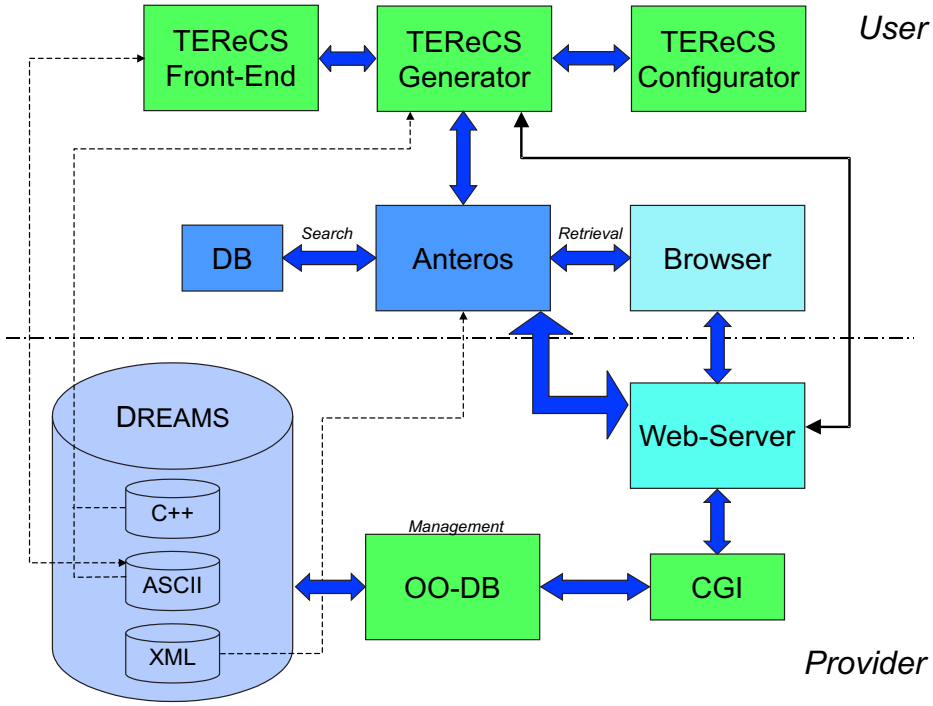

Fig. 8.2: TEReCS is conceptually embedded into an Internet-based framework for supporting third-party developments.

TEReCS had also been applied to a very special application scenario. While TEReCS had been applied to the DREAMS operating system library, TEReCS was responsible to generate an execution platform for Pr/T-net-based applications for distributed embedded control units [18, 110]. Applications, which are specified in an extended version of predicate transition nets, have been targeted to an embedded processor network. The back-end and code generator for the Pr/T-net specifications had been adapted to TEReCS. Then TEReCS constructed an optimal run-time platform for the execution of the distributed Pr/T-net. A very deep integration of the application (Pr/T-net) and the operating system in form of a *Pr/T-net Execution Engine* could be achieved.

It has been shown that the TEReCS approach makes the modelling of the configuration options of a customisable object-oriented execution platform possible. The design cycle for the development of a nearly optimal adapted run-time platform could extremely be shortened. The reuse of hardware independent software for different applications, as well as hardware dependent software when changing the target system, is excellently supported. The design methodology suggested by TEReCS works fine.

8.1 Overview of Publications of the Author Related to this Work

The overall concept that led to the current implementation of TEReCS had first been published in 1997 within the first research proposal "Entwurf konfigurierbarer, echtzeitfähiger Kommunikationssysteme" [104] for the priority programme 1040 "Entwurf und Entwurfsmethodik eingebetteter Systeme", which has been founded by the Deutsche Forschungsgemeinschaft (DFG). A summary of the developed concepts of that proposal could also be found in an article of the journal *Informationstechnik und Technische Informatik* (it+ti, no. 2, 1999) [19].

Major parts of the Sections 6.2, 6.3, and 6.4 had been published in the proceedings of the 6^{th} *Annual Australasian IFIP Conference on Parallel Real-time Systems* (PART'99) [14] and in the proceedings of the *Forum of Design Languages* (FDL'99) [16].

Section 6.5 is mainly based on an article that had been presented at the 5^{th} *IEEE International Symposium on Object-oriented Real-time Distributed Computing* (ISORC'02) [28].

The main concept that is presented in Section 6.6 about the data model had been published on the *Workshop on Architectures of Embedded Systems* (AES 2000) [15].

Some of the results that are presented in Section 7.1 had been taken from a publication of the proceedings of the 15^{th} *IFAC Workshop on Distributed Control Systems* (DCCS'98) [43].

The integration of TEReCS into the overall design framework PARADISE was published in the proceedings of the *International IFIP WG 10.3/WG 10.5 Workshop on Distributed Parallel Embedded Systems* (DIPES'98) [2] and in the EUROMICRO Journal in 2000 [59].

The design of a Pr/T-Net Execution Engine had been presented on the *International European Simulation Multi-Conference* (ESM 2000) [18] and on the *International Conference on Application and Theory of Petri Nets* (ICATPN'01) [110].

Further application scenarios for (re-)configurable operating systems had been presented on the *Workshop of Object-oriented Real-time Dependable Systems* (WORDS'03) [17].

8.2 Outlook

There are still some open problems and things to be done for TEReCS. Up to now it is left open to show the formal correctness of the generated configuration or of the generator tool itself. Up to now, TEReCS assumes that the domain knowledge description (URSG) allows only valid and "correct" configurations to be built. Actually, the real correctness is validated manually by "inspection" of the result by an expert. If an error is found, then a new constraint is integrated into the URSG in order to prevent the wrong configuration. By this way, the problem of generating a correct configuration is transformed into the

problem to specify a correct knowledge base. TEReCS assumes that this problem must be handled by the operating system expert. Nevertheless, TEReCS supports the expert in some ways. For example, cyclic (or contradictory) constraints are detected during the configuration and syntactical wrong configurations are detected by the compiler during the generation phase.

A possible extension to TEReCS would be the integration of dynamic dependencies between the services. Actually, there are only static dependencies allowed. Dynamic relationships between the services should comprise temporal orders of sub-service requests or even the specification of repeated requests as they occur in loops. Additionally, the requests may depend on parameters that have been assigned to the primitive calls. These dynamic dependencies should allow to specify the control flow of the services. Thus, the timing analysis can more specifically model the system's behaviour.

The timing analysis can also be improved. Before the schedules are calculated the load of the resources should be checked to be less than 100% or another specified limit. By this way, overload conditions can be detected much faster. Moreover, the schedules need not to be calculated when the maximum capacity is exceeded.

The design loop of TEReCS already determines the routing of the messages through the network. If alternatives exist, then the cheapest routes will be chosen first. A re-routing and the use of additional links is required when the timing analysis detects problems. By this way, a minimum set of communication links can be determined. Of course, this simple mechanism of "error and try" can be improved by a better routing and cost function for the selection of the links. Additionally, also the mapping of the tasks onto the processors can be determined similarly. This had been excluded from TEReCS, because there exists good tools, which especially handle this problem.

Another possible enhancement for TEReCS would be a visualisation tool for the presentation of the schedule outputs after the analysis. Actually, the analyser TANA generates only textual descriptions of the schedules. A presentation in form of a Gantt diagram would be more user friendly.

The most interesting extension of TEReCS is to develop a methodology in order to cope with the configuration of an operating and communication system during its run-time. This dynamic configuration approach is a major topic in the Special Research Initiative SFB 614 "Self-optimising Concepts and Structures in Mechanical Engineering" of the *Deutsche Forschungsgemeinschaft*, which is implemented at the University of Paderborn, Germany. Herein, the project C2 "Self-optimising Operating System" deals with this problem. The project C2 can be seen as the successor of the TEReCS project. The project goes towards a reflective and self-adaptive operating system in order to support changing application scenarios and changing hardware resources due to self-optimisation.

Another very challenging vision for future research in the area of TEReCS would be the generalisation of the configuration for the operating system's domain towards the application domain. The software synthesis approach of TEReCS should not only be applied to the operating system but also to the complete development process of the application.

However, the work on TEReCS had been a great inspiration and led to a lot of new ideas and also to some further questions.

* * *

APPENDIX A

Simple Configuration Example

Hardware and Topology Description

The hardware for the configuration example consists of 4 Transputers and 3 PowerPCs. The Tranputers are interconnected in the given topology by their Transputer Links. Three of the four Transputer boards have also a PowerPC processor on board. The Transputer and its neighbouring PowerPC are connected via a shared memory interface (see Figure A.1).

The following text describes the *Resource Graph* (RG) of Figure A.1:

```
// ---------------------------------------------------------------------------
// declaration and instantiation of hardware components:
// ---------------------------------------------------------------------------

HARDWARE
   DEF_CPU(0,0,0,Transputer)
   DEF_CPU(2,2,0,Transputer)
   DEF_CPU(4,4,0,Transputer)
   DEF_CPU(6,6,0,Transputer)
   DEF_CPU(3,3,1,PowerPC)
   DEF_CPU(5,5,3,PowerPC)
   DEF_CPU(7,7,5,PowerPC)
   DEF_MEDIA(0,shared_memory)
   DEF_MEDIA(1,shared_memory)
   DEF_MEDIA(2,shared_memory)
   DEF_MEDIA(3,link)
   DEF_MEDIA(4,link)
   DEF_MEDIA(5,link)
   DEF_MEDIA(6,link)
   DEF_MEDIA(7,link)
   DEF_MEDIA(8,link)
   DEF_DEVICE(0,PPCMemDev)
   DEF_DEVICE(1,PPCMemDev)
   DEF_DEVICE(2,PPCMemDev)
```

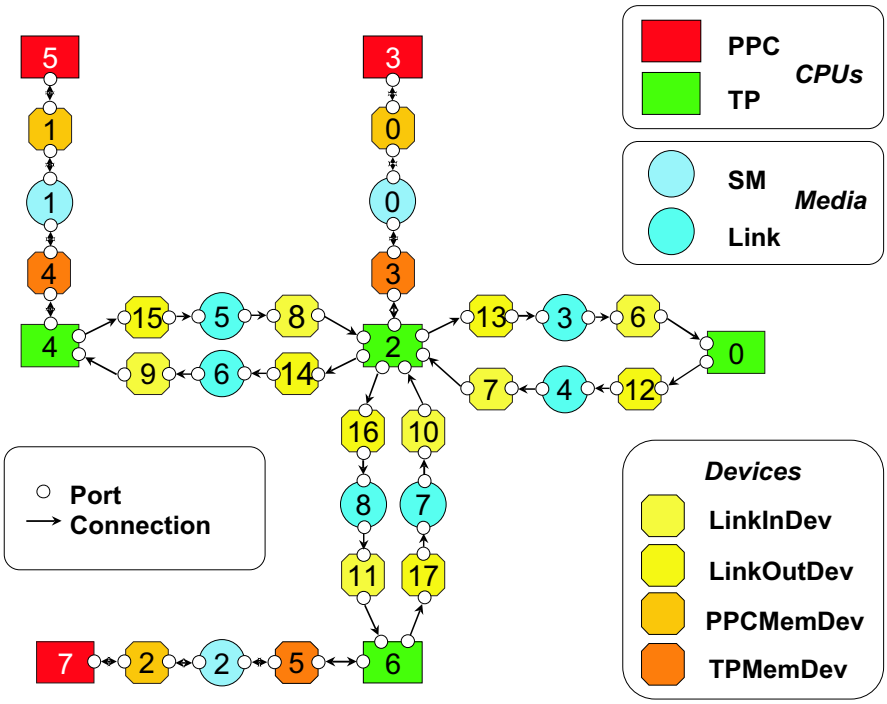

Fig. A.1: Example for a *Resource Graph* (RG).

```
   DEF_DEVICE(3,TPMemDev)
   DEF_DEVICE(4,TPMemDev)
   DEF_DEVICE(5,TPMemDev)
   DEF_DEVICE(6,LinkInDev)
   DEF_DEVICE(7,LinkInDev)
   DEF_DEVICE(8,LinkInDev)
   DEF_DEVICE(9,LinkInDev)
   DEF_DEVICE(10,LinkInDev)
   DEF_DEVICE(11,LinkInDev)
   DEF_DEVICE(12,LinkOutDev)
   DEF_DEVICE(13,LinkOutDev)
   DEF_DEVICE(14,LinkOutDev)
   DEF_DEVICE(15,LinkOutDev)
   DEF_DEVICE(16,LinkOutDev)
   DEF_DEVICE(17,LinkOutDev)
END_HARDWARE

// ------------------------------------------------------------------------
// declaration and instantiation of the topology:
// ------------------------------------------------------------------------

TOPOLOGY
   CONNECT_CPU_DEV( 0, 12, link_out, 2)
   CONNECT_CPU_DEV( 2, 13, link_out, 0)
   CONNECT_CPU_DEV( 2, 14, link_out, 1)
   CONNECT_CPU_DEV( 2, 16, link_out, 2)
   CONNECT_CPU_DEV( 4, 15, link_out, 0)
   CONNECT_CPU_DEV( 6, 17, link_out, 3)

   CONNECT_CPU_DEV( 0,  6, link_in,  2)
   CONNECT_CPU_DEV( 2,  7, link_in,  0)
```

```
   CONNECT_CPU_DEV( 2,  8, link_in,  1)
   CONNECT_CPU_DEV( 2, 10, link_in,  2)
   CONNECT_CPU_DEV( 4,  9, link_in,  0)
   CONNECT_CPU_DEV( 6, 11, link_in,  3)

   CONNECT_CPU_DEV( 2,  3, TP_SM,    4)
   CONNECT_CPU_DEV( 4,  4, TP_SM,    4)
   CONNECT_CPU_DEV( 6,  5, TP_SM,    4)

   CONNECT_CPU_DEV( 3,  0, PPC_SM,   0)
   CONNECT_CPU_DEV( 5,  1, PPC_SM,   0)
   CONNECT_CPU_DEV( 7,  2, PPC_SM,   0)

   CONNECT_DEV_MED(12,  4, OutLinkP)
   CONNECT_DEV_MED(13,  3, OutLinkP)
   CONNECT_DEV_MED(16,  8, OutLinkP)
   CONNECT_DEV_MED(17,  7, OutLinkP)
   CONNECT_DEV_MED(14,  6, OutLinkP)
   CONNECT_DEV_MED(15,  5, OutLinkP)

   CONNECT_DEV_MED( 7,  4, InLinkP)
   CONNECT_DEV_MED( 6,  3, InLinkP)
   CONNECT_DEV_MED(10,  7, InLinkP)
   CONNECT_DEV_MED(11,  8, InLinkP)
   CONNECT_DEV_MED( 8,  5, InLinkP)
   CONNECT_DEV_MED( 9,  6, InLinkP)

   CONNECT_DEV_MED( 3,  0, IO2SMP)
   CONNECT_DEV_MED( 4,  1, IO2SMP)
   CONNECT_DEV_MED( 5,  2, IO2SMP)

   CONNECT_DEV_MED( 0,  0, IO1SMP)
   CONNECT_DEV_MED( 1,  1, IO1SMP)
   CONNECT_DEV_MED( 2,  2, IO1SMP)
END_TOPOLOGY
```

Domain Knowledge Database

For the example the application tasks can only be implemented on the PowerPC processors. The Transputers act as communication co-processors. TEReCS is responsible to generate appropriate communication services on the PowerPCs and Transputers. Messages, which have to be transferred from one PowerPC to another, will be stored in the Transputer's memory of the last Transputer on the path. The message will be stored there until the message is requested from the receiving task on the assigned PowerPC. Messages will also be routed and forwarded through the Transputer network.

The only two *user primitives* that can be called from applications on the PowerPCs are `send()` and `receive()`. Both services will call the service `informTP` of the device driver for the shared memory interface. The use of this service will always activate the service `handleTP` of the coresponding device driver interface on the Transputer. A message that has to be sent can possibly transferred by using three alternative sub-services `route`, `deliver`, or `put`. The service `deliver` transfers the message directly via `informPPC` and `handlePPC` to the accompanied PowerPC. The sub-service `put` sends the message on the appropriate out-going link of the Transputer. The message will

then be received by the incoming device driver service `get` of the neighbouring Transputer. The service `get` makes again use of the same three alternative sub-services `route`, `deliver`, or `put` in order to forward the message. The service `route` routes the message through the network by determing the out-going link or shared memory interface. The services `handleTP` and `get` can only directly use the service `put` when all messages have to use the same out-going device.

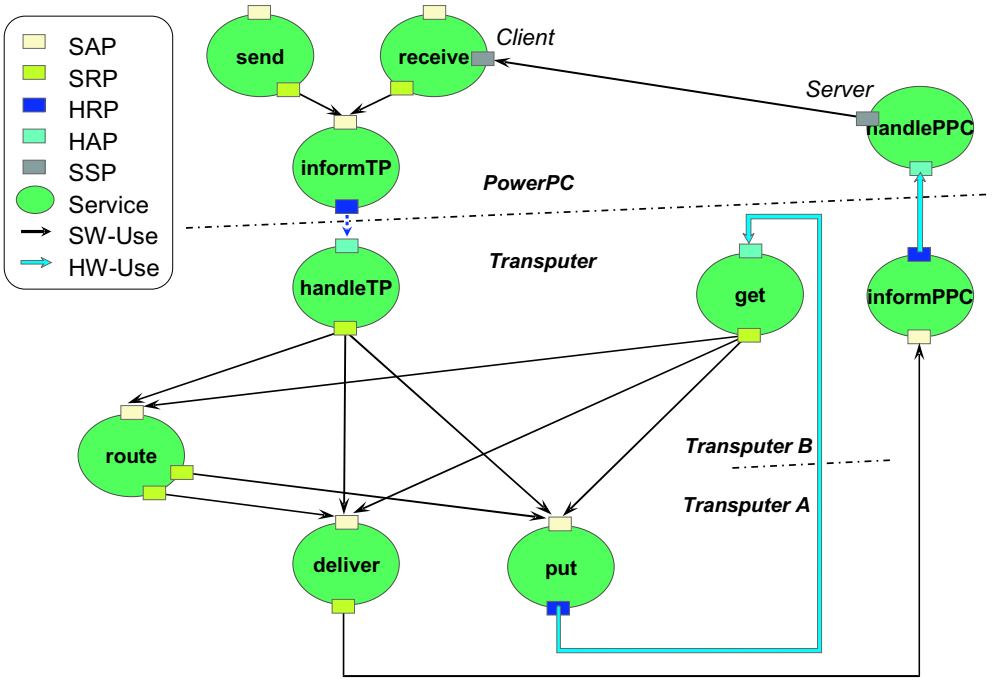

Fig. A.2: Example for an *Universal Service Graph* (USG).

The following text defines the *Universal Resource Service Graph* (URSG), which is partly presented in Figure A.2:

```
// ------------------------------------------------------------------------
// begin of database declarations:
// ------------------------------------------------------------------------
DATABASE

// ------------------------------------------------------------------------
// definition of types for ports, SAPs, SNPs, HNPs and services:
// ------------------------------------------------------------------------
// declaration of ports:

IOPORT(PPC_SM)
END_PORT

IOPORT(TP_SM)
END_PORT

IOPORT(IO1SMP)
END_PORT

IOPORT(IO2SMP)
```

```
END_PORT

IPORT(link_in)
END_PORT

OPORT(link_in)
END_PORT

OPORT(link_out)
END_PORT

IPORT(link_out)
END_PORT

IPORT(InLinkP)
END_PORT

OPORT(OutLinkP)
END_PORT

OPORT(InLinkP)
END_PORT

IPORT(OutLinkP)
END_PORT

// ------------------------------------------------------------------------
// declaration of SAPs:

SAP(ai_send)
END_SAP

SAP(ai_receive)
END_SAP

SAP(ai_informPPC)
END_SAP

SAP(ai_informTP)
END_SAP

SAP(ai_routeA)
END_SAP

SAP(ai_routeB)
END_SAP

SAP(ai_put)
END_SAP

SAP(ai_deliver)
END_SAP

SAP(ai_block)
   BLOCKS
END_SAP

SAP(ai_activate)
   ACTIVATES(receive)
```

```
END_SAP

// ------------------------------------------------------------------------
// declaration of SUPs, SCPs and SNPs:

SNP(ni_send)
   SELECT(ai_informTP)
END_SNP

SNP(ni_receive1)
   SELECT(ai_informTP)
END_SNP

SNP(ni_receive2)
   SELECT(ai_block)
END_SNP

SNP(ni_deliver)
   SELECT(ai_informPPC)
END_SNP

SNP(ni_decide)
   SELECT(ai_routeA)
   SELECT(ai_routeB)
   SELECT(ai_put)
   SELECT(ai_deliver)
   SELECT_MIN(1)
   SELECT_MAX(1)
END_SNP

SNP(ni_routeA1)
   SELECT(ai_put)
END_SNP

SNP(ni_routeA2)
   SELECT(ai_deliver)
   SELECT_MIN(0)
   SELECT_MAX(1)
END_SNP

SNP(ni_routeB)
   SELECT(ai_put)
END_SNP

SNP(ni_handlePPC)
   SELECT(ai_activate)
END_SNP

// ------------------------------------------------------------------------
// declaration of HAPs:

HAP(hi_handleTP)
   USES_IPORT(TP_SM)
END_HAP

HAP(hi_handlePPC)
   USES_IPORT(PPC_SM)
END_HAP
```

```
HAP(hi_get)
   USES_IPORT(link_in)
END_HAP

// ------------------------------------------------------------------------
// declaration of HNPs:

HNP(hi_informPPC)
   USES_OPORT(TP_SM)
   OPORTPREFERS(TP_SM, 0, ni_decide, ai_deliver)
   OPORTPREFERS(TP_SM, 0, ni_decide, ai_routeA)
END_HNP

HNP(hi_informTP)
   USES_OPORT(PPC_SM)
END_HNP

HNP(hi_put)
   USES_OPORT(link_out)
   OPORTPREFERS(link_out, 0, ni_decide, ai_put)
   OPORTPREFERS(link_out, 1, ni_decide, ai_routeA)
   OPORTPREFERS(link_out, 2, ni_decide, ai_routeB)
END_HNP

// ------------------------------------------------------------------------
// declaration of services:

SERVICE(deliver)
   IN(ai_deliver)
   OUT(ni_deliver)
END_SERVICE

SERVICE(send)
   IN(ai_send)
   OUT(ni_send)
   ASYN_CLIENT_OF(deliver)
END_SERVICE

SERVICE(receive)
   IN(ai_receive)
   OUT(ni_receive1)
   OUT(ni_receive2)
   SYNC_CLIENT_OF(deliver)
END_SERVICE

SERVICE(informPPC)
   IN(ai_informPPC)
   NEEDS(hi_informPPC)
END_SERVICE

SERVICE(informTP)
   IN(ai_informTP)
   NEEDS(hi_informTP)
END_SERVICE

SERVICE(get)
   TRIGGEREDBY(hi_get)
   OUT(ni_decide)
END_SERVICE
```

```
SERVICE(put)
   IN(ai_put)
   NEEDS(hi_put)
   INHIBITS_AT_MULTI_NEED(ni_decide, ai_put)
END_SERVICE

SERVICE(routeA)
   COSTS(3)
   IN(ai_routeA)
   OUT(ni_routeA1)
   OUT(ni_routeA2)
END_SERVICE

SERVICE(routeB)
   COSTS(2)
   IN(ai_routeB)
   OUT(ni_routeB)
END_SERVICE

SERVICE(handleTP)
   TRIGGEREDBY(hi_handleTP)
   OUT(ni_decide)
END_SERVICE

SERVICE(handlePPC)
   TRIGGEREDBY(hi_handlePPC)
   OUT(ni_handlePPC)
END_SERVICE

// ------------------------------------------------------------------------
// definition of types for CPUs, devices and media:
// ------------------------------------------------------------------------
// declaration of CPUs:

CPU(Transputer)
   PROVIDES(handleTP)
   PROVIDES(put)
   PROVIDES(get)
   PROVIDES(deliver)
   PROVIDES(routeA)
   PROVIDES(routeB)
   PROVIDES(informPPC)
   HAS_IPORT(link_in)
   HAS_OPORT(link_out)
   HAS_IOPORT(TP_SM)
END_CPU

CPU(PowerPC)
   PROVIDES(send)
   PROVIDES(receive)
   PROVIDES(informTP)
   PROVIDES(handlePPC)
   HAS_IOPORT(PPC_SM)
END_CPU

// ------------------------------------------------------------------------
// declaration of devices:
```

```
DEVICE(PPCMemDev)
   HAS_IOPORT(PPC_SM)
   HAS_IOPORT(IO1SMP)
END_DEVICE

DEVICE(TPMemDev)
   HAS_IOPORT(TP_SM)
   HAS_IOPORT(IO2SMP)
END_DEVICE

DEVICE(LinkInDev)
   HAS_OPORT(link_in)
   HAS_IPORT(InLinkP)
END_DEVICE

DEVICE(LinkOutDev)
   HAS_IPORT(link_out)
   HAS_OPORT(OutLinkP)
END_DEVICE

// ------------------------------------------------------------------------
// declaration of media:

MEDIA(shared_memory)
   HAS_IOPORT(IO1SMP)
   HAS_IOPORT(IO2SMP)
END_MEDIA

MEDIA(link)
   HAS_OPORT(InLinkP)
   HAS_IPORT(OutLinkP)
END_MEDIA

// ------------------------------------------------------------------------
// end of database declarations:
// ------------------------------------------------------------------------
END_DATABASE
```

Requirements Specification

The requirements specification defines two tasks A and B. Task A is placed on the PowerPC 5 and the task B is placed on the PowerPC 3. Only one communication (*ComLine1*) is specified. The task A wants to send a message to the task B (see Figure A.3).

```
# The resources define the tasks, their events,
# and the communication mailboxes:

RESOURCE(Mailbox1)
  DELAY(5)
  HAS(0)
  EVENT(send, ai_send)
  EVENT(receive, ai_receive)
END_RESOURCE

RESOURCE(TaskA)
  IS_TASK(80, 250, 1)
```

```
  EXECUTES(send, Mailbox1, 50) IS SEND(1)
END_RESOURCE

RESOURCE(TaskB)
  IS_TASK(100, 250, 1)
  EXECUTES(receive, Mailbox1, 50) IS RECEIVE(1)
END_RESOURCE

# The place statements map the tasks onto the CPUs
# and specify the source files:

PLACE
  RESOURCE(TaskA)
  CPU(3)
  FILE(mainA.c)
END_PLACE

PLACE
  RESOURCE(TaskB)
  CPU(5)
  FILE(mainB.c)
END_PLACE

# The communication statements connect the appropriate send
# and receive events:

COMLINE(ComLine1)
  SENDTASK(TaskA,1)
  RECVTASK(TaskB,1)
END_COMLINE

END
```

Configuration Result

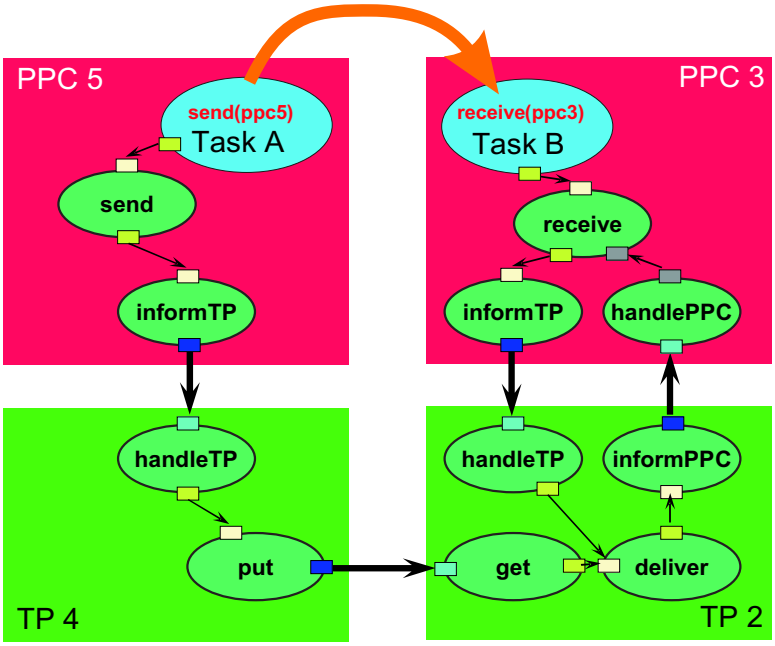

Fig. A.3: Example for the final configuration according to the requirement specification on page 173.

The final configuration, which is depicted in Figure A.3, shows the expected result. The service `receive` is not required for the PowerPC 5 and the service `send` is excluded from the PowerPC 3. The service `handleTP` on Transputer 4 calls directly the service `put`, because all messages –here, in fact, only one– have to be sent directly on the first link. the services `get` and `handleTP` on Transputer 2 use directly the service `deliver` in order to store and forward the message to the accompanied PowerPC. All other processors do not require any service.

APPENDIX B

Example for the Timing Analysis

This chapter contains the source code of the requirement specification for the scheduling example, which had been described on page 138.

Input

```
# ExecutionBlocks ----------------------

EXECUTION-BLOCK(t-dispatch)
  WAIT(OWNRES, 1)
  KILL_SIGNAL(OWNRES, ALL)
  DURATION(30)
  # restart Round-Robin scheduler with its timeslice (340):
  SIGNAL(OWNRES, 1, 340)
END_EXECUTION_BLOCK

EXECUTION_BLOCK(pre-send)
  DURATION(30)
  SIGNAL(OWNRES, 1, TRANSFER)
END_EXECUTION_BLOCK

EXECUTION_BLOCK(post-send)
  DURATION(30)
END_EXECUTION_BLOCK

EXECUTION_BLOCK(pre-receive)
  DURATION(30)
END_EXECUTION_BLOCK

EXECUTION_BLOCK(post-receive)
  DURATION(20)
  # wait on resource, iff waiting then immediately signal scheduler:
  SIGNALWAIT(OWNRES, 1, scheduler(OWNCPU), 1, 0)
END_EXECUTION_BLOCK
```

```
# General Tasks ------------------------

RESOURCE(scheduler)
  IS_TASK(0,340,0)
  CRITICALNESS(NO)
  EVENT(dispatch, t-dispatch)
  HAS(1)        # scheduler has immediately a signal and can run
  UNITS(1)
  EXECUTES(dispatch, scheduler, 0)
END_RESOURCE

RESOURCE(idle)
  IS_TASK(100,100,100)
  CRITICALNESS(NO)
END_RESOURCE

# CPU1 --------------------------------

RESOURCE(cpu1)
  IS_CPU
  HAS(1)        # CPU has immediately a signal
  UNITS(1)
END_RESOURCE

# CPU2 --------------------------------

RESOURCE(cpu2)
  IS_CPU
  HAS(1)
  UNITS(1)
END_RESOURCE

# comLine1 Resource --------------------

RESOURCE(comLine1)
  EVENT(send, pre-send, post-send)
  EVENT(receive, pre-receive, post-receive)
  DELAY(30)    # the time required for the transfer
  UNITS(5)     # e.g. comLine can buffer 5 messages
  HAS(0)       # at begin there is no message
END_RESOURCE

# task on CPU1 ------------------------

EXECUTION_BLOCK(t1-start)
  # wait on resource, iff waiting immediately signal scheduler:
  SIGNALWAIT(OWNRES, 1, scheduler(OWNCPU), 0, 1)
  DURATION(30)
END_EXECUTION_BLOCK

EXECUTION_BLOCK(t1-end)
  DURATION(10)
  SIGNAL(OWNRES, 1, PERIOD)
  SIGNAL(scheduler, 1, PERIOD, NO_KILL)
  SYNC_STARTTIME(PERIOD)
END_EXECUTION_BLOCK

RESOURCE(t1)
```

```
  IS_TASK(80, 340, 1)
  HAS(1)        # task has immediately a signal and can run
  UNITS(1)
  EVENT(start, t1-start)
  EVENT(end, t1-end)
  EXECUTES(start, t1, 0)
  EXECUTES(send, comLine1, 50)
  EXECUTES(end, t1, 80)
END_RESOURCE

# task on CPU2 -------------------------

EXECUTION_BLOCK(t2-start)
  # wait on resource, iff waiting immediately signal scheduler:
  SIGNALWAIT(OWNRES, 1, scheduler(OWNCPU), 0, 1)
  DURATION(20)
END_EXECUTION_BLOCK

EXECUTION_BLOCK(t2-end)
  DURATION(10)
  SIGNAL(OWNRES, 1, PERIOD)
  SIGNAL(scheduler, 1, PERIOD, NO_KILL)
  SYNC_STARTTIME(PERIOD)
END_EXECUTION_BLOCK

RESOURCE(t2)
  IS_TASK(90, 340, 1)
  HAS(1)        # task has immediately a signal and can run
  UNITS(1)
  EVENT(start, t2-start)
  EVENT(end, t2-end)
  EXECUTES(start, t2, 0)
  EXECUTES(receive, comLine1, 20)
  EXECUTES(end, t2, 90)
END_RESOURCE

# place tasks onto cpus ----------------

PLACE
 RESOURCE(t1)
 CPU(cpu1)
END_PLACE

PLACE
 RESOURCE(scheduler)
 CPU(cpu1)
END_PLACE

PLACE
 RESOURCE(idle)
 CPU(cpu1);
END_PLACE

PLACE
 RESOURCE(t2)
 CPU(cpu2)
END_PLACE

PLACE
```

```
 RESOURCE(scheduler)
 CPU(cpu2)
END_PLACE

PLACE
 RESOURCE
 CPU(cpu2)
END_PLACE

PLACE
 RESOURCE(comLine1)
 CPU(GLOBAL)
END_PLACE

# EOF ---------------------------------
```

Output

```
# scheduler, t1 and t2 have immediately signals!

# Schedule of cpu1:
0 EBLOCK t-dispatch   OF RESOURCE scheduler  FROM TASK scheduler;
     CONSUMES (scheduler, 1)
     KILLS (scheduler, 0)
     SIGNALS (scheduler, 1, 530)
30 TRY EBLOCK t-dispatch   OF RESOURCE scheduler  FROM TASK scheduler;
     WAIT (scheduler, 1)
30 EBLOCK t1-start     OF RESOURCE t1         FROM TASK t1;
     CONSUMES (t1, 1)
60 TASK t1;
110 EBLOCK pre-send     OF RESOURCE comLine1   FROM TASK t1;
     SIGNALS (comLine1, 1, 170)
140 EBLOCK post-send    OF RESOURCE comLine1   FROM TASK t1;
170 TASK t1;
200 EBLOCK t1-end       OF RESOURCE t1         FROM TASK t1;
     SIGNALS (t1, 1, 340)
     SIGNALS (scheduler, 1, 340)
     SYNC_STARTTIME (t1, 680)
210 TRY EBLOCK t1-start     OF RESOURCE t1         FROM TASK t1;
     WAIT (t1, 1)
     SIGNALS (scheduler, 1, 210)
210 EBLOCK t-dispatch   OF RESOURCE scheduler  FROM TASK scheduler;
     CONSUMES (scheduler, 1)
     KILLS (scheduler, 1)
     SIGNALS (scheduler, 1, 740)
240 TRY EBLOCK t-dispatch   OF RESOURCE scheduler  FROM TASK scheduler;
     WAIT (scheduler, 1)
240 TASK idle;
# 340 (END OF HYPERPERIOD OF THIS CPU)

# Schedule of cpu2:
0 EBLOCK t-dispatch   OF RESOURCE scheduler  FROM TASK scheduler;
     CONSUMES (scheduler, 1)
     KILLS (scheduler, 0)
     SIGNALS (scheduler, 1, 530)
30 TRY EBLOCK t-dispatch   OF RESOURCE scheduler  FROM TASK scheduler;
     WAIT (scheduler, 1)
```

```
30 EBLOCK t2-start     OF RESOURCE t2         FROM TASK t2;
     CONSUMES (t2, 1)
50 TASK t2;
70 EBLOCK pre-receive  OF RESOURCE comLine1   FROM TASK t2;
90 TRY EBLOCK post-receive OF RESOURCE t2         FROM TASK t2;
     WAIT (comLine1, 1)
     SIGNALS (scheduler, 1)
90 EBLOCK t-dispatch   OF RESOURCE scheduler  FROM TASK scheduler;
     CONSUMES (scheduler, 1)
     KILLS (scheduler, 1)
     SIGNALS (scheduler, 1, 460)
120 TRY EBLOCK t-dispatch   OF RESOURCE scheduler  FROM TASK scheduler;
     WAIT (scheduler, 1)
120 TASK idle;
170 EBLOCK post-receive OF RESOURCE t2         FROM TASK t2;
     CONSUMES(comLine1, 1);
190 TASK t2;
260 EBLOCK t2-end       OF RESOURCE t2         FROM TASK t2;
     SIGNALS (t2, 1, 340)
     SIGNALS (scheduler, 1, 340)
     SYNC_STARTTIME (t2, 680)
270 EBLOCK t-dispatch   OF RESOURCE scheduler  FROM TASK scheduler;
     CONSUMES (scheduler, 1)
     KILLS (scheduler, 1)
     SIGNALS (scheduler, 1, 640)
300 TRY EBLOCK t-dispatch   OF RESOURCE scheduler  FROM TASK scheduler;
     WAIT (scheduler, 1)
300 TASK idle;
# 340 (END OF HYPERPERIOD OF THIS CPU)
```

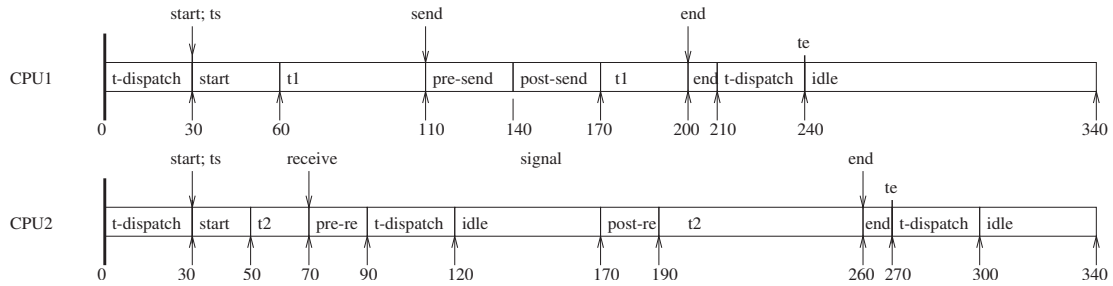

Fig. B.1: Visualisation of the schedules from the output of the timing analysis on page 180f.

APPENDIX C

Language Descriptions

In the next sections we will define, which item types can be declared per database, respectively graph, which order should be used, and which property statements are allowed for a specific item type. Moreover, the semantical meaning of the declarations will be explained.

In the given examples below, the statements, which can be multiply defined, are surrounded by "`[...]*`". The other property statements must occur only once per database item.

Comments are allowed in the form that any text after the character '#' followed by a white space is ignored until the end of the line.

C.1 General Domain Knowledge

The general domain knowledge for the configuration process is defined in a database, in which all configuration items are stored. It defines a kind of knowledge base, because all properties of the configuration items, like their dependencies, the alternative selections and the constraints, are described herein. The configurator TGEN reads this database from a text file. Comments are allowed in the form that any text after the character '#' followed by a white space is ignored until the end of the line. The definition of the database starts with the keyword `DATABASE` and ends with the keyword `END_DATABASE`. Every line before and after this keywords are ignored. Inside the database definition all `IPORTs`, `OPORTs`, `IOPORTs`, `SAPs`, `SRPs`, `HAPs`, `HRPs`, `PRIMITIVEs`, `SERVICEs`, `EXECUTION_BLOCKs`, `CPUs`, `DEVICEs`, and `MEDIAs` have to be declared. Thereby, exactly this order have to be used, because, for instance, before an SAP can be referenced in a *service*, it must have been declared.

The database should start with the declaration of ports. The port declarations have the following form:

```
IPORT(<iport-type-name>)
  VISIBILITY(<num>,<num>,<num>)
END_PORT

OPORT(<oport-type-name>)
  VISIBILITY(<num>,<num>,<num>)
END_PORT

IOPORT(<ioport-type-name>)
  VISIBILITY(<num>,<num>,<num>)
END_PORT
```

The VISIBILITY statement, which is used for nearly every item of the database that is possibly the source or destination of an edge (dependency relation), defines whether all the assigned edges are visible at different levels inside the editor. This statement is only managed by the editor tool. The ports have no other property. The port statements only declare the appropriate type names as an in-port, an out-port, or an in/out-port.

After the port declarations the declaration of SAP types follow:

```
SAP(<sap-type-name>)
  VISIBILITY(<num>,<num>,<num>)
  PREFIX("<string>")
  SUFFIX("<string>")
  BLOCKS
  ACTIVATES(<service-type-name>)
  COSTS(<int>)
  SELECTED_BY_MAX(<num>)
  [ SAPFORCES(<srp-type-name>,<sap-type-name>) ]*
  [ SAPFAVOURS(<srp-type-name>,<sap-type-name>) ]*
  [ SAPINHIBITS(<srp-type-name>,<sap-type-name>) ]*
END_SAP
```

The strings that are defined in the PREFIX and SUFFIX statements will be used, when the description of a final configuration is generated and this SAP is referred. The keywords BLOCKS and ACTIVATES show, that the service, which has this SAP assigned, works as a client or server service, respectively (see "Considering Communications" on page 119). The keyword COSTS define the costs of this SAP for the configuration (see "Cost Function" on page 117). The keyword SELECTED_BY_MAX defines that this SAP can only be accessed from a maximum of <num> SRPs. The keywords SAPFORCES, SAPFAVOURS and SAPINHIBITS define constraints for the appropriate SRP to SAP alternatives (see Section $6.3.4_{110}$). More than one constraint of each type can be declared.

After the SAP declarations the declaration of SRP types follow:

```
SRP | SNP | SUP | SCP (<srp-type-name>)
  VISIBILITY(<num>,<num>,<num>)
  SELECT_MIN(<num>)
  SELECT_MAX(<num>)
  [ SELECT(<sap-type-name>) ]*
END_SRP | END_SNP | END_SUP | END_SCP
```

Instead of the keyword SRP also the keywords SNP, SUP and SCP can be used. They will be handled identically by the configurator TGEN. With the different keywords the expert can distinguish different types of service dependencies. In the case of the description of an object-oriented model (as we have in the case of DREAMS) this can be, for example,

the inheritance (SNP), the member (SCP) and the calling (or usage) dependency (SUP) between the *service* that contains this SRP and the *service* that contains the referenced SAP. But remember, the configurator TGEN handles them identically. The SELECT statement refers to an SAP. Thereby, an edge between the *service* that contains this SRP and the *service* that contains the referenced SAP is defined. This edge represents the directed dependency between these services. Because the dependency is defined between the SRP and SAP of a service, the same dependency relation can be defined for another service, when it simply contains also this SRP. If more than one SELECT statement is given, then this defines an OR-group of alternative dependencies. If a *service* wants to define an AND-group of dependencies, then it must contain one SRP per dependency. Thereby, an AND/OR-relationship can be defined. All required SRPs (AND-group) of a *service* defines possibly an OR-group. By the optional keywords SELECT_MIN and SELECT_MAX the minimum and maximum number of services, which must be selected from this OR-group, can be specified. By default both values are assumed to be 1.

After the SRP declarations the declaration of HAP types follow:

```
HAP(<hap-type-name>)
  VISIBILITY(<num>,<num>,<num>)
  [ USES_IPORT(<iport-type-name>) ]*
  [ USES_IPORT(<ioport-type-name>) ]*
  [ IPORTPREFERS(<iport-type-name>,<num>,<srp-type-name>,<sap-type-name>) ]*
  [ IPORTINHIBITS(<iport-type-name>,<num>,<srp-type-name>,<sap-type-name>) ]*
  [ IPORTFORCES(<iport-type-name>,<num>,<srp-type-name>,<sap-type-name>) ]*
  [ IPORTPREFERS(<ioport-type-name>,<num>,<srp-type-name>,<sap-type-name>) ]*
  [ IPORTINHIBITS(<ioport-type-name>,<num>,<srp-type-name>,<sap-type-name>) ]*
  [ IPORTFORCES(<ioport-type-name>,<num>,<srp-type-name>,<sap-type-name>) ]*
END_HAP
```

The HAP of a *service* is used in order to indicate that the *service* requires a hardware device. The software *service* can bee seen as a device driver. Additionally, it is indicated that the direction of the data flow is from the software to the device. Devices have assigned ports. The HAP can be connected to this ports by the use of the USES_IPORT statement. Thereby, the HAP can be connected to an in-port and to an in/out-port. The IPORTPREFERS, IPORTFORCES and IPORTINHIBITS statements define constraints, which are guarded by this HAP.

After the HAP declarations the declaration of HRP types follow:

```
HRP(<hrp-type-name>)
  VISIBILITY(<num>,<num>,<num>)
  [ USES_OPORT(<oport-type-name>) ]*
  [ USES_OPORT(<ioport-type-name>) ]*
  [ OPORTPREFERS(<oport-type-name>,<num>,<srp-type-name>,<sap-type-name>) ]*
  [ OPORTINHIBITS(<oport-type-name>,<num>,<srp-type-name>,<sap-type-name>) ]*
  [ OPORTFORCES(<oport-type-name>,<num>,<srp-type-name>,<sap-type-name>) ]*
  [ OPORTPREFERS(<ioport-type-name>,<num>,<srp-type-name>,<sap-type-name>) ]*
  [ OPORTINHIBITS(<ioport-type-name>,<num>,<srp-type-name>,<sap-type-name>) ]*
  [ OPORTFORCES(<ioport-type-name>,<num>,<srp-type-name>,<sap-type-name>) ]*
END_HRP
```

The HRP is defined similar to the declaration of a HAP. The HRP of a *service* indicates, that this *service* also requires for a hardware device. But the data flow is from the device to the *service*. The *service* can also be seen as a device driver; but more specifically it

receives the data by the device or it is activated by the device, like an interrupt service routine. For this reason, the HRP can be connected to an out-port or to an in/out-port by the USES_OPORT statement. The OPORTPREFERS, OPORTFORCES and OPORTINHIBITS statements define again constraints, which are guarded by this HRP.

After the HRP declarations the declaration of *user primitives* follow:

```
PRIMITIVE(<primitive-type-name>)
  VISIBILITY(<num>,<num>,<num>)
  POSITION(<x>,<y>)
  OUT(<srp-type-name>) [ FIXED_AT(<x>,<y>) ]
END_PRIMITIVE
```

A *user primitive* can be requested by an application in the requirement specification. The keyword POSITION with its two parameters x and y will be ignored by the configurator TGEN. It is managed by the editor tool and indicates the position of this node in Cartesian coordinates inside the visualised *Universal Resource Service Graph* (URSG). The OUT statement defines an edge to an SRP. Only one of this statements per *user primitive* is allowed. The OUT statement can optionally be extended by a FIXED_AT statement. This is also ignored by TGEN. It indicates to the editor tool TEDIT, at which position this SRP have to be placed for this *user primitive*.

After the *user primitive* declarations the declaration of *services* follow:

```
SERVICE(<service-type-name>)
  VISIBILITY(<num>,<num>,<num>)
  POSITION(<x>,<y>)
  COSTS(<int>)
  REQUIRES(<cpu-type-name>)
  PRIORITY(<num>)
  PRIORITY2(<num>)
  MAXPRIORITY(<num>)
  ORDER(<num>)
  PREFIX("<string>")
  SUFFIX("<string>")
  PATH(<path-name>)
  FILE(<file-name>)
  [ IN(<sap-type-name>) [ FIXED_AT(<x>,<y>) ] ]*
  [ OUT(<srp-type-name>) [ FIXED_AT(<x>,<y>) ] ]*
  [ NEEDS(<hap-type-name>) [ FIXED_AT(<x>,<y>) ] ]*
  [ TRIGGEREDBY(<hrp-type-name>) [ FIXED_AT(<x>,<y>) ] ]*
  [ ASYN_CLIENT_OF(<service-type-name>) ]*
  [ SYNC_CLIENT_OF(<service-type-name>) ]*
  [ INHIBITS(<srp-type-name>,<sap-type-name>) ]*
  [ FAVOURS(<srp-type-name>,<sap-type-name>) ]*
  [ FORCES(<srp-type-name>,<sap-type-name>) ]*
  [ FORCEHWUSE(<srp-type-name>,<sap-type-name>) ]*
  [ INHIBITS_AT_MULTI_NEED(<srp-type-name>,<sap-type-name>) ]*
  [ SAY("<string>") ]*
  [ NEGSAY("<string>") ]*
END_SERVICE
```

The *service* is the central type definition. It defines an item that is subject to the configuration. The *services* can be included or excluded from a final configuration. The configurator determines, which *service* is part of the configuration and which is not part of the configuration. Remark, that each CPU will have its own configuration. For this

reason, the service can have assigned costs, a primary and a secondary priority, and a maximum priority, that can be reached by preferring in an OR-group (see "Configuring Means Choosing of Alternatives" on page 118). The ORDER, PREFIX, SUFFIX, PATH, FILE, SAY and NEGSAY statements define strings that can be referred during the generation of this service for a final configuration. When the *service* is selected to be part of the configuration the SAY statement will be evaluated and printed. If the *service* is not part of the configuration, then the NEGSAY statement will be evaluated and printed. The *services* are sorted after the value of the ORDER statement and then they are printed into the configuration description.

The IN, OUT, NEEDS and TRIGGEREDBY statements indicate, which SAPs, SRPs, HAPs and HRPs are connected to this *service*. The statements ASYN_CLIENT_OF and SYNC_-CLIENT_OF define that this *service* is a client service for the server service, which is given in the argument. There, synchronous means that the client will be blocked and asynchronous means that the client will not be blocked. The synchronous client will require for a service that will have assigned an SRP, which requests for an SAP with an BLOCK statement. The server service will request for a service that will have assigned an SRP, which requests for an SAP with an ACTIVATE statement. By this way, the server service activates the synchronous client service. The asynchronous client sends messages to the server service, whereas the synchronous client receives messages from the server service. In Appendix A a configuration example with such a client/server dependency is described.

The FAVOURS, FORCES and INHIBITS statements define again constraints. The FORCE-HWUSE statement identifies the specified choice element (SAP) to be chosen in an OR-group, if the service, which is assigned to that choice element, requests a specific CPU type or a specific architecture that is provided by the CPU for which it should be selected. The REQUIRES statement defines that this service possibly requires a specific CPU type to be instantiated on. This statement is referenced by the FORCEHWUSE statement. The INHIBITS_AT_MULTI_NEED statement prohibits the selection of the specified choice element, if this service is used by more than one communication or task.

After the *service* declarations the declaration of *execution blocks* follow:

```
EXECUTION_BLOCK(ebName)
  DISABLE(resourceName)
  ENABLE(resourcename)
  PRIORITY(p)
  DURATION(l)
  [ WAIT(resourceName, n) ]* # resourceName can be keyword OWNRES
  [ KILL_SIGNAL(resourceName, n) ]*
    # resourceName can be keyword OWNRES, n can be keyword ALL
  [ SIGNAL(resourceName[(CpuResourceName)], n, t, [NOKILL] ) ]*
    # resourceName can be keyword OWNRES, cpuResourceName can be keyword OWNCPU,
    # t can be keyword PERIOD or TRANSFER
  [ SIGNALWAIT(resourceName, n, resourceName[(CpuResourceName)], sn, st) ]*
    # wResourceName can be keyword OWNRES, sCpuName can be keyword OWNCPU,
    # st can be keyword PERIOD or TRANSFER
  SYNC_STARTTIME(t) # t can be keyword PERIOD
END_EXECUTION_BLOCK
```

The EXECUTION_BLOCK statement defines the timing behaviour of a task and mainly of a system call to the operating system. These statements are only considered by the timing

analysis. Its properties and semantic are described in Section 6.8_{133}. The definition of an execution block is also allowed to occur in the requirements specification.

Up to now all items of the *Universal Service Graph* (USG) are defined. Now, we will describe the declarations for the resource types, that can be used in the hardware and topology description, which is the *Resource Graph* (RG) (see Section C.2). These resource types are also part of the knowledge base and have connections to the services. They will have attached ports that are used in the HAPs and HRPs of the services. A resource can be of the type CPU, DEVICE or MEDIA, which is indicated by the use of the appropriate keyword. These resource type declarations extend the previously defined USG to the *Universal Resource Service Graph* (URSG). The resource type statements look like the following:

```
CPU(<cpu-type-name>)
  VISIBILITY(<num>,<num>,<num>)
  POSITION(<x>,<y>)
  ARCHITECTURE(<architecture-type-name>)
  [ HAS_IPORT(<iport-type-name>) [ FIXED_AT(<x>,<y>) ] ]*
  [ HAS_OPORT(<oport-type-name>) [ FIXED_AT(<x>,<y>) ] ]*
  [ HAS_IOPORT(<ioport-type-name>) [ FIXED_AT(<x>,<y>) ] ]*
  [ PROVIDES(<service-type-name>) ]*
END_CPU

DEVICE(<device-type-name>)
  VISIBILITY(<num>,<num>,<num>)
  POSITION(<x>,<y>)
  [ HAS_IPORT(<iport-type-name>) [ FIXED_AT(<x>,<y>) ] ]*
  [ HAS_OPORT(<oport-type-name>) [ FIXED_AT(<x>,<y>) ] ]*
  [ HAS_IOPORT(<ioport-type-name>) [ FIXED_AT(<x>,<y>) ] ]*
END_DEVICE

MEDIA(<medium-type-name>)
  VISIBILITY(<num>,<num>,<num>)
  POSITION(<x>,<y>)
  [ HAS_IPORT(<iport-type-name>) [ FIXED_AT(<x>,<y>) ] ]*
  [ HAS_OPORT(<oport-type-name>) [ FIXED_AT(<x>,<y>) ] ]*
  [ HAS_IOPORT(<ioport-type-name>) [ FIXED_AT(<x>,<y>) ] ]*
END_MEDIA
```

The PROVIDES statement of the CPU declaration enumerates all *services* that can be instantiated for this CPU type. A service, which is not mentioned, will never be selected for a configuration for this CPU type. The ARCHITECTURE statement of the CPU declaration can enumerate some architecture names, which can be referenced by a REQUIRES constraint of a *service*. Remember, that the REQUIRES constraint can also directly reference a CPU type. Additionally remark, that the REQUIRES constraint is only evaluated, if the FORCEHWUSE constraints is activated.

After all resources are declared the specification of the knowledge base must be terminated by the use of the keyword END_DATABASE.

C.2 Hardware and Topology

In the hardware and topology description, which is the *Resource Graph* (RG), only two statements are allowed. This is the HARDWARE statement and the TOPOLOGY statement.

They also have to be placed in the given order. The HARDWARE statement, which must come first, has the following form:

```
HARDWARE
  [ DEF_CPU(<cpu-name>,<cpu-num>,<domain-num>,<cpu-type-name>) ]*
  [ SET_VAR_CPU(<cpu-name>,<variable-name>,"<string>") ]*
  [ OUTPUT_CPU(<cpu-name>,<file-name>) ]*
  [ MKFILE_CPU(<cpu-name>,<file-name>) ]*
  [ DEF_MEDIA(<medium-name>,<medium-type-name>) ]*
  [ DEF_DEVICE(<device-name>,<device-type-name>) ]*
END_HARDWARE
```

Inside the HARDWARE statement concrete instances of CPUs, DEVICEs and MEDIAs are created. Therefore, Each instance have an unique identifier as its name and is of a specific type, which must have been declared in the general domain knowledge of the *Universal Resource Service Graph* (URSG). A CPU have additionally assigned a unique CPU number and a domain number, which defines the address of the CPU in the network. To each CPU there can be assigned some variables of the type string, which will be printed into a special output file per CPU. The name of this output file and the name of the makefile for this CPU is defined in two extra statements.

After the HARDWARE statement the TOPOLOGY statement follows:

```
TOPOLOGY
  TOPOLOGY_FILE(<file-name>)
  [ CONNECT_CPU_DEV(<cpu-name>,<device-name>,<port-type-name>,<num>) ]*
  [ CONNECT_DEV_MED(<device-name>,<medium-name>,<port-type-name>) ]*
  [ SAYONCE("<string>") ]*
  [ SAY("<string>") ]*
END_TOPOLOGY
```

The TOPOLOGY statement mainly defines connections between the CPUs and the devices and between the devices and the media. For each connection a port type name must be given. A port of exactly this name must be present in both connected resources. That means that only those ports can be connected, which have the same name! Remark, that the direction of the connection is defined by the ports and not by this statement. Additionally, a filename can be specified. In this file the topology and other information about the communications will be printed. Additionally, a topology file will be printed for each CPU. From this files other tools can generate the routing tables for each CPU. The SAY and SAYONCE statements contain the strings, which will be printed into these files. The strings of the SAY statement will be printed in all files, while the SAYONCE statements will be printed only in the files per CPU. Inside the strings special identifiers are recognised. All these special identifiers must start with the '%' character. These identifiers are replaced in the output by ID numbers or lists of ID numbers of the CPUs or domains. The following identifiers are allowed:

- %CPUID()
- %DOMAIN()
- %LIST(DOMAINS)
- %LIST(CPUS)
- %LIST(CPU_DOMAINS)
- %LIST(LOCAL_CPUS)
- %LIST(LINKS_TO_CPU)
- %LIST(LINKS_ADDR)
- %LIST(LINKS_NCPU)
- %LIST(LINKS_NLINK)
- %LIST(GATES_OUT)

- %LIST(GATES_TO_CPU)
- %LIST(GATES_TO_LINK)

C.3 Requirements Specification

The requirements specification allows only five different statements. With these statements the tasks, their system call events, their communications, and their placement onto the CPUs can be described.

The PACKET statement defines the data structure of a message that will be used by a communication connection. It is assumed that all messages, which are sent via one specific communication connection between two tasks, have the same data structure.

```
PACKET(name)
  [ INT8(name) ]*
  [ INT16(name) ]*
  [ INT32(name) ]*
  [ INT64(name) ]*
  [ UINT8(name) ]*
  [ UINT16(name) ]*
  [ UINT32(name) ]*
  [ UINT64(name) ]*
  [ FLOAT32(name) ]*
  [ FLOAT64(name) ]*
  [ ARRAY8(name, size) ]*
  [ ARRAY16(name, size) ]*
  [ ARRAY32(name, size) ]*
  [ ARRAY64(name, size) ]*
  [ UARRAY8(name, size) ]*
  [ UARRAY16(name, size) ]*
  [ UARRAY32(name, size) ]*
  [ UARRAY64(name, size) ]*
  [ FARRAY32(name, size) ]*
  [ FARRAY64(name, size) ]*
  [ CSTRING(name, maxlen) ]*
  [ PSTRING16(name, maxlen) ]*
  [ PSTRING32(name, maxlen) ]*
  [ PACKET(name) ]*
END_PACKET
```

The PACKET declaration is the definition of a compount type. There, basic data types, like integers or unsigned intergers and floating point types, which all can be of different bit width, can be used. Also arrays of these basic data types can be defined. Additionally, strings can be defined, which have a maximum length and consist of a sequence of 8 Bit unsigned integers (chars). The CSTRING is null terminated, whereas the PSTRINGs are preceeded by an unsigned integer value, which represents their length. Inside the packet it is assumed that the string consumes maxlen size. Another packet declaration can be specified to be a sub-type of the actual packet declaration.

The PACKET declaration have two major reasons: (1) the maximzum size of a message that is sent is specified, and (2) an algorithm can be given these information as input in order to change automatically the endianess of the message. Thereby, the tasks of a communication can be placed on processors with different endianess (low/high-byte order).

The most important definition in the requirement specification is the declaration of a resource:

```
RESOURCE(resourceName)
  IS_TASK(runtime, period, priority) | IS_CPU
  CRITICALNESS(NO | SOFT | FIRM | HARD)
  DELAY(delay)
  UNITS(items)
  HAS(items)
  [ EVENT(eventName, [ ebName ]+ ) [ IS SEND(num) | RECEIVE(num) ] ]*
  [ EXECUTES(eventName, resourceName, time [ , delay [ , items ]] )
    [ IS SEND(num) | RECEIVE(num) ] ]*
    # delay and items can be keyword DEFAULT
END_RESOURCE
```

A resource is subject to the `signal()` and `wait()` procedures of the behaviour description of an execution block for the timing analysis. With the property statements `IS_TASK` or `IS_CPU` the resource can be qualified to be a task or a CPU. Only one of these statements must be used and they are optional. Each resource has a counter for items and it works like a counting semaphore. The items which are initially present, are specified with the `HAS` property statement. Each `signal()` and `wait()`, which is executed on this resource, will increment or decrement these counter accordingly. The `DELAY` property specifies that the increment will be delayed for a specific time. Thus, between the production and a possible consume of the items must be at least $delay$ time units. If the *delay* is not given for a resource, then it will be assumed to be zero. Each resource must define some events, that are regarded as system calls. Each of these events is attached a list of execution blocks. A task or any other execution block can execute these events. An execution block can emit a `signal()` to any other resource or can execute a `wait()` for a specified number of items on any resource. With these declarations the behaviour in form of sequentially executing atomic blocks with an signal/wait mechanism can be specified (see Section 6.8$_{133}$).

When the resource is specified to be a task, then the `EXECUTES` properties are allowed to be specified. These properties of a task define when a task calss which system call. In fact, the system call must be the name of an event, which is executed on a specific resource. Additionally, the name of the event is assumed to be a primitive of the *Universal Service Graph* (USG), which is required by the task.

The communication between two tasks has to be modelled in the following way: The sender task has to call a `SEND` event. The receiver task has to call a `RECEIVE` event. Both Events are marked by specific keywords and are identified by a number. For the routing a connection, respectively an in/out-dependency between two tasks will be defined by a `COMLINE` declaration:

```
COMLINE(comName)
  SENDTASK(name1,num1)
  RECVTASK(name2,num2)
  PACKET(name3)
END_COMLINE
```

This declaration defines that task $name_1$ sends the message $name_3$ to the receiving task $name_2$. Hereby, the num_1^{th} send event of the sender task and the num_2^{th} receive event of the receiver task will be used. The period of this connection or the minimal message interarrival time is specified by the minimum of the sender's and receiver's period.

Last but not least, for the routing the placement of the tasks onto the CPUs must be specified. This is done by a PLACE declaration:

```
PLACE
  RESOURCE(resourceName)
  CPU(CpuResourceName)
  [ FILE(filename) ]*
END_PLACE
```

Each task must be placed on a CPU. Optionally, some file names can be specified, which identify the source files that define the task's source code. These will be used for the generation of the Makefiles for each CPU.

Index

D

E

F

G

H

I

J

K

L

M

O

P

Q

R

S

T

U

W

X

Bibliography

[1] P. Altenbernd. CHaRy: The C-LAB Hard Real-Time System to Support Mechatronical Design. In *Int. Conf. and Workshop on Engineering of Computer Based Systems (ECBS)*, 1997. (page 12, 133)

[2] P. Altenbernd, C. Böke, G. D. Castillo, C. Ditze, E. Erpenbach, U. Glässer, W. Hardt, B. Kleinjohann, G. Lehrenfeld, F. J. Rammig, C. Rust, F. Stappert, J. Stroop, and J. Tacken. Paradise: Design Environment for Parallel & Distributed, Embedded Real-Time Systems. In F. J. Rammig, editor, *Distributed and Parallel Embedded Systems*, Proc. of the 1st International IFIP WG 10.3/WG 10.5 Workshop on Distributed and Parallel Embedded Systems (DIPES), Paderborn University, Germany, October 5-6, 1998, pages 181–190. Kluwer Academic Publishers, Norwell, Massachusetts, number 25 of IFIP edition, 1999. (page 161, 162)

[3] B. Appleton. Assembling Configuration Management Environments Project. WWW pages at http://www.enteract.com/∼bradapp/acme/, 1998. (page 24)

[4] N. C. Audsley, A. Burns, M. F. Richardson, K. Tindell, and A. J. Wellings. Applying New Scheduling Theory to Static Priority Preemptive Scheduling. *Software Engineering Journal*, 8(5):284–292, September 1993. (page 73)

[5] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *Workshop on Real-Time Operating Systems*. IEEE, 1992. (page 73)

[6] W. Baginsky, H. Endres, G. Geissing, and L. Philipp. Basic Architectural Features of Configuration Expert Systems for Automation Engineering. In *Proc. of IEEE Conference "International Workshop on Artificial Intelligence for Industrial Applications"*, Hittachi City, 1988. Also TEX-K Report No. 11, Energy and Automation Group, Siemens AG, ESTE 12, Erlangen, Germany. (page 42)

[7] V. E. Barker, D. E. O'Connor, J. Bachant, and E. Soloway. Expert systems for configuration at Digital: XCON and beyond. *Communications of the ACM*, 32(3):298–318, March 1989. ACM Press, New York, NY, USA, ISSN:0001-0782. (page 41)

[8] S. K. Baruah, L. E. Rosier, and R. R. Howell. Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic, Real-Time Tasks on One Processor. *Journal of Real-Time Systems*, 2, 1990. (page 73)

[9] L. Baum, M. Becker, L. Geyer, G. Molter, and P. Sturm. Driving the Composition of Runtime Platforms By Architectural Knowledge. In *Proc. of the 8th ACM SIGOPS European Workshop: Support for Composing Distributed Applications*, Sintra, Portugal, September 1998. (page 25)

[10] L. Baum, L. Geyer, G. Molter, S. Rothkugel, and P. Sturm. Architecture-Centric Software Development Based on Extended Design Spaces. In *Proc. of the ARES Second Int'l Workshop on Development and Evolution of Software Architectures for Product Families*, Las Palmas de Gran Canaria, Spain, Feb. 1998. (page 25)

[11] R. Berg, J. Cordsen, J. Nolte, B. Oestmann, M. Sander, H. Schmidt, F. Schön, and W. Schröder-Preikschat. The PEACE Family of Distributed Operating Systems. Technical report, GMD-FIRST, 1991. 44 pages. (page 51)

[12] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992. (page 3)

[13] H. K. Büning, D. Curatolo, M. Hoffmann, R. Lemmen, M. Suermann, and B. Stein. ARTDECO - Entwurfsunterstützung in der Hydraulik. *KI – Künstliche Intelligenz: Forschung, Entwicklung, Erfahrungen*, 9(5):49–55, September 1995. (page 44)

[14] C. Böke. Software Synthesis of Real-Time Communication System Code for Distributed Embedded Applications. In *Proc. of the 6th Annual Australasian Conf. on Parallel and Real-Time Systems (PART)*, Melbourne, Australia, December 1999. IFIP, IEEE. (page 52, 162)

[15] C. Böke. Combining Two Customization Approaches: Extending the Customization Tool TEReCS for Software Synthesis of Real-Time Execution Platforms. In *Proc. of the Workshop on Architectures of Embedded Systems (AES)*, Karlsruhe, Germany, January 2000. (page 14, 52, 162)

[16] C. Böke, C. Ditze, H. J. Eickerling, U. Glässer, B. Kleinjohann, F. J. Rammig, and W. Thronike. Software IP in Embedded Systems. In *Proc. of the Forum on Design Languages (FDL)*, Lyon, France, September 1999. (page 52, 162)

[17] C. Böke, M. Götz, T. Heimfarth, D. El-Kebbe, F. J. Rammig, and S. Rips. (Re-)Configurable Real-time Operating Systems and their Applications. In *Proc. of WORDS 2003*. IEEE, January 2003. (page 162)

[18] C. Böke, M. Hübel, F. J. Rammig, and C. Rust. Zero-Overhead Pr/T-Net Execution. In *Proc. of the SCS International European Simulation Multi-Conference (ESM)*, Gent, Belgium, May 23-26 2000. (page 161, 162)

[19] C. Böke and F. J. Rammig. Entwurf konfigurierbarer, echtzeitfähiger Kommunikationssysteme. In journal *"Informationstechnik und Technische Informatik" (it+ti)*, volume 2. R. Oldenbourg Verlag, Germany, 1999. (in German). (page 25, 52, 162)

[20] P. Bratley, M. Florian, and P. Robillard. Scheduling with Earliest Start and Due Date Constraints. *Naval Research Logistics Quarterly*, 18(4), 1971. (page 70)

[21] P. Brinch Hansen. *Operating System Principles*. Prentice-Hall, 1973. (page 85)

[22] H. K. Büning and B. Stein. Supporting the Configuration of Technical Systems. In M. Hiller and B. Fink, editors, *Second Conference on Mechatronics and Robotics*, pages 417–432. IMECH, Mechatronics Institute, Moers, Germany, July 1993. (page 44)

[23] A. Burns. Preemptive Priority based Scheduling: An Appropriate engineering approach. Technical report, Real-time Systems Research Group, Dept. of Comp. Sc., University of York, UK, 1993. (page 105)

[24] C. Burrows. Configuration Management: Coming of Age in the Year 2000. In *CrossTalk magazine*, pages 12–16. U.S. Air Force's Software Technology Support Center (STSC), March 1999. http://www.stsc.hill.af.mil/CrossTalk/. (page 23)

[25] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. The Kluwer international series in engineering and computer science. Kluwer Academic Publishers, Boston Dordrecht London, real-time systems edition, 1997. (page 62, 63, 68, 69, 71, 72, 73, 74, 75, 76, 86, 87)

[26] R. Campbell, G. Johnston, and V. Russo. Choices — Class Hierarchical Open Interface for Custom Embedded Systems. *ACM Operating Systems Review*, 21(3):9–17, July 1987. (page 50)

[27] H. Chetto, M. Silly, and T. Bouchentouf. Dynamic Scheduling of Real-Time Tasks under Precedence Constraints. *Journal of Real-Time Systems*, 2, 1990. (page 70)

[28] R. P. Chivukula, C. Böke, and F. J. Rammig. Customizing the Configuration Process of an Operating System Using Hierarchy and Clustering. In *Proc. of the* 5^{th} *IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, pages 280–287, Crystal City, VA, USA, 29 April – 1 May 2002. IFIP WG 10.5. ISBN 0-7695-1558-4. (page 162)

[29] R. Conradi and B. Westfechtel. Configuring Versioned Software Products. In *Software Configuration Management, ICSE'96 SCM-6 Workshop*, volume LNCS 1167, pages 88–109. Springer, 1996. (page 24)

[30] R. Conradi and B. Westfechtel. Version models for software configuration management. Technical Report AIB 96-10, RWTH Aachen, Germany, October 1996. ftp://ftp-i3.informatik.rwth-aachen.de/pub/reports/CW96a.ps.gz. (page 24)

[31] M. Cornero, F. Thoen, G. Goossens, and F. Curatelli. Software Synthesis for Real-Time Information Processing Systems. In P. Marwedel and G. Goossens, editors, *Code Generation for Embedded Processors*, Proc. of the 1st Workshop on Code Generation for Embedded Processors, Dagstuhl (Germany). Kluwer Academic Publishers, Boston, February 1995. http://146.103.254.1/vsdm/projects/sw_synthesis/publication.html. (page 25)

[32] R. Cunis, H. Bode, A. Günter, H. Peters, and I. Syska. Wissensrepräsentation in PLAKON. Tex-k report no. 10, Fachbereich Informatik, Universität Hamburg, Germany, 1988. (page 43)

[33] R. Cunis, A. Günter, I. Syska, H. Bode, and H. Peters. PLAKON – Ein System zur konstruktion in technischen Domänen. In *Proc. of KOMMTECH-88*, Germany, 1988. Also TEX-K Report No. 13, Fachbereich Informatik, Universität Hamburg. (page 43)

[34] R. Cunis, A. Günter, and H. Strecker, editors. *Das* PLAKON*-Buch*. Informatik Fachbericht 266. Springer-Verlag, 1991. (page 42)

[35] Cygnus. eCos – Embedded Cygnus operating system[1]. Technical white paper, Cygnus, 1999. (page 49)

[36] A. Damm, J. Reisinger, W. Schwabl, and H. Kopetz. The real-time operating system of MARS. *ACM Operating Systems Review*, 23(3):141–157, July 1989. (page 84)

[37] M. L. Dertouzos. Control Robotics: The Procedural Control of Physical Processes. *Information Processing*, 74, 1974. (page 69)

[38] E. W. Dijkstra. Cooperating Sequential Processes. In F. Genuys, editor, Academic Press, New York, 1968. (page 85)

[39] C. Ditze. DREAMS – Concepts of a Distributed Real-Time Management System. In *Proc. of the 1995 IFIP/IFAC Workshop on Real-Time Programming (WRTP)*, 1995. (Another copy with quite identical contents appeared in journal *Control Engineering Practice*, Vol. 4 No. 10, 1996.). (page 51)

[40] C. Ditze. A Customizable Library to support Software Synthesis for Embedded Applications and Micro-Kernel Systems. In *Proc. of the Eighth ACM SIGOPS European Workshop, Sintra, Portugal*, September 1998. (page 51)

[41] C. Ditze. A Step towards Operating System Synthesis. In *Proc. of the 5th Annual Australasian Conf. on Parallel And Real-Time Systems (PART)*. IFIP, IEEE, 1998. (page 49, 51)

[42] C. Ditze. *Towards Operating System Synthesis*. Phd thesis, Department of Computer Science, Paderborn University, Paderborn, Germany, 1999. (page 24, 49, 51, 63, 127)

[43] C. Ditze and C. Böke. Supporting Software Synthesis of Communication Infrastructures for Embedded Real-Time Applications. In *Proc. of the 15th IFAC Workshop on Distributed Computer Control Systems (DCCS)*, Como, Italy, September 1998. (page 25, 51, 153, 162)

[44] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proc. of the 15th Symposium on Operating System Principles. ACM Press*, December 1995. (page 50)

[45] ASCET-SD and ERCOS by ETAS GmbH & Co. KG. at http://www.etas.de/. (page 3, 25)

[46] K. Forbus and J. deKleer. *Building Problem Solvers*. MIT Press, 1993. (page 39)

[47] P. Friedland. Knowledge-Based Experiment Design in Molecular Genetics. Report STAN-CS-79-771, Stanford University, 1979. (page 39)

[1] `http://www.cygnus.com/ecos`

[48] J. S. Gero. Design Prototypes: A Knowledge Representation Scheme for Design. *AI Magazine*, 11:26–36, 1990. (page 27)

[49] A. Gheith and K. Schwan. $CHAOS^{arc}$: Kernel support for multiweight objects, invocations, and atomicity in real-time multiprocessor applications. *ACM Transactions on Computer Systems*, 11(1)(1):33–72, February 1993. (page 50)

[50] A. Günter and C. Kühn. Knowledge-Based Configuration - Survey and Future Trends. In F. Puppe, editor, *Expertensysteme '99 Lecture Notes*. Springer, 1999. http://www.hitec-hh.de/aguenter/literatur.html. (page 37)

[51] A. Günter, I. Kreuz, and C. Kühn. Kommerzielle Software-Werkzeuge für die Konfigurierung von technischen Systemen. *Künstliche Intelligenz*, 3:61–65, 1999. ArenD-Tap Verlag, http://www.hitec-hh.de/aguenter/literatur.html. (page 44, 46)

[52] A. Günter, I. Syska, and R. Cunis. PLAKON Anforderungskatalog. TEX-K Report 4, Fachbereich Informatik, Universität Hamburg, Germany, 1987. (page 42, 43)

[53] S. Golestani. A Stop-And-Go Queueing Framework for Congestion Management. In *Proceedings of SIG-COMM'90*, June 1990. (page 82)

[54] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey. *Annals of Discrete Mathematics*, 5, 1979. (page 68)

[55] H. Güsgen, U. Junker, and A. Voss. Constraints in a Hybrid Knowledge Representation System. In *IJCAI-87*, pages 30–33, Los Altos, 1987. Morgan Kaufmann. (page 42)

[56] A. Günter. KONWERK - ein modulares Konfigurierungswerkzeug. In M. Richter and F. Maurer, editors, *Expertensysteme 95 (XPS-95)*, pages 1–18. Hundt, Köln, 1995. (page 44)

[57] A. Günter (ed.). *Wissensbasiertes Konfigurieren - Ergebnisse aus dem Projekt PROKON*. infix Verlag, St. Augustin, 1995. (page 44)

[58] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, Boston Dordrecht London, 1993. (page 3)

[59] W. Hardt, F. J. Rammig, C. Böke, C. Ditze, J. Stroop, A. Rettberg, G. D. Castillo, and B. Kleinjohann. IP-based System Design within the PARADISE Design Environment. In *Euromicro Journal "Systems Architecture"*. Elsevier, 2000. (page 161, 162)

[60] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987. (page 25)

[61] H. Haugeneder, E. Lehmann, and P. Struss. Knowledge-Based Configuration of Operating Systems – Problems in Modeling the Domain Knowledge. In *Proc. Wissensbasierte Systeme*, pages 121–134, Berlin, Germany, 1985. Springer. (page 42)

[62] M. Heinrich. Ressourcenorientiertes Konfigurieren. *Künstliche Intelligenz*, 7(1):11–15, 1993. (page 44)

[63] W. Horn. Some Simple Scheduling Algorithms. *Naval Research Logistics Quarterly*, 21, 1974. (page 69)

[64] StateMate by I-Logix. at http://www.ilogix.com/. (page 3, 25)

[65] INMOS Limited. *ANSI C toolset user manual*, August 1990. (page 14)

[66] International Organization for Standardization (ISO). International Standard 35.100.01 *Open Systems Interconnection* (OSI). ISO/IEC 7498-1, 1994. Basic Reference Model at http://www.iso.ch/. (page 52, 96, 110)

[67] ISO. Road Vehicles – Interchange of Digital Information – Controller Area Network (CAN) for High Speed Communication. ISO/DIS Standard 11898, 1992. (page 83, 142)

[68] J. R. Jackson. Scheduling a Production Line to Minimize Maximum Tardiness. In *Management Science Research Project*, volume 43, Los Angeles, 1955. University of California. (page 69)

[69] K. Jeffay and D. L. Stone. Accounting for Interrupt Handling Costs in Dynamic Priority Task Systems. In *Real-Time Systems Symposium*, pages 212–221. IEEE, December 1993. (page 73)

[70] K.-P. Jäker, P. Klingebiel, U. Lefarth, J. Richert, and R. Rutz. Tool integration by way of a computer-aided mechatronic laboratory (CAMeL). In *5th IFAC/IMACS Symposium on Computer Aided Design in Control Systems*, 1991. (page 3)

[71] H. Kise, T. Ibaraki, and H. Mine. A Solvable Case of the One Machine Scheduling Problem with Ready and Due Times. *Operations Research*, 26:121–126, 1978. (page 69)

[72] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Boston Dordrecht London, 1997. (page 64, 84)

[73] H. Kopetz, A. Damm, C. Koza, and M. Mulozzani. Distributed fault tolerant real-time systems: The Mars approach. *IEEE Micro*, pages 25–40, 1989. (page 84)

[74] H. Kopetz, G. Fohler, G. Grünsteidl, H. Kantz, G. Pospischil, P. Puschner, J. Reisinger, R. Schlatterbeck, W. Schütz, A. Vrchoticky, and R. Zainlinger. Real-time system development: The programming model of MARS. In *Proc. of the Int. Symp. on Autonomous Decentralized Systems*, pages 190–199, March 1993. (page 84)

[75] D. Ku and G. de Micheli. Relative Scheduling Under Timing Constraints. In *Proc. of the 27th ACM/IEEE Design Autom. Conference*, Orlando, Florida, USA, June 1990. (page 25)

[76] E. L. Lawler. Optimal Sequencing of a Single Machine Subject to Precedence Constraints. *Managements Science*, 19, 1973. (page 70)

[77] J.-I. Le Boudec. The Asynchronous Transfer Mode: a tutorial. *Computer Networks and ISDN Systems*, 24:279ff., 1992. (page 85)

[78] E. Lehmann, R. Enders, H. Haugeneder, R. Hunze, C. Johnson, L. Schmid, and P. Struss. SICONFEX – ein Expertensystem für die betriebliche Konfigurierung eines Betriebssystems. In *Proc. GI/OCG/ÖGI-Jahrestagung*, pages 792–805, Berlin, Germany, 1985. Springer. (page 42)

[79] J. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced Aperiodic Responsiveness in Hard Real-Time Environments. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1987. (page 75, 76)

[80] J. P. Lehoczky. Fixed Priority Scheduling of Periodic Tasks with Arbitrary Deadlines. In *Proc. of the 11th IEEE Real-Time Systems Symposium*, pages 201–209, 1990. (page 83)

[81] J. P. Lehoczky and S. Ramos-Thuel. An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1992. (page 76)

[82] J. K. Lenstra and A. H. G. Rinnooy Kan. Optimization and Approximation in Deterministic Sequencing and Scheduling. *Annals of Discrete Mathematics*, 5:287–326, 1977. (page 69)

[83] J. K. Lenstra, A. H. G. Rinnooy Kan, and P. Bruckner. Complexity of Machine Scheduling Problems. *Annals of Discrete Mathematics*, 1:343–362, 1977. (page 69)

[84] J. Y. T. Leung and J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks. *Performance Evaluation*, 4(2):237–250, 1982. (page 72)

[85] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, Vol. 20(1):46–61, January 1973. (page 71, 72)

[86] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood. A Formal Approach to Domain-Oriented Software Design Environments. In *Proc. of IEEE Knowledge-Based Software Engineering Conf.*, pages 48–57, Los Alamitos, CA, 1994. (page 25)

[87] H. Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. Phd thesis, Columbia University, 1992. (page 49)

[88] H. Massalin and C. Pu. Threads and input/output in the Synthesis kernel. In *12th Symposium on Operating System Principals, ACM SIGOPS*, volume 23, pages 191–201. ACM, December 1989. (page 49)

[89] J. McDermott. R1: A Rule-Based Configurer of Computer Systems. *Artificial Intelligence*, 19(1):39–88, 1982. (page 41)

[90] O. Najmann and B. Stein. A Theoretical Framework for Configuration. In F. Belli, editor, *5th International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems (IEA-92-AIE)*, volume 604, pages 441–450, New York Berlin Heidelberg Tokyo, June 1992. Springer-Verlag. (page 41)

[91] B. Neumann. Configuration Expert Systems: a Case Study and Tutorial. In H. Bunke, editor, *Artificial Intelligence in Manufacturing, Assembly and Robotics*, Oldenbourg, Munich, 1988. (page 19, 28, 40, 43)

[92] B. Neumann. *Ein Ansatz zur wissensbasierten Auftragsprüfung für technische Anlagen des Breitengeschäfts*. Dissertation, Gerhard-Mercator-Universität - GH Duisburg, 1990. (page 41)

[93] B. Neumann, R. Cunis, A. Günter, and I. Syska. Wissensbasierte Planung und Konfigurierung. In *Proc. of Wissensbasierte Systeme*, pages 347–357, Berlin, Germany, 1987. Springer. (page 43)

[94] N. J. Nilsson. *Principles of Artificial Intelligence*. Springer-Verlag, 1982. (page 29, 36)

[95] J. Nolte and W. Schröder-Preikschat. Modeling Replication and Placement in the PEACE Parallel Operating System — A Case for Dual Objects. 1992. (page 51)

[96] R. Oakley. QNX microkernel technology: A scalable approach to realtime distributed and embedded systems[2]. *Proc. of the Embedded Computer Conference (ECC'94)*, June 1994. (page 48)

[97] J. Pearl. *Heuristics*. Addison Wesley Publishing Company, 1984. (page 29)

[98] J. Peterson and A. Silberschatz. *Operating System Concepts*. Addison-Wesley, 1985. (page 85)

[99] F. Plagge. *Ambrosia: Ein Echtzeit-Betriebssystem für Automobil-Steuergeräte*. Number 560 in 10. VDI Verlag, Düsseldorf, Germany, 1998. (PhD Thesis, in German). (page 51)

[100] N. Pryce and S. Crane. A Uniform Approach to Configuration and Communication in Distributed Systems. In *Proc. of Conf. on Distributed Systems*, pages 144–151, 1996. (page 59)

[101] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis Kernel. *Computing Systems*, 1(1)(1):11–32, Winter 1998. (page 50)

[102] F. Puppe. *Problemlösungsmethoden in Expertensystemen*. Springer Verlag, 1990. (page 39, 41)

[103] R. Rajkumar. *A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991. (page 86)

[104] F. J. Rammig and C. Böke. Entwurf konfigurierbarer, echtzeitfähiger Kommunikationssysteme. Erstantrag im Rahmen des Schwerpunktprogramms "'Entwurf und Entwurfsmethodik eingebetteter Systeme"' an die DFG, March 1997. Fachbereich Informatik, Universität Paderborn, Germany. (page 162)

[105] B. Randell, J.-C. Laprie, H. Kopetz, and B. Littlewood. *Predictably Dependable Computing Systems*. Springer-Verlag, 1995. (page 64)

[106] J. M. Reitz. Online Dictionary of Library and Information Science. http://vax.wcsu.edu/library/odlis.html, 2002. (page 26)

[107] J. Richert and R. Rutz. CAMEL — An open CACSD environment. In *Joint Symposium on CADCS*, Tucson, Arizona, USA, March 1994. (page 3)

[2] `http://www.qnx.com`

[108] A. Rubini. *Linux Device Drivers*. O'Reilly & Associates, February 1998. (page 48)

[109] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley Publishing Company, Bonn, Germany, 1999. (page 113)

[110] C. Rust, J. Tacken, and C. Böke. Pr/t-Net-based Seamless Design of Embedded Real-Time Systems. In *Proceedings of the International Conference on Application and Theory of Petri Nets (ICATPN'01)*, Newcastle upon Tyne, U.K., 2001. Springer-Verlag. (page 161, 162)

[111] H. Saito. Optimal queueing discipline for real-time traffic at atm switching nodes. *IEEE Transaction on Communications*, 38(12):pp. 2131–2136, 1990. (page 85)

[112] S. S. Sathaye and J. K. Strosnider. A Real-Time Scheduling Framework for Packet-Switched Networks. In *Proc. of the 14th Int. Conf. on Distributed Computing Systems*, pages 182–191. IEEE, Poznan, Poland, June 21-24 1994. (page 78)

[113] H. Schmidt. Strategies towards Dynamic Alterability. In *Proc. of the 27th Annual Hawaii Int. Conf. on System Sciences*, pages 66–75. IEEE, 1994. (page 51)

[114] W. Schröder-Preikschat, U. Spinczyk, F. Schön, and O. Spinczyk. Design Rationale of the PURE Object-Oriented Embedded Operating System. In *Proc. of the Workshop on Distributed and Parallel Embedded Systems, (DIPES'99)*. IFIP, 1999. (page 49, 51)

[115] A. C. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, Vol. 39(9):1175–1185, September 1990. (page 85, 87)

[116] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic Task Scheduling for Hard Real-Time Systems. *Journal of Real-Time Systems*, 1, 1989. (page 76)

[117] M. Spuri and G. C. Buttazzo. Efficient Aperiodic Service under Earliest Deadline Scheduling. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1994. (page 76)

[118] M. Spuri and G. C. Buttazzo. Scheduling Aperiodic Tasks in Dynamic Priority Systems. *Journal of Real-Time Systems*, 10(2), 1996. (page 76)

[119] M. Spuri, G. C. Buttazzo, and F. Sensini. Robust Aperiodic Scheduling under Dynamic Priority Systems. In *Real-Time Systems Symposium*. IEEE, December 1995. (page 72)

[120] D. Sriram. ALL-RISE: A Case Study in Constraint-Based Design. *Artificial Intelligence in Engineering*, Vol. 2(No. 4):pages 186–203, 1987. (page 40, 42)

[121] D. Sriram and M. L. Maher. The Representation and Use of Constraints in Structural Design. *Proc. of Applications of Artificial Intelligence in Engineering Problems*, pages 355–368, 1986. (page 42)

[122] J. A. Stankovic. Misconceptions About Real-Time Computing. *IEEE Computer Magazine*, 21(10), October 1988. (page 64)

[123] J. A. Stankovic and K. Ramamritham. The Design of the Spring Kernel. In *Proc. of the IEEE Real-Time Systems Symposium*, pages 146–157. IEEE, December 1987. (page 70)

[124] J. A. Stankovic and K. Ramamritham. The Spring Kernel: A New Paradigm for Real-Time Systems. *IEEE Software*, 8(3):62–72, May 1991. (page 70)

[125] J. A. Stankovic, M. Spuri, K. Ramamritham, and G. C. Buttazzo. *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*. The Kluwer international series in engineering and computer science. Kluwer Academic Publishers, Boston Dordrecht London, real-time systems edition, 1998. (page 61, 62)

[126] B. Stein. *Functional Models in Configuration Systems*. Phd thesis, Department of Mathematics and Computer Science, University of Paderborn, Germany, June 1995. (page 37)

[127] B. Stein. Configuration and Diagnosis. Script to course, Universität Paderborn, Germany, 1997, 2000. http://www.uni-paderborn.de/cs/ag-klbue/staff/stein/courses/index.html. (page 29, 37)

[128] B. Stein. Heuristische Suchverfahren – Theory of Search[3]. Script to a course, Universität Paderborn, Germany, 2001. (page 29, 31, 32, 36)

[129] B. Stein and D. Curatolo. Model Formulation and Configuration of Technical Systems. In J. Sauer, A. Günter, and J. Hertzberg, editors, *10. Workshop Planen und Konfigurieren (PUK 96)*, number 3 in Proceedings in Artificial Intelligence, pages 56–70. Infix, Bonn, Germany, April 1996. (page 38, 44)

[130] B. Stein and R. Lemmen. ARTDECO: A System which Assists the Checking of Hydraulic Circuits. In *10th European Conference on Artificial Intelligence (ECAI 92), Workshop for Model-based Reasoning*, Christian Doppler Laboratory for Expert Systems, Vienna, Austria, July 1992. (page 44)

[131] B. Stein and E. Vier. An Approach to Formulate and to Process Design Knowledge in Fluidics. In N. E. Mastorakis, editor, *Recent Advances in Information Science and Technology*, pages 237–242. World Scientific Publishing Co. Pte. Ltd., London WC2H 9HE, October 1998. Second Part of the Proceedings of the 2nd IMACS International Conference on Circuits, Systems and Computers (CSC'98). (page 27)

[132] D. B. Stewart and P. Khosla. The Chimera methodology: designing dynamically reconfigurable real-time software using port-based objects. In *First Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'94), Dana Point, CA, USA*, October 1994. (page 57)

[133] D. B. Stewart and P. Khosla. Chimera methodology: designing dynamically reconfigurable real-time software using port-based objects. In *Workshop on Object-Oriented Real-Time Dependable Systems (WORDS), Proceedings 1995. IEEE, Piscataway, NJ, USA. p 46-53*. IEEE, 1995. (page 57)

[3] `http://www.uni-paderborn.de/cs/ag-klbue/staff/stein/courses/index.html`

[134] D. B. Stewart, D. E. Schmitz, and P. K. Khosla. The Chimera II real-time operating system for advanced sensor-based control applications. *IEEE Transactions on Systems, Man, and Cybernetics*, 22(6):1282–1295, November 1992. (page 57)

[135] I. Stierand. *Ambrosia/MP – Ein Echtzeitbetriebssystem für eingebettete Mehrprozessorsysteme*. Berichte aus dem fachbereich informatik. phd thesis, Universität Oldenburg, May 2001. (page 55)

[136] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The Deferrable Server Algorithm for Enhancing Aperiodic Responsiveness in Hard Real-Time Environments. *IEEE Transactions on Computers*, 4(1), January 1995. (page 75)

[137] A. S. Tanenbaum. *Betriebssysteme (Operating Systems - Design and Implementation)*. Prentice Hall, 1987. (page 85)

[138] F. Thoen, M. Cornero, G. Goossens, and H. D. Man. Real-Time Multi-Tasking in Software Synthesis for Information Processing Systems. In *Proc. of the 1st International Symposium on System Synthesis (ISSS)*, Cannes, September 12-15 1995. http://146.103.254.1/vsdm/projects/sw_synthesis/. (page 25)

[139] F. Thoen, J. V. D. Steen, G. de Jong, G. Goossens, and H. D. Man. Multi-Thread Graph – A System Model for Real-Time Embedded Software Synthesis. In *Proc. of ED&TC'97 Conference*, Paris, March 17-20 1997. http://146.103.254.1/vsdm/projects/sw_synthesis/. (page 25)

[140] K. Tindell, A. Burns, and A. J. Wellings. Analysis of Hard Real-Time Communications. In *Real-Time Systems*, volume 9, pages 147–171. Kluwer Academic Publishers, Boston, 1995. (page 82)

[141] U.S. Army Missile Command, Redstone Arsenal, Alabama 35898-5254. *RTEMS/C Applications User's Guide*[4], May 1995. (page 48)

[142] Vector Informatik GmbH. Homepage. http://www.vector-informatik.de/deutsch/index.html?../produkte?davinci.html, 2003. (page 58)

[143] E. Verhulst. Virtuoso: A virtual single processor programming system for distributed real-time applications. *Microprocessing and Micoprogramming*, 40:103–115, 1994. (page 49)

[144] E. Verhulst. Beyond transputing : fully distributed semantics in Virtuoso's Virtual Single Processor programming model and it's implementation on of-the-shelf parallel DSPs. In A. Bakkers, editor, *Parallel Programming and Java, Proceedings of WoTUG 20*, volume 50 of *Concurrent Systems Engineering*, pages 77–87, University of Twente, Netherlands, April 1997. World occam and Transputer User Group (WoTUG), IOS Press, Netherlands. (page 49)

[145] E. Verhulst. Non-sequential processing: Bridging the semantic gap of the von Neumann architecture. In *1997 IEEE Workshop on Signal Processing Systems (SiPS)*. Eonic Systems Inc., USA, November 1997. (page 49)

[4] `http://lancelot.gcs.redstone.army.mil/rg4/c/cuser.html`

[146] M. Wernicke. Design mit System – Funktionsorientierte Entwicklung von verteilter Kfz-Software. In *Sonderheft Elektronik Automotive*, pages 46–49, Poing, Germany, December 2002. WEKA Fachzeitschriften-Verlag GmbH. (page 58)

[147] M. Wernicke. Verteilte Systeme im Griff. In *Automotive electronics + systems*, number 6, pages 28–30, Munich, Germany, 2002. Carl Hanser GmbH & Co. (page 58)

[148] Wind River Systems GmbH. VxWorks 5.4[5]. Product Overview, June 1999. (page 25, 50)

[149] Y. Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. In *Proc. of OOPSLA'92*, pages 414–434, ACM Press, October 1992. (page 50)

[150] S. Young. *Real-Time Languages: Design and Development*. Ellis Horwood, 1982. (page 64)

[151] M. Zanella, T. Koch, U. Meier-Noe, F. Scharfeld, and A. Warkentin. Structuring and Distribution of Controller Software in Dependence of the System Structure. In *CBA 2000*, Florianapolis, Brazil, 2000. (page 3)

[5] `http://www.wrs.com/html/products/vxwks54.html`