DISSERTATION

Design Comprehension of Embedded Real-Time Systems

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines

Doktors der technischen Wissenschaften

unter der Leitung von

O.Univ.-Prof. Dr. Hermann Kopetz
Institut für Technische Informatik 182

eingereicht an der

Technischen Universität Wien
Fakultät für Informatik

von

Bernhard Rumpler
Matr.-Nr. 9725344
Michael Bernhard-Gasse 4/Top 3
1120 Wien

Wien, im Juni 2008 .....

Design Comprehension of Embedded Real-Time Systems

Abstract

Complexity management is a hot topic in the current computer science literature. However, usually shallow explanations at the technical level are used as arguments for specific complexity management techniques and complexity metrics. This dissertation aims at bridging the gap between computer science on the one side and the cognitive and learning sciences on the other side with an interdisciplinary approach to *design comprehension*. A special focus is put on high-criticality systems where latent design errors caused by highly complex development tasks cannot be tolerated.

This work focuses on the *cognitive complexity* of development tasks, which denotes the amount of cognitive resources that are required for a given task. Overload of the human working memory system is a major factor that affects cognitive performance. Those tasks where many relational aspects must be considered simultaneously are especially complex.

The basic goal of this work is the development of *cognitive support theories* for various system development approaches. Design for simplicity is an important principle to keep the cognitive complexity of design and maintenance tasks at a manageable level. Various techniques to achieve a simple and understandable system structure are presented. A special focus is put on the characteristics of component interfaces, as it is the characteristics and the placement of the interfaces that affects the cognitive effort of system comprehension.

It must be possible to develop and verify each component independently and then integrate the components with just minimal effort. The design concept of *near-independence* can be used to minimize relational aspects in large systems. Nearly-independent components represent units of design with a high degree of self-containedness. They are a key technique to minimize global interdependencies. *Temporal firewalls* together with a *sparse global time-base* represent suitable techniques to support a high degree of temporal and conceptual decoupling between components.

The presented approach supports determinism at the application level by the provision of various architectural services. Determinism enables deductive reasoning, which is essential for the rational analysis of behavior.

Designverstehen von eingebetteten Echtzeitsystemen

Kurzfassung

Obwohl Komplexitätsmanagement ein zentrales Thema der Informatik ist, wird es in der Literatur nur sehr oberflächlich behandelt. Erkenntnisse aus den Kognitions- und Lernwissenschaften finden kaum Beachtung. Daher verfolgt diese Dissertation einen interdisziplinären Ansatz zur Verbesserung des Designverstehens von eingebetteten Echtzeitsystemen für sicherheitskritische Anwendungen. In diesen Systemen können latente Designfehler, verursacht durch hohe Komplexität von Entwicklungsaufgaben, nicht toleriert werden.

Einen zentralen Aspekt dieser Arbeit stellt die kognitive Komplexität der Entwicklungaufgaben dar, welche das Ausmaß an kognitiven Ressourcen beschreibt, die für die Aufgaben benötigt werden. Die beschränkte Kapazität des menschlichen Arbeitsgedächtnisses stellt einen zentraler Faktor für mentale Einschränkungen dar. Jene Aufgaben, wo viele Aspekte gleichzeitig betrachtet werden müssen, sind besonders komplex.

Ein wesentliches Ziel der Arbeit ist die Betrachtung von verschiedenen Ansätzen zum Systemdesign hinsichtlich ihrer Komplexität. Die Eigenschaften von Schnittstellen zwischen den Systemkomponenten sind dabei von zentraler Bedeutung, da sie die Verständlichkeit eines Systems wesentlich beeinflussen.

Um die Komplexität eines Systems beherrschen zu können ist es erforderlich, dass die Systemkomponenten weitgehend unabhängig voneinander entwickelt und verifiziert werden können. Dazu ist es notwendig, die relationalen Abhängigkeiten innerhalb des Systems zu minimieren. Das Komponentenmodell dieser Arbeit, das auf *Temporal Firewalls*, in Kombination mit einem speziellen globalen Zeitmodell basiert, ermöglicht eine weitgehende zeitliche und konzeptionelle Entkopplung der Komponenten. Die Integration der Komponenten wird dadurch vereinfacht, dass sich die Eigenschaften der Komponenten durch die Integration nicht verändern.

Der präsentierte Ansatz ermöglicht Determinismus auf der Applikationsebene. Dies unterstützt die Verständlichkeit, da sich das Verhalten des Systems durch Deduktion logisch ableiten lässt.

Danksagung

Besonderer Dank gilt meinem Betreuer O.Univ.Prof. Dr.phil. Dr.h.c. Hermann Kopetz, der der interdisziplinären Forschung zwischen Informatik und kognitiver Psychologie einen hohen Stellenwert beimisst und mir die Arbeit am Institut für Technische Informatik ermöglichte. Er stand mir stets mit hilfreichen Anregungen zur Seite und trug maßgeblich zum Gelingen der Arbeit bei.

Ein großes Dankeschön möchte ich auch meiner Zweitbegutachterin Frau Ao.Univ.Prof. Mag.rer.soc.oec. Dr.phil. Margit Pohl aussprechen, für die wertvollen Diskussionen und Vorschläge zum kognitionswissenschaftlichen Teil dieser Arbeit.

Ganz besonders möchte ich mich bei meinen Eltern Hannelore und Herbert bedanken, welche mir mein Studium ermöglicht und mich stets mit allen Kräften unterstützt haben.

Besonderer Dank gilt auch Martina, die mich besonders bei der Fertigstellung der Arbeit motiviert hat.

Mein Dank gilt auch dem österreichischen Bundesministerium für Verkehr, Innovation und Technologie (BMVIT), welches diese Arbeit im Rahmen des FIT-IT Programms (Projektnummer 809442) unterstützt hat.

Contents

1 Introduction **1**

1.1 What is Design? 1

1.1.1 A Definition of Design 1

1.1.2 Design Problems – Definition and Characteristics . . . 2

1.1.3 Trial-and-Error vs. Guided Design 3

1.2 What is Complexity? 3

1.2.1 Definitions as System Characteristics 4

1.2.2 Cognitive Complexity 4

1.3 Embedded Real-Time Systems and Complexity 5

1.3.1 Factors Specific to Embedded Real-Time Systems . . 5

1.3.2 Factors for Growing Complexity 6

1.3.3 Problems Caused by Complexity 6

1.4 Improving Understandability 7

1.4.1 Advances in the Cognitive Sciences 7

1.4.2 Design Comprehension 8

1.5 Scope and Objectives 9

1.5.1 Focus on High-Criticality Real-Time Systems 9

1.5.2 Design for Simplicity 9

1.5.3 Cognitive Support Theories 10

1.5.4 Near-Independence and Composability 10

1.6 Related Work 10

1.6.1 Information Visualization and Human Computer Interaction 11

1.6.2 Program Comprehension 12

1.6.3 Software Engineering 13

1.6.4 Complexity Metrics 13

1.7 Overview 14

2 Cognitive Processes of Understanding **15**

2.1 The Cognitive Perspective 15

2.1.1 The Sciences of the Mind 15

2.1.2 Mental Representations 16

2.2 Memory 17

2.2.1 Working Memory 17
2.2.2 Long-Term Memory 19
2.2.3 Encoding, Storing and Retrieving Information 19
2.3 Semantic Memory 20
2.3.1 Rule-Governed Concepts 21
2.3.2 Object Concepts 21
2.3.3 Schemata and Concept-Driven Processing 23
2.3.4 Abstract Thinking 24
2.4 Expertise 24
2.4.1 Expert/Novice Differences 24
2.4.2 Becoming an Expert 25
2.5 Chapter Summary 26

3 Cognitive Complexity **29**
3.1 Cognitive Characteristics 30
3.1.1 Limits of Working Memory 30
3.1.2 Biased Thinking and Heuristics 30
3.1.3 Probabilistic Reasoning 31
3.1.4 Oversimplification 31
3.1.5 Dependency on Domain Knowledge 31
3.1.6 Dependency on Metacognition 32
3.2 Characteristics of Complex Problems 32
3.2.1 Conceptual Complexity 32
3.2.2 Dimensions of Difficulty 32
3.2.3 Emergent Views 34
3.2.4 Complex Causality 34
3.2.5 Cognitive Load 35
3.2.6 Relational Complexity 35
3.2.7 Complex Mental Models 36
3.3 Assessing Cognitive Complexity 37
3.3.1 Well-Defined Problems 37
3.3.2 Design Problems 38
3.3.3 Component Interface Level 38
3.3.4 Component Implementation Level 39
3.4 Chapter Summary 39

4 Real-Time Systems Concepts **43**
4.1 System Model 43
4.1.1 Architectures 43
4.1.2 Systems and Components 44
4.1.3 Application Structure 44
4.1.4 Component Behavior 45
4.1.5 System State 45
4.1.6 Composability 46
4.2 Models of Time 46
4.2.1 Dense Time vs. Sparse Time 47
4.2.2 Global Time 48

4.3 Real-Time Entities and Real-Time Images 48
4.3.1 Real-Time Entities 48
4.3.2 Real-Time Images 49
4.4 Interfaces 49
4.4.1 Interface Properties 50
4.4.2 Interface Types 51
4.4.3 Open vs. Closed Components 52
4.5 Specifications 52
4.5.1 Specification of Semantics 53
4.5.2 Varying Degrees of Formalism 53
4.5.3 Formal Specifications 53
4.5.4 Interface Specifications 54
4.5.5 Uses of Specifications 55
4.6 Interaction Styles 55
4.6.1 Client-Server Interactions 56
4.6.2 Publish/Subscribe 56
4.6.3 Data Passing via a Repository 57
4.7 Interaction Contents 57
4.7.1 Implicit vs. Explicit Interaction Content 58
4.7.2 Atomic vs. Composite Interaction Content 58
4.7.3 State Information vs. Event Information 59
4.8 Dependability 59
4.8.1 Fault - Error - Failure 59
4.8.2 Fault-Tolerance 60
4.8.3 Error Containment 60
4.8.4 Dependability Attributes of Interactions 61
4.8.5 Replica Determinism 61
4.9 Time- vs. Event-Triggered Paradigm 62
4.9.1 Event-Triggered Systems 62
4.9.2 Time-Triggered Systems 63
4.9.3 Two-Level Design Methodology 64
4.10 Chapter Summary 65

5 Complexity Management **67**
5.1 General Considerations 67
5.1.1 What is Complexity Management? 67
5.1.2 Accidental vs. Essential Characteristics 68
5.1.3 Design for Simplicity 69
5.2 Standardization and Architectures 70
5.2.1 Requirements 70
5.2.2 Advantages 70
5.3 Interface Issues 71
5.3.1 Support for Chunking and Segmentation 72
5.3.2 Abstraction and Conceptual Chunking 72
5.3.3 Partitioning and Segmentation 73
5.3.4 The Interface View 74
5.3.5 The Interaction View 74

5.3.6 Component Encapsulation . . . 75
5.3.7 Built-In Segmentation and Chunking . . . 77
5.3.8 The Optimal Component Interface . . . 78
5.4 Seeing Connections . . . 79
5.4.1 Making Connections Explicit . . . 80
5.4.2 Minimizing Connections . . . 81
5.5 Exploiting Regularity . . . 83
5.5.1 Structural Regularity . . . 84
5.5.2 Behavioral Regularity . . . 85
5.6 System Structuring . . . 86
5.6.1 Hierarchies . . . 86
5.6.2 Layering . . . 87
5.6.3 Layering vs. Partitioning . . . 88
5.6.4 Job-Based Structure . . . 89
5.7 Component Integration Issues . . . 89
5.7.1 Attributive, Relational and Emergent Properties . . . 90
5.7.2 Complexity of Component Integration . . . 91
5.7.3 Traditional Approaches . . . 92
5.7.4 Self-Contained System-Level Components . . . 93
5.7.5 System-Level Services . . . 93
5.7.6 Integration Levels . . . 94
5.8 Determinism . . . 95
5.8.1 Logical Reasoning . . . 95
5.8.2 Consistency . . . 96
5.9 Reducing the Problem Space . . . 96
5.9.1 Removing Irrelevant Information . . . 96
5.9.2 Using Categories . . . 97
5.9.3 Minimizing the Number of Model Interpretations . . . 98
5.10 Failure Handling . . . 99
5.10.1 Stability of Models . . . 99
5.10.2 Separation of Concerns . . . 99
5.11 Chapter Summary . . . 100

6 DECOS Analysis **103**
6.1 The DECOS Architecture . . . 103
6.1.1 Physical System Structure . . . 103
6.1.2 Functional System Structure . . . 103
6.1.3 Architectural Services . . . 105
6.1.4 DASs and Virtual Networks . . . 107
6.1.5 Development Process . . . 108
6.1.6 Diagnostic Strategy . . . 111
6.2 State Message Ports . . . 111
6.2.1 Conceptual Simplicity . . . 112
6.2.2 Unification of Interfaces - Temporal Firewalls . . . 112
6.2.3 Conceptual Decoupling . . . 113
6.2.4 Interface State . . . 113
6.2.5 State Message Objects . . . 113

6.3 Architectural Service Provision . . . 114
6.3.1 Reduction of the Problem Space . . . 114
6.3.2 Support for Determinism . . . 114
6.3.3 Separation of Concerns . . . 115
6.3.4 Support for Structural Regularity . . . 115
6.3.5 Simplified Diagnosis . . . 116
6.3.6 Stability of Models . . . 116
6.4 Sparse Global Time . . . 116
6.4.1 Natural Model of Time . . . 117
6.4.2 Modeling Simultaneity . . . 117
6.4.3 Consistency and Order . . . 118
6.4.4 Reduced Model of Time . . . 118
6.4.5 Availability of a Model of Time . . . 119
6.4.6 Temporal Decoupling . . . 120
6.5 System Structure . . . 120
6.5.1 Simple Application Structure . . . 120
6.5.2 Straightforward DAS Integration . . . 121
6.5.3 Near-Independence of Components . . . 122
6.6 Development Process . . . 123
6.6.1 Two-Level Application Design . . . 123
6.6.2 Model-Based Application Design . . . 124
6.6.3 Hardware Platform Modeling . . . 126
6.7 Time- vs. Event-Triggered Control . . . 126
6.7.1 Conceptual Structuring . . . 127
6.7.2 Modeling Cause and Effect . . . 128
6.7.3 Segmentation Boundaries . . . 129
6.7.4 Behavioral Variety . . . 129
6.7.5 Size of the Problem Space . . . 130
6.7.6 Hybrid Applications . . . 130
6.8 Chapter Summary . . . 131

7 Conclusions **135**
7.1 Summary . . . 135
7.2 Contributions . . . 137
7.3 Open Issues . . . 138

Bibliography **138**

Chapter 1

Introduction

Embedded real-time systems are becoming increasingly important in many areas of our life. We use intelligent appliances that are controlled by embedded computer systems every day, from a simple electric toothbrush to a car controlled by a large number of electronic components.

This dissertation focuses on one aspect of complexity – the cognitive complexity of understanding safety-critical embedded real-time computer systems during the development and maintenance process. This developer-centered view on system understanding is called *design comprehension*. The interrelations between technical and psychological aspects are discussed in detail, which leads to some important implications for system development.

1.1 What is Design?

The term *design* is generally accepted as a well-known term and not defined in detail. However, there exist various concepts related to this term – most notably there is a difference between a more artistic and a more technical viewpoint.

1.1.1 A Definition of Design

In this work we are interested in the technical perspective of design only. Christopher Alexander, an architect who is widely accepted as the father of *design patterns* [AIS77], which have also been adopted in computer science (e.g., [GHJV94]), has introduced a good definition. In one of his early books on design [Ale64] he writes: *The ultimate object of design is form. [...] Every design problem begins with an effort to achieve fitness between two entities: the form in question and its context. The form is the solution to the problem; the context defines the problem. [...] The form is the part of the world over which we have control, and which we decide to shape while leaving the rest of the world as it is. The context is the part of the world which puts demand on this form; anything in the world that makes demands*

of the form is context. Fitness is a relation of mutual acceptability between the two. [...] It is only through the form that we can create order in the ensemble. This is a very general definition of design. So it is very well-suited for this dissertation where a very general discussion of design from the viewpoint of comprehensibility is taken.

In essence, the main problem statement of this work is what makes a design problem and its constituent tasks either easy or hard to understand. The considerations are focused on embedded real-time systems. However, many statements apply to the much wider domain of computer systems design as a whole.

1.1.2 Design Problems – Definition and Characteristics

Now, after having a definition of design, what makes a problem a design problem? The following differentiation between *design problems* and *selection problems* has again been taken from Alexander. For a selection problem a *unitary description* can be given, whereas a design problem demands *invention*. To solve a problem just by selection, two things are necessary [Ale64] :

1. It must be possible to generate a wide enough range of possible alternative solutions symbolically.
2. It must be possible to express all the criteria for solution in terms of the same formalism.

So a problem is a design problem only if selection cannot be used to solve it.

In this thesis the tasks that can occur during system development are differentiated into *well-defined tasks* and *ill-defined tasks*. The latter are real design tasks according to the definition above, whereas the former correspond to the selection tasks. For example, a well-defined task can be fully described with a detailed work instruction, e.g., for a maintenance engineer, or it can be automated with a tool, such as an automatic scheduler. An example for an ill-defined task is the generation of a system description from a number of potentially incomplete or contradictory requirements. With most development methodologies the decomposition of a system description into a number of components is also a design problem.

In this dissertation, all tasks that are involved in the development of a computer system are called *development tasks*. A development task can be either a well-defined or a design task, if this is not mentioned explicitly.

According to Alexander, a solution to a design problem can be considered a *good design* if it minimizes the misfits – the potential problems that have to be solved – between form and context. However, a design problem is not an optimization problem; it is not necessary to find a solution that meets every single requirement in the best possible way. It is usually sufficient that

the solution satisfies the requirement at a level just to prevent misfit. It is therefore necessary that all requirements are clearly defined to be able to unambiguously classify a design a fit or a misfit. For quantifiable variables, such as the maximum latency of a reaction of a system, this is easy to define. For non-quantifiable variables, such as *understandability* the definition of a criterion to classify a solution as a fit or a misfit regarding the requirement may be considerable effort. So understandability is often ignored or defined in a very superficial manner.

A solution to a design problem that is a good design according to the definition above may not be the one that is most easily understandable. This means there may be a different solution that has better characteristics regarding comprehensibility. In this dissertation those aspects of design are considered, that can affect comprehensibility.

For the considerations about cognitive complexity both well-defined and ill-defined problems must be considered, as the creation of a new system as well as the adaption of an existing system usually involves both kinds of tasks. Some development tasks may be well-defined but very complex anyway – due to the nature of the tasks. Others may be ill-defined or involve a considerable amount of invention, which can make them difficult, or at least unpredictable.

1.1.3 Trial-and-Error vs. Guided Design

By definition, design problems can not be well-defined. Trial-and-error design is expensive and slow. But how can the design process be guided so that errors are minimized? The key is to re-use existing knowledge of previous design problems and to replace trial-and-error by guided processes. In computer science the use of *design patterns* [GHJV94] has become very popular to reuse proven concepts for new problems. *Computer system architectures* are another possibility to guide the system developers by providing a basic design framework that can be reused. Computer system architectures can thus be seen as kinds of design patterns.

1.2 What is Complexity?

There exist very different conceptualizations that all use the term *complexity*. As complexity is a central aspect of this thesis, this section first summarizes some of the common conceptualizations that are not used. Then, the term *cognitive complexity* is introduced, which is the only definition of complexity that is used in this work.

1.2.1 Definitions as System Characteristics

In formal and mathematical discussions the term complexity is used to describe system characteristics. For example, computational complexity [FH03], which covers various theoretical approaches for models of computation. It discusses, among other things, the amount of time and memory needed by a computer to solve a particular kind of problem. This definition of complexity is not used in this work.

A different concept of complexity is used by the so-called *new science of complexity* [NP89]. However, no exact definition of complexity exists in this research field, just characterizations and examples for aspects of complexity. Complexity in this area is also considered a characteristic of the system or of the system's behavior. For example, self-organization in physico-chemical systems is usually considered to be a complex phenomenon. Such a characterization of complexity is more related to the concept of complexity that is used in this work than the aforementioned computational approaches, as this kind of complex behavior is usually difficult to understand, too. However, the science of complexity takes a very formal and mathematical viewpoint that does not consider cognitive aspects at all, i.e., *why* we find it difficult to understand a system.

In hands-on oriented computer science, the term complexity is commonly used to describe a characteristic of the functionality of an artifact. For example, we can say that automotive electronics systems implement increasingly complex functionality. This complexity can be contrasted to simplicity in the sense that the complex artifact has more capabilities than the simple one.

Sometimes, more complex just means that more steps are required for a particular task. For example, when a component has to be configured in two steps this is sometimes said to be more complex than if just a single step is needed. As this is a quite different conceptualization, it is referred to as *more effort* rather than *more complex* in this work.

1.2.2 Cognitive Complexity

The cognitive complexity of a given task describes the amount of cognitive resources that are required to perform the task. If the cognitive complexity of a task is high, this becomes manifest in increased time required for the task and in a high number of errors that occur.

This task-based view of complexity is central to this thesis. Of course, a task such as component integration heavily depends on the characteristics of the underlying system. However, just looking at a system without considering tasks at all does not make sense when trying to make judgments about cognitive complexity. The characteristics that make a task either difficult or simple can either be rooted in system properties or they result from the task itself: From this viewpoint it does not make sense to consider the overall

complexity of the whole system, as the complexity always depends on the task that must be performed. Without a task, the description or even the calculation of complexity is mostly useless. So it does not make sense to "sum up" the cognitive complexity of a number of development sub-tasks to determine the complexity of the encompassing development task: Each task must be considered separately. However, it does make sense to determine the most complex task of a chain of tasks.

There are two main factors that affect the performance of a task. First, the knowledge (or expertise) in the problem domain and second, the inherent complexity of the task itself [HWP97]. A detailed discussion of cognitive complexity can be found in chapter 3.

In this thesis, the term complexity is just used as an abbreviation for cognitive complexity, so the terms complexity and cognitive complexity can be used interchangeably. If no task is mentioned explicitly, the task of gaining a general understanding of the system under consideration can be assumed. *Complex system* or *complex component* in this thesis always means that the system or component is difficult to understand.

1.3 Embedded Real-Time Systems and Complexity

The cognitive complexity of various tasks involved in the development and maintenance of many of today's embedded real-time systems has reached a threshold of pain where new paradigms for system development are required. For example, the complexity that arises during system development is one of the most challenging problems in automotive electronics development [Gri03, PYJ04]. The integration process of seemingly simple components often fails due to unexpected complexity at the system level, i.e., the complexity that arises due to the interactions of the components.

1.3.1 Factors Specific to Embedded Real-Time Systems

In this thesis a focus is put on *hard real-time systems* which are real-time systems where a single failure to produce results on time may be a system failure. Hard real-time systems are fundamentally different from *soft real-time systems* where the occasional violation of a deadline can be tolerated. Hard real-time systems require more development effort and are more difficult to design than soft real-time systems because the developer must consider two dimensions of the design problem simultaneously: the value dimension and the temporal dimension [Kop98].

Another important source for complexity in embedded real-time systems is their interaction with the physical world [SS99]. The system environment behaves non-deterministic, with events occurring asynchronously, concurrently and in an unpredictable order, but the system itself must respond

appropriately and in a timely fashion.

Embedded real-time systems often have high dependability requirements. Fault-tolerance is usually used to achieve the required level of dependability, but fault-tolerance can also contribute to considerable complexity: Various fault scenarios must be considered and error containment and recovery strategies must be provided. As will be shown in this thesis, the amount of additional complexity introduced by fault-tolerance mechanisms heavily depends on the type of fault-tolerance that is supported by the computer system architecture.

1.3.2 Factors for Growing Complexity

Embedded electronics systems have been growing fast over the last years: The introduction of new functionality as well as the replacement of mechanical by electronic components are the two major factors for this growth. However, system design methodologies for embedded real-time systems have not kept pace with this development and now face severe problems.

Years ago, electronics systems were designed according to a federated approach. This means that every subsystem was built as a separate system with only minimal or no interaction with other systems. Today, there is a trend from federated to *integrated architectures*, where multiple application subsystems share the same physical resources. This trend can be observed, e.g., in automotive and aerospace computer systems architectures, with AUTOSAR [AUT06, HSF+04], Integrated Modular Avionics (IMA) [Aer91] and DECOS [KOPS04] representing key initiatives towards such integrated systems. The major benefits of this integration are increased interoperability, a reduction of the number of computational nodes, cables and connectors, and an increased reliability of the overall system. A major drawback is the increased complexity of the development process and additional interdependencies between subsystems that are introduced due to the integration. These interdependencies are usually not modeled explicitly, but may exist due to the fact that the same physical hardware is shared. For example, the communication system can be highly loaded, or the processor may run at high load which results in varying delays and runtimes depending on other application subsystems.

1.3.3 Problems Caused by Complexity

There are high dependability requirements for embedded real-time systems in many application areas, e.g., automotive and avionics control, nuclear power stations, or air traffic control to mention just a few. A failure of such a system can have fatal consequences, including the loss of life. But also in other areas where real-time systems are used, e.g., for telecommunications systems, errors may have severe consequences – at least financial ones – if a system is out of service for some time.

In automotive electronics the growing cognitive complexity of development tasks has led to a high number of design faults in the recent past. These latent faults remained undetected during system design, testing and verification and were a major cause for recalls. Design faults usually have their roots in lacking understanding or overlooking of some system aspects. Such aspects may just be very inconspicuous details at a first glance, but have severe consequences if their influence is not fully understood during system development. If it is hard to analyze all possible influences of such an aspect, it is very likely that this analysis is not performed fully, or probably not done at all. The more cognitive effort a particular task requires, the more likely it is that errors are introduced. Wherever human understanding is required, such problems can occur. As of today, system design, implementation, integration and test is largely done by humans. Formal methods are still not mature enough for wide-spread use. See section 4.5.3 for a brief discussion of the limitations of formal methods.

1.4 Improving Understandability

The central approach of this thesis is to make use of the advances in the cognitive and learning sciences to analyze current real-time systems architectures and their design concepts with regard to comprehensibility. These results can then be used for improvements and to support developers in choosing an architecture that best fulfills their requirements. Understandability or simplicity are common requirements these days.

1.4.1 Advances in the Cognitive Sciences

Research about various aspects of cognitive complexity is quite active. Especially the cognitive and learning sciences are still fields with many open questions. However, there are some aspects of complexity and learning that are widely accepted. Lots of progress has been made in recent years regarding how understanding works and what makes up a complex problem. [SC94, Hol95, RW98, Rei01, Kel03, QKG05]. Extensive research has been done with regard to what understanding and expertise mean [PG00, Jac01, HSP04]. Furthermore, properties of material that is difficult to understand have been identified [HWP97, Ree99, FCS01, Bir02, HBMB05]. Even if there remain lots of open questions, the research results provide a valuable basis for application in other domains, such as computer science, where cognitive complexity is a fundamental problem and usually only tackled in a very superficial manner. The characteristics of human cognition play a fundamental role in understanding computer systems and must be regarded by system developers in order to create comprehensible systems. This is especially important in high-dependability domains where failures caused by errors rooted in highly complex tasks can have fatal consequences.

1.4.2 Design Comprehension

Although making sense of complex systems is a basic issue in the technical sciences, the recent advances in the cognitive and learning sciences have received surprisingly little attention in computer science [Rum05]. There exist some areas where psychological concepts and theories from the learning sciences are applied, e.g., learning systems, visualizations, or program comprehension. See section 1.6 for a summary of research areas of computer science where interdisciplinary approaches drawing on psychological foundations are used. However, computer science in general and embedded systems development in particular, largely ignore psychological issues. This practice starts with computer science courses at universities and continues throughout most of the scientific community. It is also present in companies that develop embedded systems. The AUTOSAR specification is an example where a central goal was complexity management, but except for standardization, the architecture does not provide any explicit complexity management techniques. There does not even exist a concise conceptual explanation of the architecture, just countless documents describing single technical aspects of the standard at a very detailed level. So reading and understanding the standard itself is quite a challenge.

In computer science, most concepts and theories that deal with complexity management have been developed without the theories of the cognitive and learning sciences in mind.

This thesis is about *design comprehension* [Rum05, RK06], which aims at bridging the gap between computer science on the one side and the cognitive and learning sciences on the other side with an interdisciplinary approach – see figure 1.1. The term design comprehension includes all aspects of understanding that occur during system development and maintenance. In other words, it investigates and tries to amplify the understanding of the relevant aspects of a computer system for a given task.

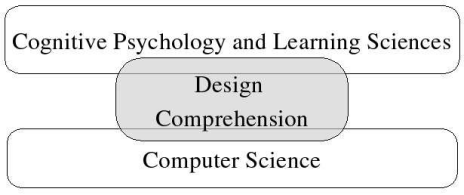

Figure 1.1: The interdisciplinary approach

The consideration of the characteristics of human comprehension allows to design more comprehensible systems and can help to improve current complexity management techniques by providing a systematic and scientific basis for the development of computer systems architectures.

To be able to make statements about the complexity of development tasks of embedded real-time systems, a conceptual framework that is rooted in the theories and principles of cognitive psychology is needed. The char-

acteristics of human cognition can be used to explain why we perceive a particular task as either simple or complex. The approach taken in this thesis is mainly on the theoretical level: The theories and principles from the cognitive and learning sciences are directly applied to the domain of embedded real-time systems concepts.

The terms *understanding* and *comprehension* are used synonymously in this work. The choice of one or the other term is just a matter of linguistic variation.

1.5 Scope and Objectives

This work focuses on the architectural characteristics of embedded real-time systems, such as the overall system structure, (component) interface design, the basic component interaction techniques, and those aspects of the development process that are described by the presented architecture.

The development of visual models [Obj03], visualization techniques, (visual) user-interface design, and the development of design or verification tools are interesting related topics. However, their consideration is beyond the scope of this thesis. So this thesis focuses on how computer system architectures can support comprehension, not on development tool support.

1.5.1 Focus on High-Criticality Real-Time Systems

This work focuses on embedded real-time systems for high-criticality domains, such as steer-by-wire systems for automotive, or fly-by-wire systems for avionics systems. For these dependable systems, the correct function is especially important; latent design errors cannot be tolerated.

Safety-critical systems as well as their development process are quite different to other computer systems, such as desktop computer applications. The development and verification processes of dependable systems are far more formal and rigorous than for other systems where failures do not have severe consequences. Certification is an important part of the development process of safety-critical systems, for example according to DO-178B [RTC92], which is the de-facto standard for avionics software systems.

Some of the concepts described in this work may also apply to other areas of computer system development, but the discussion is focused primarily on embedded real-time systems for high-dependability domains.

1.5.2 Design for Simplicity

When comparing automotive and avionics electronics development, the most severe difference is that the aerospace industry already has some experience with computer systems for safety-critical applications, whereas the automotive industry is just beginning to create safety-critical systems. This differ-

ence is also reflected by the application of standards that ensure that the development processes are sufficiently mature for highly reliable computer-controlled products. Currently, the automotive industry and most suppliers do not have sufficient process maturity to handle safety-critical projects [FFL07]. An important implication of this fact is that – especially in the automotive industry – we are far away from fully and formally specified and verified systems. This means that understandability is even more important to minimize design faults that are not detected due to deficiencies in the development process. But of course, even if all process objectives of a good development process are followed, development and certification are significantly easier if the system itself and the development tasks are easily understandable.

1.5.3 Cognitive Support Theories

A central goal of this work is the development of *cognitive support theories* for complexity management techniques of embedded real-time systems. Work about cognitive complexity that is based on psychological theories and experiments is scarcely found in the main-stream computer science literature. Too often, shallow explanations at the technological level are used as arguments for various approaches to complexity management. This work tries to dig a bit deeper and bridge the still existing gap between complex technical systems and psychological theories.

1.5.4 Near-Independence and Composability

Classical divide and conquer approaches, such as structured or object-oriented development, do not allow for independent component development and seamless integration. This dissertation favors a constructive development approach that builds on stable, nearly-independent components that can be developed and verified in isolation. The integration process is a very critical phase in the development of a real-time computer system as the properties that were established at the component level shall not be refuted by the integration. The approach presented in this thesis aims at the integration of composable components with just minimal effort.

1.6 Related Work

Measuring the complexity of systems, supporting the understanding of complex problems and avoiding complexity with appropriate development methodologies has been subject to research in the history of computer science. In this section related work from the computer science literature is presented where theories and principles from psychology have been used.

In computer science, most concepts and solutions that deal with complexity management and that are intended to support understanding have

been developed without any psychological principles in mind. The theories and practices that have proven to be useful are often implicitly based on principles already known to cognitive psychologists. However, lots of information is left implicit and the improvement of existing techniques as well as the development of new techniques could be accelerated if the findings from psychology and the learning sciences were paid more attention.

Even though the advances of the cognitive and learning sciences have received very little attention in mainstream computer science, there are some research communities where interdisciplinary approaches are being used. In the following subsections the ones that are most relevant are shortly summarized. E-Learning systems are not mentioned in the summary, as for these systems the relations to learning theories are obvious. Furthermore, artificial intelligence is also not considered here as the goal of artificial intelligence research is to model or imitate intelligence, rather than the development of comprehensible computer systems.

1.6.1 Information Visualization and Human Computer Interaction

Information visualization is the use of interactive visual representations of abstract data to amplify cognition. It is about *external cognition*, that is, how resources outside the mind can be used to support the cognitive capabilities of the mind. Information visualization is an interdisciplinary topic involving both the machine side and the human side [War04].

Until recently, the field of information visualization has been more craft than science [Spe01, BS03], with many techniques not being based on sound scientific foundations [Ber81, Ber83, Tuf83, Tuf91] and rarely paying attention to cognitive principles. Many different visualization techniques have been developed, some of them very usable for certain applications, others less usable. In recent years, information visualization is developing more and more into a science by providing a scientific basis for information visualization concepts – also by paying attention to psychological theories. For example, various models of working memory and the integration of information stored in long-term memory are discussed extensively in the recent information visualization literature [War04].

Human Computer Interaction (HCI) is a similar field of research but it has a broader perspective. It is concerned with all aspects of interactions between humans and computers, which involves user interface design [Ras00], and also social aspects of the work environment [BB87]. One goal is to make the usage of computers understandable, for example so that even novices can use a system, or that critical failures displayed by the computer system are immediately recognized by the users. By definition, HCI is an inherently multidisciplinary field. Scientific work in this area usually draws on theories from psychology [CMN83].

1.6.2 Program Comprehension

When in the 1970s it had been recognized that most large programs undergo significant change during the in-service phase of their life cycle [BL76] and the problem of increasing complexity due to the development of ever larger software systems had become obvious, the field of *program comprehension*, has begun to develop. Program comprehension aims at supporting effective maintenance and successful evolution of software systems [Cor89, vMV95, BBH+95].

Brooks [Bro83] has developed a very influential theory for program comprehension. It is, in essence, a top-down strategy for experts who use their extensive programming knowledge together with domain knowledge to drive the comprehension process.

Von Mayerhauser and Vans [vMV95] have surveyed the most important code cognition models, most of which follow either a top-down or bottom-up comprehension model. Some models make use of research in cognitive psychology, but as the authors admit, the development of program comprehension models is still in the theory-building phase, relying mainly on observational experiments. There have been some correlational and hypothesis-testing experiments, but almost exclusively on very small-scale comprehension tasks of programs consisting just of a few hundred lines of code. An evaluation of the models with large-scale experiments of large applications in a professional environment has not yet been done.

Most code comprehension models draw on existing knowledge to build new knowledge about the mental model of the software that is under consideration: The understanding process matches existing knowledge with newly acquired knowledge until the programmers believe they understand the code. During code understanding, hypotheses are formed, they are checked for validity and revised if necessary. The key is to keep the number of open hypotheses manageable while increasing understanding. The understanding covers both programming concepts and the recovery of domain knowledge [BMW93, LY01, HGK04].

In recent years, some authors have tried to make use of theories of human learning, especially concerning the role of concepts in program comprehension. Rajlich and Wilde [RW02] have created a model that does not rely on the top-down vs. bottom-up dichotomy of the earlier authors. Their model is based on concept formation: Programmers tend to use an as-needed strategy to understand how specific concepts are reflected in the code. They are seeking the minimum essential understanding for a particular task. These concepts can either be domain-level concepts or programming concepts, such as *design patterns* [GHJV94]. An important aspect of program comprehension is to locate domain-level concepts in the code.

It has been shown that program comprehension can be supported by tools: Domain-level as well as programming concepts can be identified and dependencies can be found [AW95, BEW95, BK01a, BK01b].

The discipline of program comprehension is currently just starting to make use of theories of cognition and it is expected that this will lead to interesting insights and techniques [RW02]. The psychological rationale behind the design of present program comprehension tools is rarely described in detail and, generally, the rationale is poorly articulated, leaving lessons primarily implicit. *Cognitive support theories* [Wal02] of program comprehension must be developed, not merely shallow explanations at the technological level.

1.6.3 Software Engineering

Software engineers often have to handle complex problems and they sometimes create large systems. So complexity management is an important issue in software engineering. Object-oriented programming (OOP) has become popular in the last 15 years and has made it possible to develop larger software systems with more elaborate features than were possible before. One important aspect of OOP concerning complexity management has been the shift from a centralized to a decentralized control model with objects/classes communicating with each other instead of monolithic blocks of code. Patterns [GHJV94] and heuristics [Rie96] have been developed to support software engineers in creating comprehensible object-oriented programs. Most of the heuristics aim at distributing functionality evenly over various small, comprehensible parts that are loosely coupled with the other parts of the system. In general, the heuristics have not been created with cognitive principles in mind but they have proven to be helpful. They are implicitly based on the properties of the human mind, at least in part. A common approach to complexity management is information hiding, such the containment of an object in another object which means that the hidden information can be ignored at a higher level of abstraction. Another approach is the decomposition of a complex part into a number of collaborating components each of which is easy to understand and has just limited interactions with other components [Rie96].

An important aspect of OOP is the possibility to model the real world more closely, allowing the developer to make use of analogies and well-known concepts. For example, an object-oriented program implementing the functionality of an ATM machine can contain the classes `customer`, `account` and `withdrawal`. This is very helpful for understanding, as the concepts used in the program resemble those in the real world – see section 2.3.2 for a detailed discussion of object concepts.

1.6.4 Complexity Metrics

Complexity metrics can be used to assess the overall design of a system according to some properties and then use this assessment to perform corrective actions early in the development process [MB98, Kan03], or for reengineering purposes [KB98].

There exist various approaches to measure the complexity of software systems, most of which focus on program code complexity [HS96]. There also exists work on measuring the architectural complexity of software [KB98, MB98, Zha98]. The latter work focuses on the high-level structure rather than on the implementation details of any specific source module.

Complexity metrics can give an overview over some specific characteristics of a system or subsystem – those measured by the metric. However, the currently existing metrics in general do not take into account the cognitive complexity of individual tasks that developers or maintainers have to perform. For instance, a metric that measures the overall architectural regularity of a system may return a high value, which means that the system is highly regular and should thus be easy to understand. Nonetheless, tasks performed with this system, even "general understanding" tasks, may be very hard as the measure does not consider the complexity of the tasks to be performed. However, it is the cognitive tasks that the developers are performing, which must receive attention when making statements about the complexity of system development. Unfortunately, this problem has not yet been addressed in the literature.

A measure of cognitive task complexity from psychology – relational complexity – is presented in section 3.2.6. For such a measure to be applicable to development tasks, cognitive process models will have to be developed – see sections 3.3 and 7.3.

1.7 Overview

Chapters 2 and 3 present mostly non-technical information that represents the theoretical background of this thesis from cognitive psychology and the learning sciences: First, chapter 2 summarizes some relevant theories mainly from cognitive psychology, including theories of human working memory, long-term memory and concept formation. Then, chapter 3 reviews various theories that describe cognitive complexity and learning difficulties. Furthermore, it is discussed how the cognitive complexity of computer system design can be assessed.

Chapter 4 describes the technical background of this thesis. It summarizes the basic concepts and definitions around distributed embedded real-time systems. Then, chapter 5 uses the theories presented in chapters 2 and 3 to introduce complexity management techniques for embedded real-time systems development. These techniques are used for a detailed analysis of the architectural design concepts of the DECOS integrated architecture which follows in chapter 6. In this chapter it is shown how the theoretical considerations map to a real-world computer system architecture. Finally, chapter 7 summarizes the most important contributions of this dissertation, presents the conclusions that can be drawn and describes open issues that need further research.

Chapter 2

Cognitive Processes of Understanding

With the emergence of *cognitive psychology* in the last decades, much progress has been made concerning the development of theories of how our brain works. This chapter summarizes some fundamental cognitive processes and characteristics that are at the basis of understanding. A wide variety of factors that have an influence on learning and understanding have been identified. The author has tried to make use of the mainstream theories, building up a coherent picture of the common aspects, instead of pointing out the differences between the theories and the disputes of their advocates.

2.1 The Cognitive Perspective

In this section the scientific disciplines of cognitive psychology and the learning sciences are briefly introduced. Then, the concept of mental representations, which are at the basis of all cognitive abilities, is presented.

2.1.1 The Sciences of the Mind

Cognitive psychology emerged as a separate discipline in its modern form in the late 1950s [Mil56, BGA56, Bro58]. It is a branch of psychology that had a very deep impact on psychology as a whole, called the *cognitive revolution*, which has taken place in the 1950s and 1960s [Rei01]. Due to the shortcomings of the nineteenth-century movement that emphasized *introspection* and the following *behaviorist* movement, psychologists turned to a method in which one focuses on observable events, but then asks what (invisible) events must have taken place in order to make these (visible) effects possible. The *computer metaphor* is often used in this context where theories are based on an information-processing approach that became popular with the introduction of computers. The interdisciplinary research area that emerged during the cognitive revolution is called *cognitive sciences*. It covers

psychology, philosophy, linguistics, anthropology, neurosciences and artificial intelligence. Today, cognitive psychology can be seen as the science that plays the central role in the cognitive sciences, investigating human mental processes and their role in perception, thinking, feeling, and behaving.

The *learning sciences* are an interdisciplinary field that studies teaching and learning [Saw06]. They developed into a separate discipline in the late 1980s when researchers realized that new scientific approaches to learning had to be developed. It became clear that the single sciences that play a role in this area, such as the cognitive sciences, educational psychology, computer science, and sociology must be integrated. The goal of the learning sciences is to better understand the cognitive and social processes that result in the most effective learning. In this work those aspects and theories are considered, that affect the presentation of material and the learning process itself.

With the advances of cognitive psychology and the development of the learning sciences, much progress has been made: In the last years, theories have been developed and principles were discovered that are supported by a wide variety of experimental data. Although there is still much active research in the field and many issues remain unknown, some theories are quite well-accepted and can thus be considered sufficiently mature to be readily used in the technical sciences.

2.1.2 Mental Representations

The information processing approach in cognitive psychology is based on the assumption that an organism's ability to perceive, comprehend, learn, decide, and act depends on *mental representations*, which are unobservable internal codes for information [Kel03]. These internal representations of the human mind are usually contrasted with physical external representations. For example, you can imagine a dolphin which results in an image that only you can experience, or you can see a dolphin in the real world, which can also be seen by others. Mental representations are the basis of all cognitive abilities.

It is assumed that there exist different kinds of mental representations, for example verbal representations, or spatial representations. Depending on the task that is performed, different mental representations are used. An aspect that is very interesting for this thesis is that there exists evidence that spatial representations are used in both spatial and temporal reasoning [Gil04]. This view is supported by the concept of a time-line that is commonly used in technical sciences to model time and as an aid for understanding temporal problems.

The result of the processes that construct mental representations of the information available in the environment is called *perception*. The processes of perception organize and interpret the information. The recognition of many objects is influenced by the context in which the objects are en-

countered, thus perception is both a *bottom-up* (data-driven) and *top-down* (concept-driven) process [Rei01].

Mental representations are processed in stages, such as encoding the information, storing it in memory, retrieving it, and manipulating it.

Attention refers to the selection of certain stimuli for processing to the exclusion of others as well as for the concentration of mental resources on a particular process.

2.2 Memory

Cognitive psychologists use a three-store model of memory distinguishing among sensory, short-term and long-term stores. Sensory memory is the initial, unconscious and very short-lived recording of information from our senses. This kind of memory is not discussed in this work as the recognition of sensory information is not considered.

2.2.1 Working Memory

The concept of *working memory* has been developed in the 1960s and 1970s [MGP60, BH74]. Various models have been successful in the explanation of a variety of phenomena [Rei01, Ste04]. Working memory refers to the system or mechanism for temporarily maintaining mental representations that are relevant to the performance of a cognitive task. There exist different working memory models, e.g., with respect to whether working memory is a part of long-term memory that is activated for a particular task [Cow01, Kel03], or whether it uses separate structures [BH74, Bad98]. Fortunately, many of these unknown aspects are not relevant for the considerations in this dissertation. Many characteristics of working memory that are important for this work – and that a working memory system exists at all – are widely accepted in cognitive psychology [Bad98, MS99, Rei01, Gil04, Ste04]. The basic characteristics common to most models is that working memory is a system consisting of storage components, processes for achieving and maintaining activation, and controlled attention [MS99].

In this dissertation the working memory model of Baddeley and Hitch [BH74] is adopted as it is the most influential and well-known [MS99, Rei01]. It is also supported by a wide variety of experimental data [Bad98, Bad00, AGWA04]. For a detailed discussion of the differences and similarities of working memory models see [MS99].

The working memory system comprises multiple components [Bad00]:

Phonological loop: This component stores verbal representations and maintains active memory via rehearsal mechanisms of inner speech.

Visual-spacial sketch pad: This part is used to store visual and spatial

representations and to maintain active memory via rehearsal mechanisms of image generation or action preparation.

Episodic buffer: This is a limited capacity system that provides temporary storage of information held in a multimodal code. It is capable of binding information from the subsidiary systems, and from long-term memory, into a unitary episodic representation.

Central executive: This component constitutes the supervisory control of the working memory system. It coordinates the two storage components, focuses and switches attention and retrieves representations from long-term memory. Evidence suggests that the central executive is a task-general mental resource with limited capacity [Rei01].

What is important is that – like in most other models of working memory – the working memory system is not a simple storage container; it contains both processing resources and memory. It is at the base of all cognitive activities. As the mental resources of the working memory system are limited, this has important consequences on human cognitive performance.

A *chunk* is a collection of mental representations that have strong associations to one another and much weaker associations to other chunks currently in use. By grouping a number of items into a larger chunk, the working memory capacity limit can be stretched. A very simple example for chunking is to group meaningful information together: Instead of remembering the single numbers 1, 7, 7, 6, 2, 0, 0, 5, 1, 9, 4, 5 they can be grouped to the year dates 1776, 2005, 1945 which can be remembered far more easily as they only form three chunks instead of 12 chunks when seeing them as single numbers. Thus, a chunk does not hold a fixed quantity of information.

There exists lots of work investigating the capacity of working memory. One of the first and still one of the most influential papers is that of Miller [Mil56] who claimed that the capacity of the short-term memory is about seven chunks of information. This number was the result of experiments that investigated immediate recall of chunks of information, such as numbers or words. One reason for the popularity of the number of seven is probably that a single number is easy to understand and to remember. The number was simply transferred to other domains, where it was treated as a magic number, without any validation. This resulted in quite bizarre recommendations, such as to limit the number of function parameters to seven, not to put more than seven items on a presentation slide, and the like.

More recent research has investigated more complex tasks than just immediate recall of numbers. For more complex tasks than simply recalling a list of items, a working memory capacity limited to about four chunks of information seems more realistic [Cow01]. Also see section 3.2.6 for a discussion of relational complexity theory that uses the same number as the maximum number of chunks we can integrate mentally in a single task.

The limited capacity of working memory of both the processing and memory components is a major cause for limitations of cognitive tasks

[Kel03, Gil04]. When working memory is overloaded, errors occur [SC94, Gil04].

2.2.2 Long-Term Memory

Long-term memory allows the retrieval of stored mental representations even decades after they have been stored. It's capacity is unknown.

Current memory theories propose that long-term memory is not unitary. It can be subdivided into declarative and nondeclarative memory [Rei01, Kel03]. *Declarative (explicit) memory* refers to knowledge of events, facts, and concepts ("knowing what"). *Nondeclararive (implicit) memory* refers to skills and related procedural knowledge ("knowing how") [Kel03].

Declarative Memory consists of two subsystems: *Episodic memory* contains specifically dated occurrences of events in a particular context. *Semantic memory*[1], refers to factual and conceptual knowledge about the world.

Implicit learning refers to the unconscious acquisition of complex rules that cannot be verbalized. Implicit memories can be the consequence of processing fluency, produced by experience in a particular task and can be registered as a sense of "specialness" attached to a specific stimulus [Rei01].

2.2.3 Encoding, Storing and Retrieving Information

Encoding and storing information in long-term memory involves rehearsal [Rei01, Kel03]. *Maintenance rehearsal* refers to recycling information within working memory by covertly verbalizing it, with little thought about what the items mean, or how they are related to other bits of knowledge. *Elaborative rehearsal* draws on semantic features of the to-be-remembered information. Elaborative rehearsal can be, for example, organizing items into categories or associating them with information already known. This kind of rehearsal usually involves linking information in working memory with information already stored in long-term memory.

In analogy to elaborative and maintenance rehearsal, there are different depths of processing in learning: *Shallow processing* means to engage information in a superficial fashion, *deep processing* is to think about the meaning of the items. The deeper the processing, the easier the items are remembered since thinking about the meaning of items and categorizing them means that they are related to knowledge already in long-term memory by establishing memory connections. These connections serve as retrieval paths so that the learned items can be found later on. Thus, learning is not simply a matter of placing information in long-term memory. It also involves the creation of appropriate retrieval paths – otherwise the stored information becomes inaccessible, i.e., it is forgotten. The deeper the processing, the easier the information can be retrieved from memory, just doing maintenance rehearsal

[1] Also referred to as *generic memory* [Rei01].

does not lead to good retrieval performance [Rei01, Kel03].

From the perspective of storing and retrieving information in memory, *understanding* can be defined as *seeing connections to related knowledge*. A good understanding thus means that most or all relevant connections are known, whereas a shallow understanding means that just a fraction of all relevant connections are known.

When people have extensive knowledge about a specific area, this provides a framework on which new materials can be "hooked". So it is much easier for them to remember and understand the new information.

However, the same memory connections that are crucial for understanding can be a source of memory error called *source confusion* as some kind of interference can occur: When the new information is interwoven with similar prior knowledge, this can create confusion as one might lose track which elements belong to which concepts or events [Rei01].

Organization provides retrieval cues: It has long been known that well-organized information is better remembered. A learner must discover the relations among items or, when none are apparent, create his or her own subjective relations. The categories imposed by the materials or by the learner serve as highly effective retrieval cues [Kel03]. We remember poorly if we can neither find nor create an organizing scheme, but arbitrary organization is not very effective – we remember best, what we understand best: The optimal organization of complex materials is generally dependent on understanding [Rei01].

Chunking also helps to store and retrieve information in long-term memory: By integrating and unifying (i.e., understanding) the materials, one ends up with less to remember, reducing the load on memory [Rei01].

Learning serves best if, at the time of retrieval, the material is approached in the same way, i.e., via the same connections. It is much harder to locate the sought-for information via a different retrieval path, i.e., from another perspective [Rei01]. The *encoding specificity* principle says that information is recognized or recalled only when the retrieval cues at the time of test match the encoding cues at the time of learning. The retrieval cues enable the activation of the to-be-remembered information and its context. Even the environmental and psychological context has an effect on retrieval performance: If the environmental conditions (the same room, the same sounds, etc.) and the person's mood match the conditions and mood of learning, then the recall performance is significantly better [Kel03].

2.3 Semantic Memory

Semantic memory allows us to categorize the world. Without the ability to acquire, represent and use knowledge about meaningful symbols, high-level cognition would not be possible. Whenever we reason about *kinds*, such as chairs, cars, or dogs, we are employing categories. We are also using

categories if we intentionally perform any *kind* of action (e.g., "going to a supermarket"), which again consists of *kinds* of motor activities (as they are never exactly the same) [Lak87, Kel03].

Concepts are the general ideas that enable the categorization of unique stimuli as related to one another. The stimuli categorized may be concrete objects or abstract ideas [Kel03]. In this section some fundamental theories of concept formation are summarized. These theories try to explain how we categorize the world around us and how we use our conceptual knowledge for reasoning. *Reasoning* is usually defined as the process of drawing conclusions [Lei04]. Conclusions are drawn in problem-solving and decision-making tasks.

2.3.1 Rule-Governed Concepts

The classical approach of categorization assumes that concepts can be defined by a set of singly necessary and jointly sufficient rules [Kel03]. This means that for every concept a set of defining features can be provided. Some abstract concepts, especially in mathematics and physics, can be seen as *rule-governed*. However, for many concepts it is quite impossible to find a set of defining features, e.g., for the concept of "dog", so that all dogs fit to the concept, whereas all other animals that are no dogs do not. The problem here is that there is a fuzzy boundary of class membership [Ros78, LM99, Kel03].

2.3.2 Object Concepts

Object concepts refer to natural kinds or biological objects, and artifacts or human-made objects. They are often organized hierarchically. For example, a parrot and a dove are both birds.

A flexible boundary can usually be observed for object concepts, depending on which other concepts are active in memory [Lab73]. For example, whether we perceive object B shown in figure 2.1 as a cup or a glass heavily depends on the context in which it is shown.

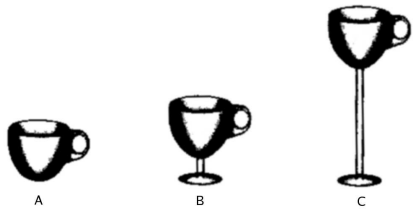

Figure 2.1: A cup or a glass or something else? Source: [Lab73]

Object concepts show a *family resemblance*, which means that the category is not defined by a small set of defining features, but by a quite large

number of features that apply to some but not all instances. For example, it is hard to define a dog by a complete set of characteristics that really apply to all dogs we will ever see. There will always be some special dogs that do not have some characteristics – as, for example, an ill dog that cannot bark is still a dog... So there is a difference between *typicality* and *category membership*. Even though typicality matters considerably in our judgments, typicality is not crucial for category membership. What seems more important is that a case has the *essential properties* for the category: If those properties are present, a case can be in the category even if it is highly atypical; if they are not present, a case is likely to be excluded from the category. For example, object A in figure 2.1 will not be perceived as a wine glass as it lacks a stem and a foot which are typical for wine glasses, while object C might easily be perceived as a wine glass – even though it is not a typical wine glass due to its handle. The importance of a property depends on our beliefs about what matters for that category [Rei01].

Human categorization is not the arbitrary product of historical accident or of whim, but rather the result of psychological principles of categorization. Two basic principles have been proposed [Ros78]:

Cognitive economy: The function of category systems is to provide maximum information with the least cognitive effort.

Perceived World Structure: Maximum information with least cognitive effort is achieved if categories map the perceived world structure as closely as possible.

These two basic principles have implications both for the level of abstraction of categories and for the internal structure of categories: Categories can be seen as having a vertical dimension concerning the inclusiveness of the category, and a horizontal dimension concerning the segmentation of categories at the same level of inclusiveness. The implication of the two principles of categorization for the vertical dimension is that not all possible levels of categorization are equally useful. The most basic level will be the most inclusive (abstract) level at which the categories can mirror the structure of attributes perceived in the world. Thus, the term *level of abstraction* within a taxonomy refers to a particular level of inclusiveness. The implication for the horizontal dimension is that if the distinctiveness of categories is increased, they tend to become defined in terms of prototypes. Cognitive economy dictates that categories should be as separate from each other and as clear-cut as possible [Ros78].

The *prototype* is the most typical member of a category. Categorical judgments become a problem if one is concerned with class boundaries. The *basic-level categories* are the ones that are the most useful for the given context. They are usually first created and learned, spreading both upward and downward as taxonomies increase in depth. The basic level provides the cornerstone of a taxonomy [Ros78]. For example, in the area of distributed embedded real-time systems, the concepts of *state*, *cycle*, *messages*,

and *components* represent the basic level for reasoning at the system level [Kop08]. These concepts can be refined for more detailed considerations at lower levels, or abstracted for considerations at higher levels.

Concepts usually cannot be characterized in isolation; they often have interrelationships with other concepts. There exists a wide network of beliefs in which concepts are embedded. The beliefs describing a category in relation to various other concepts are referred to as *mental models* or *implicit theories* [Rei01, JL04]. For instance, to be able to understand the term *widget*, one needs a basic understanding of what a graphical user interface of a computer is, which in turn requires one to know what a computer is, and so on. The list of concepts that is needed can be extended almost endlessly. This is a basic characteristic of nearly all concepts which has consequences on all semantic descriptions: Most definitions and specifications – especially on the semantic level – cannot be complete. They almost always require some background knowledge to be interpreted meaningfully.

Prototypes and exemplars can serve as *categorization heuristics*, which allow efficient and usually accurate judgments about category membership. However, as with all heuristics, such judgments may not be adequate in all circumstances [Rei01].

2.3.3 Schemata and Concept-Driven Processing

A *schema* [Bar32] is an integrated chunk of knowledge that organizes related concepts and integrates past events. A schema allows us to form expectations and to form inferences. So schemata are important in understanding the constructive aspects of perception and memory. *Schemata* shape how information is retrieved from long-term memory. A schema can be seen as a set of organized concepts that provides expectations about the world. There exist conceptually driven processes that guide the reconstruction of information from memory. Details may be assimilated or normalized so as to fit the expectations provided by the schema. Schemata also influence how information is encoded by establishing expectations that result in the selection of features that are encoded [Rei01, Kel03]. A *kitchen-schema* makes us expect that a stove and a refrigerator are likely to be present, whereas a bed is not likely to be present.

When reading a text, the concepts and schemata of semantic memory provide the reader with a personal interpretation. Propositions contributed from the reader's long-term memory rather than from the text itself constitute the *situation model* [Kel03].

Schema theory is sometimes contrasted to mental model theory (see section 3.2.7). A main reason for criticism is the rather static nature of schemata, as compared to mental models which support changes more easily. However, the differences between the two theories are not relevant for this thesis. The relevant aspect that is supported by both theories is that human reasoning is characterized by a large amount of concept-driven processing.

So the knowledge from long-term memory influences how we perceive the world and how we solve problems. In the literature both terms – schema and mental model – are used to refer to a mental representation that organizes knowledge about related concepts.

Once a schema or a mental model is formed, it can replace carefully considered analysis to increase processing performance. Hence, concept-driven processing provides automatisms ("short-cuts") at the conceptual level.

2.3.4 Abstract Thinking

Thinking means to manipulate mental representations. This definition covers problem solving, as well as reasoning and decision making [Kel03]. Abstract human thought is possible only because of the conceptual representations that represent the building blocks of semantic memory.

Human memory can not only represent reality quite accurately, it also provides the flexibility to distort reality to see the world in different ways. A *meta-representation* is a mental representation of another mental representation. This allows us to think about other thoughts. Thus, there are primary representations representing objects in the world which must be as accurate and literally correct as possible, and adaptive meta-representations that add a high degree of flexibility and creativity to cognition [Kel03].

2.4 Expertise

Experts are individuals who have learned lots of factual and conceptual knowledge and have developed the necessary procedural skills to excel in a specific domain of tasks [Kel03].

2.4.1 Expert/Novice Differences

The basis of expertise seems to be that experts organize their knowledge around deep principles of a domain that represent key phenomena and their interrelationships [HSP04]. They can differentiate easily between relevant and less important information.

As a result of extensive practice, experts are able to retrieve information from memory more effectively than are novices. They have highly specialized retrieval structures to gain access to what they know. These structures allow experts to anticipate what they need to remember and to encode the relevant information in a format that ensures later retrieval (elaborative rehearsal). An expert's organization of long-term memory guides the encoding of information into meaningful chunks [Kel03].

According to schema theory [Bar32, SC94], schema acquisition and automation are major learning mechanisms. Experts have acquired a large

number of schemata and can use them very efficiently – irrelevant details can be ignored and the key phenomena can be focused. Novices do not have a large repertoire of suitable schemata in long-term memory. So in essence, schema theory also says that elaborate retrieval and structures and reasoning strategies account for the differences between experts and novices.

Experts in a domain solve problems in a qualitatively different fashion from that of novices. Novices engage less in *metacognition*, i.e., they do not reflect so much on different strategies to solve a problem and use any information that is at hand. Experts think through problems carefully before taking any steps toward solving them [Kel03, MG03]. Novices often rely on perceptually available, static components of a system, whereas experts integrate structural, functional and behavioral elements [HSP04].

The differences in problem solving between experts and novices can also be explained in terms of complex systems concepts, such as emergence characteristics, multiple agents, hierarchical levels, etc. These are, in essence, the properties found to be difficult to understand (see chapter 3). Experts use more of these complex systems concepts than do novices, which allows them to acquire a better and more accurate understanding than novices who mostly exhibit a quite reductive understanding [Jac01, HSP04].

Novices tend to rely on *folk theories*, which are naive commonsense explanations of phenomena as opposed to scientific facts.

Complex systems often involve concepts that are in conflict with learners' prior experience. Most people prefer explanations that assume central control, single causality and predictability [RW98, Jac01].

Furthermore, experts use more sophisticated reasoning strategies and don't rely on heuristics used by novices. This results in less errors as the situations where the heuristics are not applicable can be recognized in advance [Rei01]. For example, the *recognition heuristic* is often used if the only information available is whether an option has ever been encountered before. If this is the case, then we typically choose the known option, and not an unknown option. Detailed discussions of cognitive heuristics can be found in [MTG04, Rob04].

The *power law of practice* says that the reaction time decreases as a power function of the degree of practice [Log88]. Within their specific domain, experts perceive, remember, think, and behave considerably faster than do novices.

2.4.2 Becoming an Expert

The amount of practice is a critical factor that can lead to excellence. Thus, we cannot become experts in many different domains, as extensive deliberate practice of many hours per day is not possible for various domains.

In the literature ten years of persistent deliberate practice is generally assumed as the minimum time to become an expert. This is called the

ten-year-rule [Kel03]. This rule implies that expertise is not a short-term solution to many problems and real experts in a specific domain are usually rare. Thus, regarding computer systems development, the topic of this thesis, a large number of non-experts will usually have to do the job. This does not mean that the developers are complete novices, but they are not experts according to the explanation above.

2.5 Chapter Summary

Mental representations are the basic internal codes used in our brain. A chunk is a collection of mental representations that can hold different quantities of information.

Working memory is the system for temporarily storing mental representations that are relevant to the performance of an active cognitive task. It contains processing resources and memory. Its limited capacity is a major cause for limitations of cognitive tasks.

Long-term memory allows the retrieval of information even decades after it has been stored. Learning involves the creation of retrieval paths so that the stored information can be found later on. It is especially important that connections between related knowledge are created, so that a network of concepts and beliefs is formed. From this perspective, understanding means seeing connections to related knowledge. We remember best what we understand best.

Concepts are the general ideas that allow to categorize related chunks of information. A fuzzy boundary of class membership can be found for many concepts. There is a family resemblance between the members of a category, but usually there is no set of defining features. Depending on the context, an object may be seen as a member of different categories. There may exist essential properties for a category that must be present for an object so that it is perceived as a category member. Quite often, categorization heuristics are used.

Concepts usually cannot be characterized in isolation as there exist networks of beliefs around a category which form relations to other categories. These beliefs are called mental models or schemata.

Categories can be seen at different levels of abstraction, which refers to a particular level of inclusiveness. The basic-level categories are the ones that are the most useful for a given context and provide the cornerstones of a taxonomy. Abstract thinking is supported by meta-representations that provide the flexibility to see a given concept at different levels of abstraction.

Concept-driven processing is an inherent feature of the human mind. Our knowledge from long-term memory influences how we perceive the world and the ways we think. Details may be assimilated or normalized so as to fit the expectations provided by the schema.

Expertise in a particular domain can be acquired by extensive practice over many years. Within their domain, experts perceive, remember and think considerably faster than do novices. An important difference between novices and experts is that the extensive knowledge in long-term memory supports various cognitive tasks: Experts have highly specialized retrieval structures that support access to their knowledge. This speeds up processing and ensures that the relevant knowledge can be found. Moreover, experts organize their knowledge around the core principles of the domain, which supports the separation of relevant information from irrelevant detail. The use of appropriate heuristics also plays an important role in expert reasoning, whereas novices are more prone to rely on folk theories.

Chapter 3

Cognitive Complexity

This chapter outlines what makes concepts and tasks either simple or complex by reviewing relevant literature from the cognitive and learning sciences. In the literature, quite different views on complexity exist. Some authors use a task-based view, while others follow a material-based view. As already outlined in section 1.2.2 a task-based ("problem-based") view is needed when making judgments about cognitive complexity, as a system per se cannot be complex – it is just the tasks that must be performed, that can exhibit a certain degree of cognitive complexity. The tasks of course follow from the system characteristics, so the two viewpoints must be integrated and used in combination.

Human mental performance also depends on individual differences and other factors, such as age, motivation, as well as social and cultural aspects. In this dissertation, a focus is put on the material that must be processed and on the tasks that are performed, discussing human mental processes in the area of computer system design and understanding. These processes and characteristics affect all of us to a similar extent. So general principles for design can be derived, without restriction to a special class of people.

Unfortunately, there is no unified theory about cognitive complexity. There exist just various factors and characteristics that can be described by individual psychological theories. It seems that all those aspects must be taken into account when assessing the cognitive complexity of tasks.

First, section 3.1 describes the most important cognitive characteristics that affect task performance, thereby discussing cognitive limitations that cannot be overcome, e.g., by training, and characteristics that are affected by the level of expertise of an individual. Then, section 3.2 integrates the cognitive limitations mentioned in section 3.1 with problem (or system) characteristics. Furthermore, theories of cognitive complexity are summarized, that all suggest that cognitive overload is the main reason for complexity. Finally, some considerations about assessing and measuring cognitive complexity are presented.

3.1 Cognitive Characteristics

The characteristics of our brain and our way of thinking influences what kinds of errors are likely to occur and what kinds of tasks we find either difficult of simple. In the following subsections the most important factors that affect us all to a considerable extent are summarized. Factors such as motivation and attention are very important, too, but they are not considered in this work as they rather represent topics for learning environment development. Their detailed consideration would be far beyond the scope of this thesis.

3.1.1 Limits of Working Memory

Reasoning and problem solving act on the transient contents of working memory. As discussed in section 2.2.1, the working memory system is a limited resource system. So these limits also impose limits on our reasoning and problem solving capabilities. There exist many examples where peoples' failures with complex tasks can be attributed to limited working memory capacity: Readers of a text have to resolve ambiguities of interpretations. Large working memory capacity has been found typical for good readers [Kel03]. Similarly, when solving a problem, people often forget or fail to represent all possible combinations of premises [Kel03, Gil04]. Moreover, making connections among different levels of a complex system places added demands on working memory, especially for systems characterized by complex causality, i.e., systems where there are intermediate steps between cause and effect [PG00].

There exist many theories in psychology and the learning sciences that relate the degree of complexity of a problem to the amount of working memory that is needed – see sections 3.2.5, 3.2.6, and 3.2.7.

3.1.2 Biased Thinking and Heuristics

Another important source for reasoning errors are various kinds of biased thinking, which are typical for humans. We usually try to avoid thinking in ways that contradict prior knowledge. For example, *belief bias* refers to people accepting any and all conclusions that happen to fit with their network of beliefs [Kel03].

We can differentiate into implicit and explicit reasoning processes. The former are automatic, associative and pragmatic in nature, and the latter are slow and sequential, load working memory, and potentially enable us to reason in an abstract logical manner [EF04]. So it depends on the used reasoning mechanism whether our thinking is influenced by belief bias or follows from logic conclusions. If we do not attend to all premises, or if we do not know all premises, then we are very likely to use more implicit than explicit reasoning processes. So it is very important to have a solid

foundation of understanding before drawing conclusions.

The use of heuristics is an inherent characteristic of human reasoning. Heuristics usually serve as shortcuts to reduce the cognitive processing demands. Without the use of heuristics, many cognitive tasks would not be possible due to overwhelming resource requirements [MTG04]. Sometimes, shortcuts are unconscious and implicit procedures, but often they are also applied deliberately [Rob04]. Heuristics not always yield correct results, but often they yield satisfactory results very fast.

3.1.3 Probabilistic Reasoning

Especially when we have to perform tasks that involve *probabilistic reasoning*, we are very likely to make errors because we do not pay attention to relevant information or give undue importance to irrelevant information [TK74, MTG04].

Good performance with probabilistic reasoning can only be achieved if the tasks are clearly described in a causal structure and the statistics clearly map onto that structure [KT07]. In other words, probabilistic reasoning tasks must be presented in certain ways so that judgment errors can be avoided. If tasks are presented in an arbitrary form or are embedded in a setting where lots of task-irrelevant information is available, errors are very likely.

3.1.4 Oversimplification

Humans show an inclination to construct overly simplistic understandings and categories. This reductive tendency is an inevitable consequence of the learning process [FRR04]. When one acquires new knowledge, such as the creation of a new category, or the forming of an new understanding, the knowledge is necessarily incomplete. So, *at any point in time, a person's understanding of anything that's at all complex, even domain experts' understandings, is bound to be simplifying at least in some respects* [FRR04]. However, these necessary simplifications may persist even if there is evidence that they are wrong. If learners are confronted with evidence contrary to their views, they often *perform mental maneuvers to rationalize their faulty beliefs without fundamentally altering them* [FRR04]. So oversimplification is supported by our tendency towards belief bias.

3.1.5 Dependency on Domain Knowledge

Many studies have shown that domain knowledge is power for reasoning and problem solving [Kel03]. Knowledge of the essential concepts of a domain is at the core of high-level cognition. Without that network of basic concepts, understanding is not possible – abstract explanations will fail to be understood.

3.1.6 Dependency on Metacognition

Monitoring of cognitive activities is an important issue, especially for complex tasks: Metacognitive skills allow to monitor progress in solving a problem. Good thinkers are able to evaluate problem-solving efforts and abandon an unproductive way of representing or searching a problem space and looking for alternatives. Poor thinkers lack these metacognitive skills [MG03, Kel03].

3.2 Characteristics of Complex Problems

Various characteristics of complex problems have been identified by authors from the learning sciences community and by cognitive psychologists. Unfortunately, there exist quite different viewpoints, but all of these views seem to be part of the whole story.

3.2.1 Conceptual Complexity

The characteristics of concepts have a severe influence on how difficult they are to comprehend. Complexity minimization plays an important role in human concept learning [CV03]: Concept learning involves the extraction of a simplified (abstracted) generalization from examples. This means that we try to find patterns that provide the simplest explanation of available data. For example, high-level cognition involves finding patterns in information to extract categories, and to infer causal relations.

There exists evidence that inductive concepts are represented in terms of the *regularities* (patterns in the observed examples) that they obey [Fel03]. Maximally complex concepts have no common regularities. The most efficient way to store them is item by item, as exemplars. This view of complexity is closely related to *Kolmogorov complexity*, which defines complexity of a string as the length of the string's shortest description in a given description language. So complex concepts are incompressible, whereas simple concepts can be compressed. If the categories covered by a concept differ widely, the concept is more difficult to learn than if the categories are very similar.

3.2.2 Dimensions of Difficulty

Research in the area of medical education identified characteristics of learning material that cause cognitive difficulty for learners. The characteristics have also been confirmed in the area of complex socio-technical systems [FRR04]. These "dimensions of difficulty" provide a basis for evaluations of computer systems architectures regarding cognitive complexity. The characteristics are as follows [FCS01, FRR04]:

Static vs. dynamic: Are important aspects captured by a fixed "snap-

shot", or are the critical characteristics captured only by the changes from frame to frame? Are phenomena static and scalar, or do they possess dynamic characteristics?

Discrete vs. continuous: Do processes proceed in discernible steps, or are they unbreakable continua? Can we describe attributes by using a few categories, or must we use continuous dimensions or many different categorical distinctions?

Separable vs. interactive: Do processes occur independently or with only weak interaction, or do strong interaction and interdependence exist?

Sequential vs. simultaneous: Do processes occur one at a time, or do multiple processes occur at the same time?

Homogeneous vs. heterogeneous: Are components or explanatory schemes uniform or similar across a system, or are they diverse?

Single vs. multiple representations Do elements in a situation afford single or just a few interpretations, functional uses, categorizations, or do they afford many? Do we need multiple representations (such as multiple perspectives, schemata, analogies, models, or case precedents) to capture and convey the meaning of a process or situation?

Mechanism vs. organicism: Are effects traceable to simple and direct causal agents, or are they the product of more system-wide, organic functions? Can we gain important and accurate understandings by understanding just parts of the system, or must we understand the entire system to understand even the parts well?

Linear vs. nonlinear: Are functional relationships linear or nonlinear (that is, are relationships between input and output variables proportional or nonproportional)? Can a single line of explanation convey a concept or account for a phenomenon, or does adequate coverage require multiple overlapping lines of explanation?

Universal vs. conditional: Do guidelines and principles hold in much the same way (without needing substantial modification) across different situations, or does their application require considerable context sensitivity?

Regular vs. Irregular: Does a domain exhibit a high degree of regularity or typicality across cases, or do cases differ considerably even when they have the same name? Do concepts and phenomena exhibit strong elements of symmetry and repeatable patterns, or is there a prevalence of asymmetry and an absence of consistent patterns?

Surface vs. deep: Are important elements for understanding and for guiding action delineated and apparent on the surface of a situation, or are they more covert, relational, and abstract?

Research found out that people often deal with complexity through oversimplification, which leads to misconceptions and errors [FRR04], as described in section 3.1.4.

3.2.3 Emergent Views

The concept of *levels* of description can be used to characterize a system with lots of interacting parts [WR99]. The higher levels arise from interactions of objects at lower level. E.g., a traffic jam can be seen as a phenomenon on a higher level than the single cars that enter and leave the jam. This *emergent view* of levels is fundamental to scientific theory. Emergent phenomena typically can be conceptualized as having two levels, a macro or aggregate level, and a micro or individual objects level. The macro level is an orderly pattern that can often (although not necessarily) be perceived; but there is no mechanism at that level that is directly responsible for that pattern. Instead, a collection of local interactions among the individual objects allows the pattern to emerge.

Emergent phenomena are characterized by new properties at the aggregate level that cannot be derived directly from the properties at the micro level. For example, the properties of a diamond, such as brilliance or hardness, are substantially different from the characteristics of its constituent atoms.

It has been shown that confusion of levels is the source of many of peoples' misunderstandings, not only in the study of science, but even in everyday life [WR99]. Humans have difficulties in understanding processes with an emergent structure [Chi00]. Robust misconceptions often arise when people treat emergent phenomena as causal events.

As today there is a great need to develop systemic approaches for designing and understanding the world, a deep understanding of the concept of levels is crucial for developing such systemic approaches and understanding the *sciences of complexity* [NP89], where complex phenomena can arise from simple components and simple interactions. The ability to shift levels, viewing the same object as either singular or plural, depending on the situation, is a prerequisite for building deep understandings of scientific phenomena. However, for systems development tasks that shall be as simple as possible, reasoning about emergent phenomena must be avoided.

3.2.4 Complex Causality

According to some authors, difficulties in understanding scientific or elaborate technical concepts is caused by misunderstanding causal relationships [PG00]: Most learners have a limited model of causality. Their simple styles of causal modeling contrast with the esoteric character of scientific models, which is referred to as *complex causality*.

Thinking in causal series instead of causal nets is another common prob-

lem [Rea90]: People have a tendency to think in linear sequences. They are sensitive to the main effects of a certain cause, but remain unaware of the side-effects on the rest of the system. So they only see influences within the narrow sector of their current concern.

The problem of biased thinking and oversimplification is often found when complex causality is not fully understood: Learners often have little experience or comfort with *epistemic moves* [PG00] such as remaining alert to gaps in causal relations or seeking disconfirmation for theories – moves that lead to more complex models.

3.2.5 Cognitive Load

Cognitive load theory [SC94] claims that material that is difficult to understand is characterized by high element interactivity. High interactivity means that there exist many relevant relations between the elements that must be integrated, resulting in high working memory load. We are forced to process elements simultaneously when they interact and cannot be considered in isolation. Elements may interact either because of the intrinsic structure of the elements, or because of the manner in which they are presented. In this view, learning difficulty is not just a function of the number of elements that must be learned, but also a function of the number of elements that must be learned simultaneously.

The authors of the cognitive load theory suggest that learning mechanisms have the primary function of circumventing our limited working memory and emphasizing our long-term memory. According to the theory, schema acquisition and transfer from controlled to automatic processing are the major learning mechanisms that reduce the burden on working memory. Extraneous activities that are not directed to concept acquisition and automation must be minimized. The cognitive load heavily depends on the knowledge of the learner as chunks of information can be formed more efficiently if underlying concepts are already known. A chunk for one person may be several dozen elements for another.

The original cognitive load theory is mainly based on schema theory. However, the view that the presence of irrelevant information can hinder task performance is also supported by experiments around mental model theory [Gir04, JL04]. The way information is presented is of great importance for good comprehension results.

3.2.6 Relational Complexity

Relational complexity is a theory from cognitive psychology that is supported by a wide variety of empirical data [HWP98, HBMB05, Bir02]. According to the theory, the processing load of a cognitive task is determined by the complexity of the relations that must be processed in a given step. The *arity* of a relation describes its dimension, i.e., the number of independent

elements that must be considered simultaneously. For example, BIG(dog) is a binding between the unary relation BIG and one argument. The relation can be interpreted as expressing a state or an attribute. Class membership, e.g., DOG(fido) can also be expressed as a unary relation. A binary relation such as BIGGER(dog,mouse) relates two components. Univariate functions and unary operators can be represented at this level. Ternary relations are needed to represent bivariate functions and concepts such as transitivity or class inclusion. Formal similarity mappings, independent of content, can also be made at this level. Greater abstraction is thus related to higher dimensionality [HWP97]. Quaternary relations are the most complex we can handle [HBMB05]. At this level four-way comparisons are possible. An example for a quaternary relation is a matching task of objects according to four independent attributes, such as color, shape, filling pattern, and orientation.

Relational complexity theory proposes that the cognitive demand can be reduced through conceptual chunking and segmentation [HWP98]. *Conceptual chunking* means recoding of a relation to lower dimensionality. For instance, velocity can be considered as a function of distance and time ($v = s/t$), which is a ternary relation. It can, however, also be considered as a unary relation if it is conceptualized as VELOCITY(50km/h). The chunked variables distance and time become inaccessible with this representation, i.e., they cannot be considered. *Segmentation* means to reduce problems of high dimensionality into a number of tasks of lower dimensionality that can be solved serially. However, not all relations can be decomposed into simpler relations and then recomposed into the original relation. Even if relations are decomposable, people may not have the necessary strategies. This explains that higher cognitive processes often depend on expertise [HWP97].

3.2.7 Complex Mental Models

Mental model theory [Gir04, JL04] postulates that individuals use the meaning of assertions and general knowledge to construct *mental models* of the possibilities under description. Models are mental representations that represent states of affairs, real or imaginary.

The theory assumes that each mental model represents a possibility. So *a conclusion is necessary if it holds in all the models of the premises, probable if it holds in most of the equipossible models, and possible if it holds in at least one model* [JL04].

Mental models are *iconic*, which means that reasoners can draw conclusions that do not correspond to any of the assertions used to construct the model. This characteristic seems to correspond to the technique of conceptual chunking of relational complexity theory.

An important characteristic of mental models is that a reasoning problem becomes harder with the number of mental models (or model interpretations) [Gir04]. For example, when an interface lacks coherence such that

a user can develop two or more mental models of the interface, the user is forced to deal with both possible interpretations [Ree99].

We are also more likely to err by overlooking a model if a high number of models must be considered for a given task. Similar effects can be observed when people have to handle inconsistencies: The consistency of a set of assertions is checked by searching for a single model in which all the assertions are true [JL04]. So the larger the number of assertions that is relevant for a given problem, the harder the creation of a mental model becomes. Consistency of the premises is essential for the simple creation of mental models. Ambiguity and uncertainty require the construction of larger models.

3.3 Assessing Cognitive Complexity

As already summarized in section 1.6.4 there exists a wide variety of different metrics that try to measure the complexity of software systems.

Metrics that deliver some value for a system might be of interest to project managers to assess the state or effort of a project, but such metrics measure something different than the cognitive effort that is actually required for a specific task. So instead of trying to measure the overall complexity of some system, it seems to be more reasonable to consider the complexity of the tasks that must be performed. Even a subsystem that rates very high on a given complexity scale might support some tasks that are quite simple.

As the development tasks are the central concern of a system development, it is these development tasks that must be subject to the considerations about cognitive complexity.

This section discusses the problem of assessing the cognitive complexity of computer systems development tasks. First, possible approaches for assessment of well-defined and of design problems are considered. Then, two different viewpoints where the assessment of complexity is helpful are presented: the component interface level and the component implementation level.

3.3.1 Well-Defined Problems

A cognitive process model can be developed for a well-defined problem. However, this may be hard – especially for problems that draw on existing knowledge as every individual person has a different knowledge base and might thus use different cognitive processes. For very isolated problems, cognitive process models could in theory be developed, but this is still an open research issue – see section 7.3. Some simple models have been developed for isolated domains, such as program comprehension (see section 1.6.2), but these models are far too general and leave too many aspects open

to serve as a basis for analysis of a wide variety of development tasks.

An experimental approach seems to be more promising than a purely analytic approach. We can compare problems that require specific development tasks and then measure the performance of the participants. What is important here is to avoid comparing apples and oranges. It does not make sense to compare completely different problems and then come to the conclusion that the one is far more difficult than the other. As we are interested in aspects that make a problem either simple or difficult, we have to vary the problems just in the given aspect we want to investigate.

3.3.2 Design Problems

As a design problem involves invention and so draws on unknown aspects, the development of a detailed cognitive process model is not possible. Thus, a fully analytic approach is not even a theoretic option in this case.

Regarding computer system development, the characteristics of the underlying system influence the design tasks that must be performed. For example, if a system is implemented according to a given architecture, these architectural characteristics restrict and guide the design in certain ways.

A design problem can thus be influenced by restricting its freedom. This means that the design process is guided into a particular direction by prohibiting, or at least by discouraging some possibilities. In this way it should not just be possible to guide the design of components into directions that have desired technical characteristics, but also into directions with good characteristics regarding comprehensibility. These characteristics can pay off during all phases of the product lifecycle. See section 5.3 for examples of such characteristics.

Experiments can of course also be performed with design tasks. The design freedom, i.e., the invention that is necessary, does not restrict the evaluation of measures such as required time, and fitness of the solution according to a given definition. What is problematic with design experiments is to assess the quality of the design if it is not easily possible to test it according to a given specification. Thus, experiments that involve design problems can only be evaluated objectively if a well-defined test criterion for the fitness of the design exists.

3.3.3 Component Interface Level

The component interface level is the level of abstraction of a component as seen by the component integrator. Numerous properties have been identified that contribute to the complexity when having to understand a component interface. However, the development of accurate complexity metrics is still in its infancy [HS96, GeA04]. A major drawback of existing metrics is that they do not consider chunking aspects, even if they are intended to measure cognitive complexity. For example, the behavior of a function that relates

three conceptually completely different parameters may be much harder to understand than a function that uses three parameters that are more or less the same and that are split into tree parts just as an accidental characteristic. But even if the complexity of interface comprehension often cannot be measured in terms of concrete numbers, complexity management at the component interface is an issue of utmost importance to be able to use, integrate, and re-use a component. Some important aspects of good interface design are presented in section 5.3.8.

3.3.4 Component Implementation Level

This is the level of abstraction that is required for the component developer. At the implementation level the internals of a component must be considered. Depending on the chosen methodology, either just the component itself, or also relational aspects to other components can be relevant.

To acquire an understanding of a component we have to consider the program code and its run-time behavior. The run-time behavior may require knowledge about the job scheduler, potential resource conflicts, ordering constraints, communication to other components, timing aspects and the relations to relevant real-time entities. So the behavior also includes the relationship between the component and its environment, as usually many relevant real-time entities are situated outside a component.

The complexity of the implementation or maintenance of program code alone is not considered in this thesis as this is not specific to embedded real-time systems architectures, but a general software engineering issue. However, the influence of different system structures, communication mechanisms, and models of time are discussed in detail in chapters 5 and 6.

3.4 Chapter Summary

There exists no unified theory about cognitive complexity. Some authors follow a problem-based approach, where the complexity is related to a certain problem (cognitive task). Others have identified system characteristics that can make a system hard to understand. As the tasks that have to be performed with a system heavily depend on the characteristics of the system, these two approaches do not contradict each other; they only represent different viewpoints.

The limits of working memory are a major limiting factor of our cognitive abilities. Many theories relate the complexity of a problem to the amount of cognitive resources that are required.

Conceptual complexity can be related to the compressibility of the categories that are covered by the concept: If the categories are incompressible, the concept is harder to understand than if the categories contain lots of regularities.

Another major factor that influences the performance of cognitive tasks is domain knowledge. The more domain knowledge a person has, the better is usually the performance of cognitive tasks that are related to the domain. Cognitive heuristics leading to reasoning errors are used if we have insufficient domain knowledge. Biased thinking and oversimplification are factors that mainly depend on prior beliefs.

Various dimensions of difficulty have been identified. The difficult side of these dimension is can be categorized into a high level of variety, interactivity, simultaneity and dynamics. In addition, emergent phenomena that require a thinking at different levels of abstraction seem to be especially hard to understand. Moreover, complex causality, such as causal nets are another common problem.

Cognitive load theory also says that a high degree of element interactivity is a major reason for complexity. The theory suggests that we can circumvent our limited working memory by a transfer from controlled to automatic processing, i.e., the automation of tasks so that the load on working memory is reduced. The presentation of the material can heavily affect task performance. Tasks should be reduced to essential aspects for optimal processing.

Relational complexity theory focuses on the detailed characteristics of complex tasks. According to the theory, the processing load is determined by the arity of conceptual relations that must be processed in a single step. Quaternary relations are the most complex we can handle. Conceptual chunking means to recode a relation to lower dimensionality. The chunked relations become inaccessible. Segmentation means to reduce problems of high dimensionality into a number of tasks of lower dimensionality that can be solved in a series. The used segmentation and chunking strategies are influenced by the level of expertise.

Reasoning problems that involve multiple mental models are harder than problems with just a single model. Moreover, consistency of the premises is essential for the easy creation of mental models. So deterministic models are generally easier to understand than those that involve non-determinism.

The development of metrics that can measure how difficult we find it to understand a computer system, or even just a piece of program code, is still in its infancy. It is yet unclear if it will ever be possible to develop effective measures that can predict the cognitive complexity of a wide variety of tasks. One problem is that cognitive complexity is always related to the tasks that must be performed, so measuring the cognitive complexity of a system itself is not possible. Moreover, design tasks are not well-defined by definition. So the creation of cognitive process models is not possible at all. What could be possible is to experimentally correlate system characteristics to the difficulty of a given task. A major problem with this approach is that every person has a different knowledge base and may use different cognitive processes, such as different segmentation and chunking strategies.

Even if it is not possible to create exact cognitive process models of

development tasks, it is possible to create systems that do not have characteristics of which it is known that they account for a high level of complexity in a wide range of tasks. This is the approach that is followed in this thesis.

Chapter 4

Real-Time Systems Concepts

This section provides an overview over the technical concepts of embedded real-time systems, which are relevant for the discussion in the next chapters.

4.1 System Model

This section introduces the basic terminology around real-time computer system architectures, such as systems and components, state, a definition of behavior, and composability.

4.1.1 Architectures

A *computer system architecture* describes the overall design of computer systems that share a set of common characteristics [KB03]. It provides the basic concepts for the development of a class of computer systems. Each computer system is developed according to its own rules and conventions concerning data representation, protocol choices, error handling, etc. These conventions are called the *architectural style* of the system [GIJ+03].

Examples for computer system architectures according to this definition are, e.g., the DECOS architecture [KOPS04] for integrated systems in high-criticality domains, and AUTOSAR [HSF+04], which is an architecture for automotive electronics. Also the well-known personal computers we all use in our homes and offices, together with the operating system and device drivers represent computer system architectures according to the definition used in this work.

This definition of architecture is not constrained to pure hardware, which is commonly denoted as *computer architecture*, but also includes software aspects. In this work, the term *architecture* is used as an abbreviation for computer system architecture.

4.1.2 Systems and Components

In this thesis, a *system* is defined as a collection of components or subsystems. Depending on the viewpoint, a system may be a subsystem of an even larger system, or – if considered in isolation – a subsystem may be seen as a system itself.

With this definition of system, a *subsystem* is just a part of a larger system. The terms system and subsystem are more general than the notion of components, which usually refer to technical subsystems. The notion of a *component* refers to a technical subsystem that is used as a building block for a larger system. This is a very general definition that is not restricted to a specific level of abstraction.

For the discussion in this work, the larger, whole is always denoted as system, whereas its constituent parts are called components. As this work is about computer systems, the term system usually refers to a *computer system*. If not indicated otherwise, the system view always comprises the whole computer system. If a different viewpoint is taken, such as a component being discussed as an independent system, this is mentioned explicitly.

Components are used as units of design. They can offer some degree of self-containedness, e.g., in function or error-containment. For distributed real-time systems adhering to the federated approach, each node computer is considered a component [KS03b]. In an integrated architecture a node computer may host more than one component. This definition of component reflects its most general meaning in technical systems without being restricted to pure software components on the one hand, and software-hardware components on the other hand. Besides the constructive system design approach that uses components to build up a larger system, components can also be seen as the result of a top-down design process, in which a large system is decomposed into a number of smaller components, e.g., for complexity management.

As already mentioned, systems and components can be considered at varying levels of abstraction. *System-level components* represent the components at the level of abstraction that is used by the system integrator. A system-level component can be built-up from a number of smaller components, but this is considered an implementation detail of the component at the system level.

4.1.3 Application Structure

A software application that is executed by the distributed real-time system can be structured into a number of *jobs*. Depending on the architecture and the programming environment, a job can be simply a function in a programming language, such as C [KR78], or an aggregate construct including middleware. If the jobs of an application are distributed, this is called a *distributed application system (DAS)*. Examples of DASs in present day au-

tomotive applications are body electronics, the power-train system, and the infotainment system.

In this thesis the term component is always used in a generic way. It can refer to both an application job as well as to a whole DAS.

The communication mechanisms that are used for coordination and information exchange among collaborating jobs are usually provided by the computer system architecture. However, they can also be provided by an additional middleware layer.

4.1.4 Component Behavior

In accordance to the DSoS conceptual model [GIJ+03] *component behavior* is defined as *traces of activity at* (component) *interfaces* which are *sequences of (perhaps timestamped) send and receive operations* of the component. This definition of behavior represents a viewpoint of the component as seen from the outside, e.g., by a system integrator.

During component design the internals of the component – such as the component state – must also be considered, of course. Thus, from this viewpoint, behavior involves all traces of activity of the component, not just those at the component interfaces. So a definition of behavior always requires a given viewpoint and thus a suitable level of abstraction. When discussing behavior in this work, the component-external viewpoint is used unless mentioned otherwise.

4.1.5 System State

The state of a system can be defined either in a forward-looking style, or in a backward-looking style [GIJ+03]: According to the forward-looking definition, the state of a system at a given instant is a notional attribute of the system that is sufficient to determine its potential behavior. The backward-looking definition of the state of a system at a given instant is the total information explicitly stored by the system up to the given instant. The second definition is therefore often called *stored state*[1]. The stored state alone does not determine the potential behavior of a system, it must be considered together with the system definition. For example, some systems do not have a stored state, and yet have a well-defined behavior. The *abstract state of a system* at a given instant is a notional attribute of the system that is sufficient to determine its potential behavior. The *declared state of a system* at a given instant is the value assigned to a declared data structure that can be accessed via an interface. The declared state is a representation of the stored state that is made available at a system interface.

A *ground state* of a component of a distributed system at a given level of abstraction is defined as a state where no job is active and where all

[1] or "internal state"

communication channels are flushed, i.e., where there are no messages in transit [AKC90]. Thus, if a component is in a ground state then the state of a component is contained in its data structures. The concept of a ground state is important for the re-integration of components after failures, as the ground state can be used as a reintegration point [Kop97].

4.1.6 Composability

Composability is defined as the ease of forming a whole by combining parts [KS03b]. In the terminology of the Time-Triggered Architecture [KB03], the parts to be combined always denote whole node computers comprising hardware and software. In this dissertation composability is defined slightly different due to the more general definition of component introduced in section 4.1.2: Not whole computational nodes are used, but just the system-level components used by the system integrator.

It is not possible to consider pure software systems with regard to composability, as software alone does not have temporal properties. The temporal properties always require some knowledge about implementation constraints so that they can be specified, e.g., by a platform-specific model (PSM) [OMG01]. For the definition of composability, a component must be fully specified in the temporal domain. Even with the slightly different definition of components in this work, technical composability can be defined as [Kop00]:

(1) Independent development of components: This principle is concerned with the design at system level. Components can only be developed in isolation if the architecture supports the precise specification of all component services at the system level.

(2) Stability of prior services: This principle is concerned with the design at the component level. The validated service of a component must not be affected by the integration of the component into the system.

(3) Constructive integration of components: This principle is concerned with system-level properties. It must be ensured that the n already integrated components are not disturbed by the integration of the $n+1^{th}$ component.

(4) Replica determinism: If fault-tolerance is achieved by replication, a set of replicated components must provide replica determinism (see section 4.8.5).

4.2 Models of Time

A central characteristic of a real-time system is that either parts of it or the system as a whole is *time critical*. Systems usually become time-critical if they directly interact with the physical world: A computer system that controls a train must work in a timely fashion so that no dangerous situations

can occur, e.g., that a train enters an occupied section of a track which might lead to collisions.

A real-time system is thus an entity that is capable of interacting with its environment and may be sensitive to the progression of time [GIJ+03]. This means that the system may react differently to the same input at different points in time.

Time is also important for a system's failure to be observable by some other system. For example, if we do not have a well-specified reaction time of a system for a given request, we might be waiting for the answer infinitely and don't know if the system is still working correctly. If the system has a well-specified reaction time and does not answer in time, we know that the system has failed.

The characterization of a system as being *non time critical* does not mean that issues like performance are irrelevant, but such systems can be specified without explicit references to the progression of real time. This means that such systems can be specified by simpler approaches that are less constraining in terms of time awareness so that more design freedom is possible [GIJ+03]. However, not modeling time means that no statements or assumptions about timing can be made. Time is thus an essential notion for real-time systems [Kop97, SS99, dAH01].

4.2.1 Dense Time vs. Sparse Time

Real time can be modeled by a *directed time-line* consisting of an infinite set of *instants* [Kop97]. An instant is a cut in the time-line. A real-time system can use either a *dense time-base* or a *sparse time-base* [Kop97]: If significant events are allowed to occur at any instant of the time-line, this is called a *dense* time-base. If the occurrence of significant events is restricted to active intervals of duration ε, with an interval of silence of duration Δ between any two active intervals, then the time-base is called *sparse*[2] – see figure 4.1. All events that occur within a duration of activity are considered to happen *at the same time*.

Only events that are in the sphere of control of the computer system can be restricted to a sparse time-base, e.g., the sending and receiving of messages. The events that occur outside the sphere of control of the computer system, such as depressing the brake pedal in an automobile, cannot be restricted. These external events are based on a dense time-base. However, the observation of external events can be based on a sparse time-base.

The implications of dense and sparse time-bases in real-time systems are discussed in detail in section 6.4.

[2] or ε/Δ-sparse

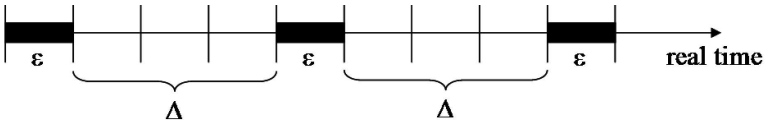

Figure 4.1: Sparse time-base

4.2.2 Global Time

Some distributed real-time systems synchronize the local clocks of the node computers in order to establish an approximation of a common *global time* [KO87]. In many distributed systems there exists no global time. In these computer systems every node has its own local clock that establishes a *local time-base* for the node.

A *sparse global time* [Kop92, Kop97] is a global time available to all components of the computer system that adheres to the principles of a sparse time-base. In a distributed system, events that happen at the same global clock tick in different components are considered simultaneous. Events that happen during different durations of activity and that are separated by the required interval of silence can be temporally ordered according to their timestamps.

If a system has a sparse global time-base, the ticks of the global clock can be seen as a globally synchronized action lattice. Events within the system can be restricted to the globally available lattice points. So these events are synchronized perfectly.

In many distributed systems there exists no notion of global time. In such a system every computational node has its own local oscillator that establishes a local time-base for this particular node. Sometimes just very primitive concepts of time exist, such as the definition of timeouts.

4.3 Real-Time Entities and Real-Time Images

When a real-time system is designed, the states of various objects in the environment of the computer system or in the computer system itself must be modeled. For this purpose, the notions of real-time entities and real-time images can be used.

4.3.1 Real-Time Entities

A *real-time entity* is a state variable that is relevant for a given purpose [Kop97]. It can be located either in the environment of in the computer system or within the computer system itself. Examples for real-time entities are the current speed of a vehicle, or the intended position of a flap of an airplane. A real-time entity has a number of static attributes, such as a name, the data type, the value domain and the maximum rate of change.

The value of a real-time entity at a given point in time is its most important dynamic attribute.

An *observation* is the information about the state of a real-time entity at a particular point in time. An observation is an atomic data structure consisting of the name of the entity, the point in real time when the observation was made, and the observed value [Kop97].

An observation is called a *state observation* if the value of the observation contains the state of the real-time entity. The time of the state observation refers to the point in time when the real-time entity was sampled [Kop97]. An essential characteristic of a state observation is that every reading is self-contained because it carries an absolute value. Many control algorithms use sequences of equidistant state observations which are provided by periodic (time-triggered) readings.

An *event* is a state change that happens at a particular instance. If an observation contains the change in value between the states of the real-time entity before and after the event, this is called an *event observation* [Kop97].

If the time of an observation is not known or of too little accuracy, this is called an *untimed observation* [Kop97]. In a distributed system without a global time, a timestamp can only be interpreted within the scope of the component that has created the timestamp. Often, the arrival time of a message is taken to be the time of the observation. In some systems, timestamps do not even exist, e.g., in purely event-triggered systems where reactions of the system can only be triggered by events. Usually, these events are queued according to the observed event occurrence at the particular node computers, but no time differences between subsequent events are measured.

4.3.2 Real-Time Images

A *real-time image* is a current picture of a real-time entity [Kop97]. It is valid while it is an accurate representation of the corresponding real-time entity. It is invalidated by the progression of time. In contrast, an observation remains valid forever as it is just a statement about a real-time entity at a particular point in time. Real-time images can be created from state observations, from event observations, or with a technique called *state estimation*, which involves building a model of a real-time entity to compute the probable state of a real-time entity [Kop97].

4.4 Interfaces

An *interface* is a boundary between subsystems. The purpose of an interface is information exchange between subsystems. A component interacts with its environment via interfaces. The environment may be either other components, or sensors and actuators that are connected to the component.

An *interaction* is a sequence of message exchanges between connected

interfaces [GIJ+03]. A *protocol* is a set of rules that specifies the interactions between two or more components with connected interfaces [GIJ+03].

4.4.1 Interface Properties

An interface can be characterized by a set of attributes that determine the types of interaction that are possible across the interface, e.g., the encoding, the byte-order, the structure, or the meaning of the information. In addition, the temporal characteristics are of central concern for real-time systems. These attributes of an interface are called *interface properties* [GIJ+03]. A *property mismatch* is a disagreement among connected interfaces in one or more of their properties.

A *boundary line* is a connection between at least two interfaces with matching properties [GIJ+03]. Matching interfaces can be connected directly via a boundary line. If there are property mismatches, a connection system must be introduced: A *connection system* hast at least two interfaces. Its purpose is to connect component interfaces with property mismatches, to coordinate multicast communication, or to introduce emerging services [GIJ+03].

An interaction via an interface usually involves *data flow* and/or *control flow*. Data flow denotes the data structures that are exchanged during the interaction, whereas control flow denotes the immediate influence on the interaction itself that may be exerted by the interacting components. Data flow across an interface can be unidirectional or bidirectional. Control flow in an interaction can also be either unidirectional or bidirectional: Unidirectional control flow occurs if the sender creates a control message/signal to the receiver, but the receiver has no possibility to influence the sender in this interaction, e.g., to block the sender. Bidirectional control flow takes place in interactions where the receiver can also exert control on the sender, e.g., to block the interaction or to slow down the communication.

Flow control is concerned with the control of the speed of information flow between a sender and a recipient to ensure that the recipient can keep up with the sender [GIJ+03]. *Explicit flow control* means that the recipient sends explicit acknowledgments to the sender after a successful arrival of a message, which informs the sender that the receiver is ready to accept the next message. With this control mechanism, the sender is under control of the receiver as the recipient can exert back pressure on the sender. *Implicit flow control* means that the sender and the recipient agree a priori, i.e., before the communication is started, on the transmission rate and when messages are going to be exchanged. No acknowledgments are needed. This control mechanism requires a common time-base available at the sender and receiver. Implicit flow control is well-suited for multicast communication, e.g., publish/subscribe and time-triggered protocols.

Interfaces can be differentiated by the types of interactions that are allowed [Kop99]:

Elementary Interface: This is an interface across which only elementary interactions can occur. In an elementary interaction all messages are transmitted according to the information push model. This means that the consumer of a message does not exert control (no back pressure) on its transmission. As an example, an information producer writes its information periodically into shared memory which can be read by all interested consumers.

Composite Interface: This is an interface across which composite interactions can occur. In a composite interaction at least one message is transmitted according to the information pull model. This means that the consumer of a message exerts control (back pressure) on its transmission. As an example, in a request-response transaction the information consumer must read a control message to the producer to get the required information.

4.4.2 Interface Types

In the context of embedded real-time systems it is common to distinguish between four different types of component communication interfaces [KS03b]:

Service Providing Linking Interface (SPLIF): This interface provides the component services to its users. It is the primary interface of a component.

Service Requesting Linking Interface (SRLIF): A component may request services from other components via this interface. A user of the service providing linking interface may not be aware that a component requests services from other components.

Configuration Planning Interface (CP): This interface is used to configure a component.

Diagnostic and Management Interface (DM): This interface provides selective access to the internals of a component for monitoring and diagnosis purposes.

The service providing and the service requesting interfaces represent real-time service interfaces of a component. So they are time critical. Moreover, they represent *linking interfaces (LIFs)*, which are interfaces through which components are connected to other components within a system [GIJ+03]. Figure 4.2 depicts a system of three components A, B, and C that are connected via LIFs.

A *local interface* is a component interface that does not represent a linking interface. The configuration planning and the diagnostic and management interface are examples for local interfaces. They are usually not time critical.

The *controlled object interface (COI)* is another example for a local interface. It connects sensors and actuators to a component. Hence, it is usually time critical.

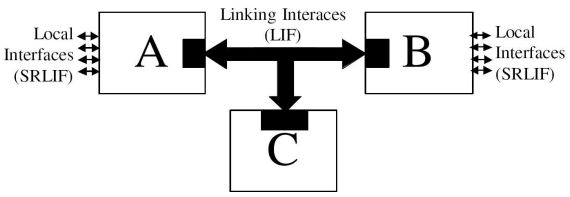

Figure 4.2: A system of three components connected via LIFs (from [KS03b])

4.4.3 Open vs. Closed Components

Based on the relation between a component and its environment, two basic component types can be distinguished [KS03b]:

Closed Component: A component that interacts with its environment only via a SPLIF. The output messages that are produced by the SPLIF are only a function of the SPLIF input messages and the SPLIF state.

Open Component: A component that has one or more SRLIFs that accept inputs from the natural environment[3]. As the natural environment possesses a boundless number of properties it is difficult to provide a complete and rigorous specification.

An example for a closed component is a component implementing a *stack* providing the well-known functions *push* and *pop*. A component that accepts interrupts from the natural environment is an example for an open component.

A special case of a closed component is a *semi-closed component*, which has, in addition to the LIF input messages a (hidden) *clock message*, which represents the beginning of a sparse global time-granule (see section 4.2.2). So a semi-closed component is time-aware, whereas a closed component is not time-aware.

A special case of an open component is a *semi-open component*, which can exchange data with the natural environment without delegating control to the environment, e.g., a sampling system.

When closed components are connected to all SRLIFs of an open component, this can transitively close the component [KS03b].

4.5 Specifications

In nearly all engineering disciplines there is a need to understand the relevant properties of a component for a given purpose. Such descriptions are usually

[3]This does not mean that the component receives the input directly from the natural environment; the inputs can also be passed on by other components that are connected to the environment.

called *specifications*. A specification describes *what* a system should do, rather than *how* it is achieved [GIJ+03].

4.5.1 Specification of Semantics

Specifications can describe aspects of a system at different levels of abstraction. Usually, specifications of technical systems just describe purely technical aspects, e.g., the data types and the rate of change of a value that is provided at an interface. However, the semantics are usually not fully described by specifications, so a specification is – in general – incomplete with regard to application semantics. This means that lots of information is either implicit or it is assumed that the reader of the specification has sufficient knowledge to fill the gaps. The more semantic descriptions there are in a specification, the less information is left for interpretation by the reader.

One reason for the semantic incompleteness of specifications is due to the inherent characteristics of concepts, which usually cannot be fully specified (see section 2.3.2). For example, "driving at a safe speed" needs a defined context to be interpreted meaningfully. Even when trying to avoid such concepts, a designer is likely to rely on other, contextual concepts when writing a specification.

4.5.2 Varying Degrees of Formalism

There exist various specification techniques, with different degrees of formalism. Formal here means abstract in the sense that mathematical symbolisms are used. These formalisms, and the way they are handled, may be conceptually quite different from a less formal specification. Of course, specifications of semantics can not be as formal as operational specifications of a real-time service. In general, the level of formalism must be suitable for the purpose of the specification. If a specification is used for for formal verification, it must be very abstract and formal. However, for various development tasks, less formal specifications are more useful, as the information that is required for the task at hand should be easily accessible and not require the translation across different levels of abstraction.

4.5.3 Formal Specifications

Formal specifications are very useful for formal verification. A thorough concept formation is an essential prerequisite for any formal analysis or formal verification [Kop08]. Only if the underlying conceptual model is precisely specified and understood it can be formalized correctly. However, the creation of formal specifications from a set of informal requirements is very difficult and requires a high degree of expertise. Even worse, the formalization must usually also be sufficiently documented to be understandable

later on [Bar87]. If formalizations are done manually, they may of course also contain errors. So in the ideal case the formal models can be derived automatically from implementation models.

Unfortunately, due to the excessive need of computing resources of today's formal verification techniques the model sizes that can be verified in reasonable time are very limited. This requires the use of techniques to either split the model into parts that can be verified separately, or to generate abstractions that considerably simplify the model. For this to be possible, however, it must be ensured that the model that is verified really corresponds to the actual application. So again a deep understanding of the formal model and the corresponding implementation model are necessary.

Due to the fact that specifications can hardly be complete on the semantic level, we cannot use formal techniques to verify aspects that rely on the missing semantic aspects. For example, the compatibility of interfaces regarding their semantics can hardly be assured with strictly formal techniques. On the semantic level we have to rely on the human mind to be able to fully understand the interface services.

Highly formal specifications are of little use in a design process, or as a basis for system understanding as they are on a different abstraction level and usually require considerable transformations from the conceptual world of the application into the formalism. Moreover, many aspects of a system cannot be formally specified, especially concerning the semantic level. Formal specifications usually can only handle some low-level characteristics. Furthermore, most formal specifications can only be created and read by experts with extensive knowledge in the used formalism; or they require at least substantial training that is not feasible for wide-spread use in general system development.

4.5.4 Interface Specifications

The most frequently used and most important kind of specifications are interface specifications. They specify a component from a particular point of view. If different views (models) of a component are required, different interfaces can be provided – each for a specific viewpoint.

In the context of embedded real-time systems, it must be possible to specify a component interface in the value and time domain, with timing assumptions and timing guarantees [dAH01]. Thus, a LIF is characterized by data properties, i.e., the structure and semantics of the data crossing the interface, and by the temporal properties that must be satisfied. Ideally, a component should be encapsulated so that it maintains its properties (value and temporal) when it is used in a larger context (see section 4.1.6).

For successful communication across an interface, a consistent specification of the interface is required. A LIF specification consists of the *operational specification* and the *meta-level specification* [KS03b]. A *rich interface specification* describes all relevant operational and meta-level aspects of an

interface [BCC+03].

The operational part of the specification describes the syntax, the value, and the temporal properties of the interface. Such a specification must be precise and formal. Moreover, it can be formally verified that interfaces of different subsystems are compatible by ensuring various formally definable constraints [dAH01].

The meta-level part of the specification assigns a meaning to the information that is exchanged via the interface, describing the interface semantics. As already mentioned in section 4.5.1, such semantic specifications are usually incomplete. However, besides this limitation, it is also not possible to provide a rich interface specification without considering the context of use of the component [KS03a]. For example, it is important to consider the use context of a temperature sensor in a car to be able to determine which temperature it delivers – that of the oil or that of the radiator water. This use context is usually not contained in the interface specification of the temperature sensor itself – it is contextual information of the system that uses the sensor.

4.5.5 Uses of Specifications

Specifications are important for the creation of new systems and components as they serve as a reference for development. If each component has a complete interface specification, this makes its behavior well-defined and serves as a stable reference for component implementation.

Another important use of specifications is the integration of already existing components into a system. Only if the behavior of a component is explicitly described, it can be reused without having to consider its implementation.

Furthermore, a specification can serve as a reference for testing and error detection. For example, if a specification says that a message received via an input port must always be in a certain value range, then it is possible to check the actual values against this specified value range. If the actual value is outside the specified range, its sender can be considered erroneous. Similarly, an interface specification describes all possible values and operations supported by a component. So test cases can be derived from the specifications.

4.6 Interaction Styles

Communication is essential for the interplay of the components of a distributed computer system. Even if a system is not distributed but structured into components, the components must interact. Many different techniques of interactions between components and within the components of a system are used in different computer systems. This section summarizes the most

relevant interaction and communication mechanisms for embedded real-time systems. The classification is taken from [GIJ+03].

An interaction may be *connection-oriented* [CDK94]. This means that a state is shared between the communication partners, which is modified by their interactions as a *virtual connection* is set up. For example, for an FTP transaction first a connection must be set up, then the files can be transferred, and finally the connection is closed.

If the individual interactions are independent of each other, this is called a *connectionless* interaction [CDK94]. An example for this are messages based on single datagrams.

4.6.1 Client-Server Interactions

The client-server model is very common in today's real-time systems. It is based on request-reply interactions between a client and a server which are usually one-to-one and synchronous. The interactions can be connection-oriented or connectionless. In the basic model, clients have a fixed a-priori knowledge of the identity of servers. The introduction of a naming service allows more flexibility by determining appropriate servers dynamically.

An example for a standard client-server interaction is the well-known HTTP protocol, where a client requests webpages from a webserver. *Remote procedure calls (RPC)* are another example for a client-server interaction where the arriving call causes the activation of a remote procedure at the receiving component. Remote procedure calls can be implemented to be transparent to the caller. This means that regarding the functional properties, their invocation does not differ from the invocation of a locally implemented procedure. Timing properties and possible failures may be different, though.

The interactions of object-oriented systems can also be considered client-server interactions, with the caller being the client and the server being the callee. An object may thus be both – server and client. In distributed object-oriented systems *remote method invocation (RMI)* is used to call methods of objects residing in different components. The difference to RPC is the late binding of the method call in RMI – the object instance can be created dynamically, immediately before it's methods are called [GIJ+03].

4.6.2 Publish/Subscribe

Publish/subscribe means that systems do not communicate directly with each other, but use a publication mechanism to announce that an event has occurred. The subscribers of the event are notified when the event has occurred. This interaction style provides decoupling between component systems [GIJ+03]: *Space decoupling*, as producers do not need to know who has subscribed to their events, which allows consumers to remain anonymous; and *time decoupling*, as subscribers do not need to be alive at the instant the

events are produced. Typically, the announcer of an event is not informed about who receives the event; it does not know the order in which the event notifications are delivered, and when processing of the events at the subscribers has finished. The publish/subscribe mechanism usually depends on a middleware infrastructure that is responsible for managing subscriptions and for propagating events from the producers to the consumers. It is a purely event-triggered approach.

An example for publish/subscribe interactions are *network variables* in the Neuron C programming language [Ech03]. This language extends ANSI C with network communication functionality. Neuron C is used for smart transceivers in LONWORKS applications, which is intended for industrial, building and home automation. A network variable is written by one component and read by the other components of the LONWORKS application. The readers execute appropriate event handling functions every time the network variable is updated by the writer. Not just pure notifications can be implemented with this approach, it is also possible to transfer data values through the network variables.

4.6.3 Data Passing via a Repository

For this interaction style a shared memory is established that can be accessed by two or more components. The sender writes the data into the shared space and the reader can decide which data to read at what times (information pull). To avoid the mutilation of data due to concurrent access, the repository must ensure atomicity of those actions that might lead to incorrect data, such as concurrent write or read operations on the same data items. Data passing via a repository can be used for both time-triggered and event-triggered interactions.

A well-known example for this interaction style are database systems, which are usually event-triggered applications. An example for time-triggered interactions that are based on data passing is the *temporal firewall* [KN97]. This is an interface model for hard real-time systems that aims at the avoidance of control error propagation and temporal coupling via the interface. It provides a strict data sharing interface: The producer periodically updates the state information in the *input firewall*; the consumer periodically reads the information from the *output firewall*. The timing properties, ensuring, e.g., the temporal accuracy of the data, are established *a priori* – during system design, that is.

4.7 Interaction Contents

In this section the characteristics of the information that is exchanged in an interaction are discussed: Interactions can involve implicit information as well as explicitly passed data items. If data is transferred explicitly, it can be classified according to its semantics. Furthermore, a data item can be

either an atomic unit, or it can represent an aggregate data structure.

4.7.1 Implicit vs. Explicit Interaction Content

When two or more components in a computer system interact, the information can either be transferred explicitly, or implicitly by using a priori knowledge. In some interactions, the receiver is just notified that an event has occurred, no additional data is passed. For example, a timer interrupt signals a component that a timer has expired. The receiver knows what the meaning of the notification is, so no message content needs to be transferred.

The implicit interaction content is common knowledge of the sender and receiver: The receiver must know what the interaction means to be able to react to it properly. With the publish/subscribe interaction technique, it is quite common to implement purely implicit interactions.

Most interactions involve explicit information that is passed from a producer to one or more consumers. The information is transferred as data items (payload), which is part of the interaction. For example, an FTP transaction involves lots of explicit interaction content – the files that are transferred. Another common example where data is transferred explicitly are datagrams that are sent from a sender to one or more receivers.

In many interactions, both explicit and implicit content is transferred, e.g., if an event handler is not just activated, but also additional information about the event is available.

4.7.2 Atomic vs. Composite Interaction Content

As can be seen in the discussion of the previous sections, there exist various means of communication between components. The choice of communication mechanisms is a central aspect of every computer system architecture. One of these mechanisms that is frequently found in embedded real-time systems are *simple messages*. Simple messages are atomic data units, such as the value of a sensor. In this thesis, simple messages are referred to as *messages* for brevity. As the data used in automotive and avionics control systems usually are atomic sensor and effector data values that are transformed by the components of the computer system [Lea94], simple messages can be found in most embedded real-time control systems.

Aggregate data structures, such objects or records are not be treated as atomic units. Depending on the level of abstraction, they either represent an opaque data structure without any known semantics, or they are treated as an aggregate chunk with a visible sub-structure. An aggregate data transfer does not represent a simple message.

4.7.3 State Information vs. Event Information

Corresponding to the concepts of state and event observations (see section 4.3.1), the semantics of data exchanged in an interaction can also be classified into state and event information. Of course, such a classification is just possible with atomic data units. Composite data structures can contain both state and event information.

Messages that contain event information are called *event messages*, state information is transmitted via *state messages*. Event messages are typically transmitted whenever a significant event occurs, whereas state messages can either be transmitted in case an event occurs, but also periodically at a priori known instants. These instants are common knowledge to the sender and the receivers [Kop97].

Messages containing event information are usually sent in an event-triggered fashion. An event-triggered event message combines unidirectional data flow with bi-directional control flow. Except for time-stamped event messages, they are not idempotent, so exactly-once processing semantics are required.

A different approach are periodic state messages, which are characterized by unidirectional data flow and implicit control flow. Since a state message contains state information, a new version of the message updates-in-place the old version of the message. No tight synchronization between sender and recipient is needed, as the recipient can read the state information never, once, or many times, because state information is idempotent. Hence, no queuing or buffering is required.

4.8 Dependability

Dependability is the ability of a system to deliver a service that can justifiably be trusted. It covers characteristics of a computer system that relate the quality of a service that the system delivers to its users during an extended interval of time. There exist different measures of dependability [Lap92] – reliability, safety, maintainability, availability, and security: *Reliability* is the probability that a system will provide the specified service until a given time. *Safety* is reliability regarding critical failure modes. *Maintainability* is a measure of the time required to repair a system. *Availability* is a measure of the fraction of time that the system is ready to provide its specified service, compared to incorrect service. *Security* is a characteristic of a system that prevents unauthorized access to information or services.

4.8.1 Fault - Error - Failure

Computer systems are expected to provide dependable services to their users. The users may be humans or other (technical) systems. Whenever the service of a system deviates from the agreed specification, this is called

a *failure* of the system [Lap92]. Failures are said to be *malign* if the failure costs are orders of magnitude higher than the normal utility of a system. If the costs of a failure are of the same order of magnitude as the loss of normal utility of a system, the failures are said to be *benign* [Kop97]. Examples of malign failures are airplane crashes due to a failure of the flight-control system, or automobile accidents due to a failure of a computer-controlled braking system. If the quality of the braking system just deteriorates but the car can still be stopped safely, this is a benign failure.

Most computer system failures can be traced to an incorrect internal state of the computer. Such an unintended state is called an *error*. The cause of an error is called a *fault* [Lap92]. Thus, a fault often leads to an error, which can become manifest as a failure of the system.

4.8.2 Fault-Tolerance

Fault-tolerance is the provision of the system service in spite of faults. As parts of computer systems can fail, fault-tolerance is essential in safety-critical real-time systems as otherwise a single component failure can lead to a catastrophic system failure.

The developers of safety-critical systems have two options to achieve fault-tolerance [Kop97]:

Systematic fault-tolerance: This means that the fault-tolerance is achieved transparently to the application, either by architectural services or by a dedicated middleware layer. An architecture must provide replica determinism so that fault-tolerance can be implemented by the spatial or temporal replication of computations. The replicated results can then be used to mask faults.

Application-specific fault-tolerance: Fault-tolerance functionality is implemented at the application level, within the application code. The fault-tolerance mechanisms intertwine the normal processing functions.

Making use of systematic fault-tolerance means that the amount of application-specific fault-tolerance functionality can be minimized, so that the application code can remain mostly free from fault-tolerance functions.

4.8.3 Error Containment

A fault-tolerant system must provide *error-containment regions* so that errors that occur in one of these regions can be detected and corrected or masked before the rest of the system becomes corrupted [Kop97]. Error-containment regions can be introduced at different levels. A whole node computer is usually used as an error-containment unit for hardware faults. In integrated systems where application jobs with different criticality levels

can run on a single node computer, dedicated smaller units are required for software faults.

In an integrated architecture, *partitioning* [Rus99] provides appropriate hardware and software mechanisms to ensure strong fault containment. To avoid error propagation by the flow of erroneous data, the error detection mechanisms must be part of different fault-containment regions than the message sender. Otherwise the failure could affect the error detection service. The set of fault-containment regions that perform error containment is called an *error containment region* [Kop03]. An error-containment region must consist of at least two independent fault-containment regions.

4.8.4 Dependability Attributes of Interactions

In a real-time system, component interactions must fulfill various non-functional properties. For example, a collision avoidance system in a car requires not only the immediate recognition of a dangerous situation. The system also depends on the time it takes for the selected maneuver (e.g., an emergency brake request) to propagate to the wheel controllers. *Timing guarantees* can be decomposed into *latency* and *jitter* [GIJ+03]. Latency is a fixed amount of time that is required for an action (delay), whereas jitter is the varying amount of the delay, that may be different each time an action is executed.

Different computer system architectures vary considerably regarding their *delivery guarantees* concerning the non-loss of messages, the order of delivery, and the amount of latency and jitter that can occur.

4.8.5 Replica Determinism

If fault-tolerance is implemented by the replication of components, replica determinism becomes an issue. A set of components is *replica deterministic* if all members show correspondence of their outputs and/or service state changes under the assumption that all components start in the same initial state and execute corresponding service requests within a given time interval [Pol96].

A point in an algorithm that provides a choice between a set of significantly different courses of action is called a *major decision point* [Kop97]. As non-deterministic replicas may follow different courses at a major decision point due to the differences in the data used for the decision, replica determinism is a desired property of fault-tolerant real-time systems architectures. If replica determinism cannot be provided, agreement functions are needed on the application level to resolve all potential conflicts that can be introduced by the non-determinism.

Some important causes for replica non-determinism are [Pol96, Kop97]:

Inconsistent inputs: If different input values are presented to components

then the outputs may differ, too. An example for this is a temperature sensor reading of an analog sensor that results in slightly different values in every component due to the inaccuracies of the sensor and the digitization errors. Not just the value domain is affected, inconsistent inputs are also a problem in the temporal domain. For example, the same temperature sensor may be read at slightly different times by the replicated components.

Dynamic scheduling decisions: If dynamic preemptive scheduling is used, the points in time where an external event is recognized may differ at the different replicas. These differences may lead to different internal states and different behavior of the components.

Non-deterministic program constructs: Replica determinism can be lost if non-deterministic program constructs, such as the select-statement in the Ada programming language are used.

Race conditions: Synchronization constructs, such as wait operations of semaphores can cause non-determinism because of the uncertain outcome which process will win the race. Moreover, communication protocols that resolve access conflicts via random number generation, such as Ethernet, can also cause non-determinism. Similarly, communication protocols that resolve access conflicts by relying on non-deterministic temporal decisions, such as CAN, can give rise to non-determinism of replicated components.

4.9 Time- vs. Event-Triggered Paradigm

There exist two major approaches for embedded real-time systems development: The *time-triggered* paradigm and the *event-triggered* paradigm. Between this theoretical dichotomy, many real-world systems can be seen as hybrid systems that contain time-triggered as well as event-triggered aspects. For example, the FlexRay [Con05] communication protocol supports both time-triggered message transfer as well as a dynamic segment that can be used for event-triggered messages.

4.9.1 Event-Triggered Systems

The event-triggered paradigm is the classic approach used in computer science. In a purely event-triggered system all actions of the system are in fact reactions that are caused by events. These events may be triggered either by the environment or by the computer system itself. This is the way most computer systems work.

The communication in event-triggered systems is typically also driven by events, which means that components only have to communicate if new information is available.

The advantage of the event-triggered approach is that both the computation and communication resources can be utilized very efficiently as they are just used in case events occur. However, this advantage very much depends on the characteristics of the application as event-triggered message transfer often causes overhead due to large message headers, acknowledgment messages and retransmissions in case of failures.

A main drawback of event-triggered systems that is relevant for the considerations in this thesis is that its behavior is not so predictable as that of a time-triggered system. For example, depending on the events that occur in the environment, the computations of the system may be triggered in different order, which leads to different behavior. Furthermore, in high load scenarios the system may behave differently compared to a normal load scenario, due to delayed computations, increased communication system latency and jitter, or lost messages due to buffer overflows.

4.9.2 Time-Triggered Systems

The time-triggered model of computation [Kop98, EBK03] was introduced to develop highly predictable systems with low latency and minimal jitter. Such systems are used, e.g., for safety-critical applications in the automotive and avionics domain. With the time-triggered approach, all actions of a system are triggered by the progression of real time. A sparse global time-base provides a synchronized system-wide action lattice.

Algorithms using real-time images containing state information are typical for time-triggered systems. Usually, these real-time images are updated periodically. If a real-time image can become temporally inaccurate before it is updated again, this is called a *phase-sensitive* real-time image. If the image always is a temporally accurate representation of its corresponding real-time entity, this is a *phase-insensitive* real-time image [Kop97]. Phase-insensitive real-time images can always be used by applications, whereas for phase-sensitive images the temporal validity must be ensured.

In a time-triggered system a time-triggered communication system is used for the transport of messages from a sending component to the receivers. Time-triggered protocols, such as TTP [TTT04], FlexRay [Con05, RWW07] or Time-Triggered Ethernet [KAGS05] can be used. The messages are transferred autonomously by the communication system, based on an a periodic schedule that is generated off-line during the development of the system. Each component has assigned durations for sending messages within the period. Broadcast is typically provided to all interested receivers. The data that is either sent or received is provided in special memory areas that can be accessed both by the application components and by the communication system [Kop97].

Due to the a priori defined schedule, the events of state message transmission depend only on the progression of real time and not on the availability of new information. Even if no new information is available, the messages

are sent anyway, containing the same data as the previous message. So unlike an event-triggered system a time-triggered system always runs with the same load, regardless of the dynamics of the environment. The advantage of this approach is that overload scenarios do not occur, the system is highly deterministic and delivers its service with minimal jitter.

The drawback of the time-triggered approach is its limited flexibility. However, as discussed in the next chapters, this limited flexibility can also be seen as an advantage as it makes the system easier to understand.

4.9.3 Two-Level Design Methodology

A *two-level design* [Nos97, KB03] methodology is typically used for time-triggered systems. It was introduced to achieve composability and distinguishes sharply between system-level design and component design. The system-level design is typically done by the system integrator[4], whereas the component design is done by a subsystem supplier[5]. The system-level components used by the system integrator are application jobs, or even sets of jobs in case a whole DAS is implemented by a specific vendor.

The development of a system starts with the system-level design. The input to this design step is the requirements specification [Nos97]. If a new system is created from scratch, then in the system-level design phase involves decomposing the application into system-level components. Moreover, the linking interfaces between the system-level components must be fully specified in the time and value domain. For example, it can be specified that a message must be sent from one component to another component every 5 milliseconds. Based on these specifications and on a global notion of time, the communication planning can be performed. This means that a global communication schedule can be established from which the exact data fetch and delivery instants at all components can be derived.

Existing components can also be reused in the system-level design as their interface specifications can be integrated. In this case just the calculation of the fetch and delivery instants must be re-run for the new system.

So, more precisely, the system-level design consists of two sub-phases: First, there is the interface specification phase, which is mainly manual design work that has to be done by engineers. Then the verification and configuration phase follows, in which the communication schedule is generated and the compatibility of the interfaces can be verified. This second sub-phase is usually automated except for semantic verification.

The system-level design phase is typically supported by a system-level design tool [KN95, TTT07b, TTT07d].

The second large step is the component design phase, which involves the development of the component internals – mainly the application software.

[4] e.g., an OEM for an automotive system

[5] e.g., a tier-one supplier

This step can also be subdivided into two sub-phases: First, the application software is developed according to the interface specifications that were established in the system-level design phase. So the system-level design constrains the component design. This first sub-phase involves again manual design effort that must be done by engineers. If existing application software is re-used, nothing must be done for this first sub-phase.

Then, in the second sub-phase the necessary middleware layers and operating system configuration can be generated. This is necessary if multiple system-level components are mapped onto the same computational node (hardware unit). This second sub-phase is usually also automated with appropriate development tools [KN95, TTT07a, TTT07c]. For this step the precise data delivery and fetch instants of the global communication schedule are taken into account to generate appropriate node-local configurations, such as node-local job scheduling tables. Finally, all applications together with the node-local configurations must be compiled, linked and downloaded onto the hardware units.

4.10 Chapter Summary

A computer system architecture describes the overall design of computer systems that share a set of common characteristics, such as data representation, protocol choices, or fault-tolerance mechanisms.

A computer system can be decomposed into a number of components. Components represent units of design that can offer some degree of self-containedness, e.g., in function or error containment. System-level components represent the level of abstraction of components that is used by the system integrator.

An integrated system architecture supports multiple distributed application subsystems that can further be decomposed into a number of jobs. These jobs are the basic units of design and represent system-level components.

Component behavior from a component-external view can be defined as traces of activity at the component interfaces, which are the message send and receive operations.

Composability is an important architectural principle that ensures that the effort of component integration is minimal as the properties that were established at the component level do not change when the component is integrated.

An interface specification consists of the operational specification, which covers the syntactic and temporal aspects of the messages exchanged across the interface, and the meta-level specification which deals with the meaning of the information contained in the messages. The operational specification can be used for formal verifications; semantic descriptions can usually not be verified formally.

The stored state of a system is the total information stored by a system up to a given instant. The declared state is a representation of the stored state that is made available at a system interface.

A ground state of a component of a distributed system is defined as a state where no job is active and where there are no messages in transit, so the state of the component is fully contained in its data structures.

There exist different interaction styles of components, with client-server interactions, publish/subscribe and data passing via a repository being the most important ones. Data passing via a repository is the only interaction style that avoids control flow via the interface, which enables temporal decoupling between components.

The availability of a notion of time is essential for real-time systems. With a dense time-base significant events can occur at any instant of the time-line. With a sparse time-base the occurrence of significant events is restricted to certain intervals of activity that are separated by intervals of silence. All events that occur within an interval of activity are considered to happen at the same time. In a distributed system with a sparse global time-base, events that happen at the same global clock tick in different components are considered simultaneous. So a sparse global time-base provides a system-wide action lattice for perfectly synchronous operation. Events that happen during different intervals of activity can be temporally ordered according to their timestamps.

Fault-tolerance is an important property for dependable systems as component failures must not lead to a system failure. Systematic fault-tolerance is transparent to the application. Application-specific fault-tolerance is implemented at the application level, within the application functionality.

Replica determinism ensures that replicated components show the same behavior if they start in the same initial state and execute corresponding service requests. Replica determinism is essential to achieve fault-tolerance by the replication of components. If replica determinism cannot be provided, agreement functions are required to resolve all conflicts that are introduced by non-determinism.

In an event-triggered system all actions of the system are reactions caused by events. This control strategy allows to utilize resources very efficiently and provides a high degree of flexibility.

In a time-triggered system all actions are triggered by the progression of real time. Time-triggered systems are highly predictable with low latency and minimal jitter.

A two-level design methodology is typically used for time-triggered systems. It supports the development of composable components and strictly separates between system-level design and component design. The system-level design is created by the system integrator and includes the precise specification of all component interfaces. Then, the components can be created independently of each other based on the respective interface specifications.

Chapter 5

Complexity Management

In this chapter complexity management in the context of embedded real-time systems is discussed for different tasks and viewpoints, such as the decomposition of large systems into components, component development and component integration. In addition, some general techniques that can be used throughout system development are presented.

5.1 General Considerations

This section gives an introduction to complexity management, defines accidental and essential system characteristics, and provides a rationale for design for simplicity.

5.1.1 What is Complexity Management?

As discussed in chapter 3, human cognitive resources are limited. Overloads must be avoided. Ensuring that we only have to consider a small number of chunks simultaneously is of utmost importance for tasks such as system development and maintenance. Tool support can help us with those tasks where our working memory is likely to be overloaded as too many items must be dealt with simultaneously. But of course, not all tasks can be automated and performed by tools. Just well-defined tasks for which algorithms are known can be done by tools. Considerable parts of system development will always be design tasks where human understanding is required.

Large computer systems, e.g., integrated architectures for automotive and aerospace control applications [KOPS04] cannot be considered in all details as a single unit. For almost any task it is necessary to reduce the relevant aspects to a manageable level. Each development step must have a level of complexity that can be handled. If tasks with high cognitive complexity must be performed, errors are likely to occur.

Model building is an inherent characteristic of the human mind [Hay91, JL04]. We usually do not see the world as it really is, but create abstractions

(models) from it. For example, when we use a cooking spoon we do not think about it in terms of particle physics.

Model building is also an essential activity in the development of a computer system: We have to build models of the system that focus on the relevant properties for a given task and ignore irrelevant details. A model can be defined as a *deliberate simplification of reality with the objective of explaining a chosen property of reality that is relevant for a particular purpose* [Kop08]. For example, when we are interested in the physical structure of a distributed system, we can create a model that corresponds to the physical building blocks, such as node computers. If we are interested in the logical structure, we can create a model that corresponds to the functional units, such as application jobs.

Complexity management techniques aim at keeping the cognitive complexity of various development and maintenance tasks low, e.g., by providing appropriate models, or by structuring the system in certain ways. A common goal of many techniques is to reduce the number of aspects that must be considered for a given task instead of having to handle large parts of the system at once.

Low coupling between modules and high cohesion inside each module are commonly accepted characteristics of good software design [YC79, Rie96]. In this chapter it is discussed how this can be achieved in the context of component-oriented embedded real-time systems – mainly by drawing on concepts and theories introduced in chapters 2 and 3.

Most of the techniques presented in this chapter do not focus on a specific task. They rather try to provide good system characteristics to support various development tasks, such as component implementation, verification, or integration. The techniques are often used in combination, and the underlying mechanisms are sometimes closely related to each other. So the borders between the concepts presented in this section are somewhat fuzzy. As a consequence, the the descriptions are slightly overlapping to avoid too many cross-references where just a single thought of another section is relevant.

5.1.2 Accidental vs. Essential Characteristics

Computer systems are often unnecessarily hard to understand because of functionality and characteristics that are introduced during their development. These aspects are not inherent to the problem that has to be solved. Brooks [Bro87] has made an important distinction between two kinds of "complexity": *Accidental complexity* is caused by bad design, e.g., by paradigm or interface mismatches, whereas *essential complexity* is inherent to the problem at hand. In this respect, the design of a comprehensible system must avoid accidental complexity. To avoid confusion with the terminology of cognitive complexity introduced in section 1.2, the terms *accidental characteristics* and *essential characteristics* of a system are used instead of accidental and essential complexity.

An example for an accidental characteristic is the introduction of a connection system (a wrapper) for the only purpose of being able to connect two interfaces that do not match.

Sometimes it is hard to judge whether a specific characteristic of a system or component is accidental or essential. A given computer system architecture or design philosophy may have special aspects that are essential for the architecture, while according to a different design philosophy the same aspect may be seen as accidental. So it seems that accidental characteristics can only be defined with respect to a given computer system architecture.

Accidental characteristics often can only be discovered by a sudden insight or by hints from other, usually more experienced people. As long as a developer does not see a different, more simple solution, in the developer's view the system has just essential characteristics.

As explained in section 2.4, experts use more elaborate reasoning strategies than do less experienced people. So computer system architectures should be created by experts. The actual implementation and development of systems according to a given architecture can then be done by less experienced people. The constraints imposed by the architecture must provide sufficient guidance so that systems with good characteristics can be developed. Thus, complexity management must be considered at the architectural level so that comprehensible systems will be created. A computer system architecture should not introduce a large number of accidental characteristics that increase the cognitive effort of the development tasks.

5.1.3 Design for Simplicity

Complexity becomes manifest in increased time needed for a task, and in a high number of errors that occur while performing the task.

Increased time needed for a task is reason enough to strive for simple systems and simple tasks so that development and maintenance can be sped up. Regarding the number of errors and overlooking of aspects, mental overloads can result in even more serious problems: For similar tasks, situations of mental overload occur for most people, or at least for people with a similar background (similar knowledge base in long-term memory). So these errors caused by mental overload are especially hard to detect – even with considerable reviewing effort. However, peer reviews are a common technique to achieve high quality products with a low number of errors, for example in certification projects of computer systems for automotive or aerospace applications.

Moreover, if people face difficult tasks, they often misinterpret the complex problem as if they were facing a simpler problem – see section 3.1.4 for a discussion of oversimplification. This is also a serious problem with peer-reviewed artifacts, as errors may remain undetected if the reviewer uses an oversimplified mental model of the artifact under review. So design for simplicity is an important principle for the development of real-time systems

that must be certified according to the highest criticality level.

5.2 Standardization and Architectures

A usual approach proposed for complexity management is standardization. But how can standardization help? In this section first the requirements and then the advantages of standardization for comprehension are discussed.

5.2.1 Requirements

To be able to make use of standardization, a standard must exist, of course. This standard must be well-known to all involved users so that they all have the same conceptual model of the standard and how it must be used. This means that a suitable conceptual description for the development of a mental model of the standard is available.

A standard that does not offer an in-depth conceptual description is hard to understand and needs considerable learning effort. Unfortunately, many industry standards do not offer good conceptual descriptions. They often just provide detailed technical specifications and ignore the requirement for high-level conceptual models.

A usual way to specify a standard is to define a computer system architecture. Architectures enforce standardized interfaces, communication mechanisms, system structure, and methodology. Moreover, appropriate development tools should be available to support the development process.

An important requirement for an architecture is that it restricts the design considerably. If an architecture allows too much design freedom it cannot offer sufficient guidance. For example, if very diverse communication mechanisms are supported, then it is again up to developer to choose the appropriate one.

The architectural concepts should provide an easy-to-understand basis for development and not provide too many exceptions to accommodate special cases. Otherwise most of the advantages mentioned in the next subsection are just minimal or not even present at all.

5.2.2 Advantages

Architecting is a consequence of system complexity [Rec91]. An architecture reduces the effort of the design process as the most fundamental decisions have already been made. It guides the designers and constrains the possible design decisions. So the architectural level is where considerations about complexity must start: A comprehensible computer architecture must be designed with the characteristics of human cognition in mind.

Of course, to be able to choose a suitable architecture, a system designer must first consider possible dichotomous architectural alternatives [Chr07].

In this way, both newcomers and experts can get a clear understanding of the different options available to make better initial architectural decisions.

A major advantage of a standard in the form of a computer system architecture is that a set of well-known core concepts can be established. These concepts can form the basis for system development and can also serve as basic-level categories as described in section 2.3.2. So with an architectural approach one does not have to start from scratch to re-invent the wheel anew.

An architecture that does not explicitly define a network of basic concepts may be quite hard to understand and require considerable learning effort: The essential basic-level categories must first be identified in an often overwhelming network of architectural concepts. Unfortunately, this fact is ignored in many architecture and standard definitions.

Standards and a architectures restrict the design according to given aspects. This means that not all possible directions can be implemented so that the problem domain becomes smaller. The basic design decisions have already been made so that the system development is guided into a specific direction.

An architectural development approach with a well-defined methodology enables the use of tools that can check the well-defined aspects of the architecture. For example, architectural models and constraints can be established and the design can be checked against these definitions. Moreover, an architecture enables the establishment of development guidelines and the provision of design patterns that can guide those aspects of development that cannot be formally constrained.

If an architecture defines standardized interfaces among the components, which is quite usual, then a high degree of regularity or similarity can be achieved among the components. This regularity can in turn support the re-use of existing knowledge as well as comprehension strategies from one component to the next.

One of the most important advantages of a standard is that if it is widely used, there are many experienced experts. These experienced people can develop new systems more easily than less experienced people. So a re-use of knowledge from one system to the next is possible.

5.3 Interface Issues

As defined in section 4.4, an interface is a boundary, i.e., a technical border line, between linked subsystems. The nature and the placement of interfaces in a computer system decide on the system's structure and the degree of coupling between the components. As this coupling is a major factor that affects comprehension, interfaces are discussed extensively in this section.

First, the relation between mental and technical representations is con-

sidered, then conceptual chunking and segmentation support by interfaces is discussed in detail. In addition, component encapsulation is considered from the technical and conceptual viewpoint. Then, the role of hierarchies, layering and stable intermediate forms are considered. Finally, the characteristics of an ideal component interface are described.

5.3.1 Support for Chunking and Segmentation

Interfaces can represent reduced representations (models, abstractions) of subsystems if they restrict the view of the interfacing subsystems in certain ways. Furthermore, interfaces technically structure a system into a number of subsystems. These two technical mechanisms of reduction and partitioning are closely related to the psychological concepts of conceptual chunking and segmentation.

According to relational complexity theory (see section 3.2.6), conceptual chunking and segmentation are used to handle complex tasks. For the task of understanding a computer system, conceptual chunking and segmentation are essential, as a whole system cannot be considered at once. So the introduction of interfaces can split a large system into subsystems which can then be better suited for more specific comprehension or development tasks.

If the technical border lines (interfaces) and conceptual structures of the system match, this can support comprehension. However, as described in the following subsections, the conceptual structures that are required to get an understanding of a system, are often not reflected by corresponding interfaces at the technical level.

5.3.2 Abstraction and Conceptual Chunking

According to relational complexity theory, chunking means to reduce the dimensionality of the problem space by hiding information that is not needed for the task at hand. Interfaces support chunking by hiding component-internal details. So they are the primary means for conceptual chunking at the technical level, often implicitly, by providing an abstract model of a component.

According to cognitive load theory (see section 3.2.5), all information that is not needed for a particular task should be hidden. So ideally, an interface hides as much information as possible, so that just the information that is really necessary for a given task is available. According to relational complexity theory, the dimensionality of the tasks must also be considered. The dimensionality of a task can be reduced if a component interface does not have a large number of highly relational properties, but encapsulates relations as much as possible. For example, a temperature sensor component may have two or more physical sensors. It can encapsulate this fact by providing an average value, or by making use of an even more elaborate algorithm to achieve a sensible temperature value. If, however, values from

multiple physical sensors are passed on as raw values that require further processing before they can be used, this introduces additional conceptual relations into the system.

The elimination of relational component properties is especially important at the system level, where relations typically involve more levels of abstraction than node-local relations. In addition, small-scale relational aspects within a component do not impair the understandability at the system level if they can be fully encapsulated within the component.

So for a given task, an interface can reduce the dimensionality of the task by encapsulating relational operations within a component, thus providing a more abstract view of the component. To achieve this reduction, the interface must be designed with regard to the tasks that must be supported.

If very different tasks must be supported it is likely that the component must provide separate interfaces to support all tasks optimally. So interfaces can be used *aspect-based*. This means we can create different abstractions of the same system, i.e., we can see the system from different perspectives, depending on the task that must be performed. E.g., a maintenance engineer in a garage needs a different view of a brake-by-wire system in a car than the developers of the system.

The discussion so far has just considered the abstraction provided by interfaces as a simplification of the components. However, simplification can also be achieved in the opposite direction: An interface of a component can serve as an abstraction of its environment. An interface insulates an artifact from the environment and so it serves as an invariant relation that is maintained between the component and its environment, independent of variations over the defined range of parameters [Sim81]. Thus, abstraction is possible in both directions. The abstraction of the environment for the component-internal view provides a well-defined model of the environment. For example, if all relevant real-time entities of the environment are specified in the time and value domains, this allows to ignore all those aspects of the environment that are not contained in the interface specification.

5.3.3 Partitioning and Segmentation

Segmentation according to relational complexity theory means to split a complex task into a number of subtasks which can be performed in a series. This reduces the arity of the task to keep the cognitive resource requirements to a manageable level. A related technique is the decomposition of large computer systems into smaller components. This partitioning also serves to keep the complexity of design and maintenance tasks low: If just a part of the system must be considered, e.g., for component development, the rest of the system can be ignored. So segmentation alone can already considerably reduce the complexity of various development tasks.

Regarding general system comprehension, segmentation on the system level means to decompose the large system into smaller conceptual units.

These conceptual units can be the components (i.e., the technical units) that build up the system so that the component interfaces represent segmentation boundaries. So a high degree of conceptual decoupling from the rest of the system can be achieved. However, this kind of system structure cannot be found in all component-oriented systems – see section 5.3.7.

In practice, segmentation is used in combination with conceptual chunking: A component interface mostly not just serves as a pure borderline between two subsystems. As described in section 5.3.2 it can also provide an abstraction of at least one of the interfacing subsystems.

5.3.4 The Interface View

Segmentation and chunking is not just used for comprehension tasks at the system level by splitting up the large system into a number of components. Segmentation and chunking can also be performed within a component interface: There, it supports comprehension by providing appropriately structured information and functionality. If the interface is badly structured or does not exhibit any obvious structure at all, the user first has to find an appropriate chunking and segmentation strategy, i.e., to split the interface into parts and identify the ones that are needed for the task at hand. Such a search for a comprehension strategy of course takes some time and cognitive effort. It involves model building and concept formation about the interface to identify a meaningful structure. From this perspective, an interface should provide a meaningful concept and not just present a mixture of unrelated functionality. If the latter is the case, further partitioning or restructuring at the system level may be necessary.

Chunking of data is frequently found in computer systems to support higher levels of abstraction. For example, aggregate data structures are often used to group related data items that belong together. They can be seen as a single unit at a higher level of abstraction. These abstract data chunks can then be used for communication between components to raise the level of abstraction of the communication.

5.3.5 The Interaction View

Segmentation and chunking is also possible for the interactions between components. Chunking can be found frequently, as it often makes sense to chunk some low-level details of an interaction into a single high-level interaction. For example, transferring data from one component to another may involve a series of low-level interactions, such as establishing a connection, transferring the single data bytes, probably performing retransmissions of lost data, and finally closing the connection. At the application level this can be handled as a single message passing operation.

5.3.6 Component Encapsulation

A component interface can encapsulate a component so that the component internals are neither visible nor accessible from the outside. This principle of *information hiding* [Par72] is very well known and often used – most prominently in object-oriented programming.

The encapsulation an interface provides of a component is usually only considered from a viewpoint outside of the component. However, as already mentioned in section 5.3.2, an interface can also serve as an abstraction of the environment. This purpose of an interface is not known so widely and so it is also used far more scarcely. For an interface to provide an abstraction in both directions, it must specify output ports and input ports. So both directions of interactions are covered by the specification.

If just a service interface as in a client-server interaction is specified, the interactions that are initiated by the component itself (outbound connections) are usually not covered by the interface specification. Just the possible inbound connections are covered, as the specification only describes the potential service requests that can be fulfilled by a component. This difference has important implications on comprehension. If a component interface is fully specified in both directions, the following advantages can be achieved:

Environment Simplification: If the interface encapsulates both inbound and outbound service requests and data access, the interface serves as an abstraction of the environment. A component design has to deal only with those aspects of the environment that pass through the interface.

Stability: If an interface fully describes all relevant aspects of the environment, this means that the interface represents a stable view of the environment that cannot change as long as the interface is not explicitly changed. So, for system development, the interfaces of a component represent *stable intermediate forms* [Sim81] that provide a stable basis for development.

Explicit Connections: If the interface specification covers all kinds of interactions, this makes all connections explicit, instead of having them scattered all around the component implementation. Moreover, all connections between components can be identified easily.

The simplification of the environment is an essential characteristic to reduce the complexity of component development. For example, if an interface just says that the temperature is either too high, too low, or just right, this will require less effort during component development, than if the raw temperature values are passed.

The stability property of a fully specified interface with input and output ports avoids the unintended propagation of changes across the interface.

A change inside a component rather leads to interface mismatches, which can be detected. So a change in a single component cannot result in unintended effects in other components. From the viewpoint of the component developer, also the environment appears more stable by the introduction of a fully specified interface with input and output ports.

As already explained in section 2.2.3, understanding means to see relevant connections. So getting an understanding of a component involves the identification of relevant connections to other components. If the interface specification does not cover all outbound interactions, as for example, in an object-oriented architecture, this means that the outbound interactions are scattered all around the component implementation. So these outbound interactions, such as remote method calls, represent an *implicit interface* within the component implementation. This has severe consequences on the level of abstraction that is required for analyzing a component: As the outbound interactions are scattered all around the component implementation, the component implementation level must be used when identifying the connections [1]. If, however, all interactions – including the outbound ones – are covered by the interface specification, this means that the level of abstraction can be higher: All connections can be investigated without having to consider the component implementation. So in general, a fully specified interface with both input and output ports supports comprehension at the component interface level, whereas implicit interfaces require a consideration of the component implementation level. In the latter case the component interface specification does not offer a sufficient relational description of the component that is required for comprehension.

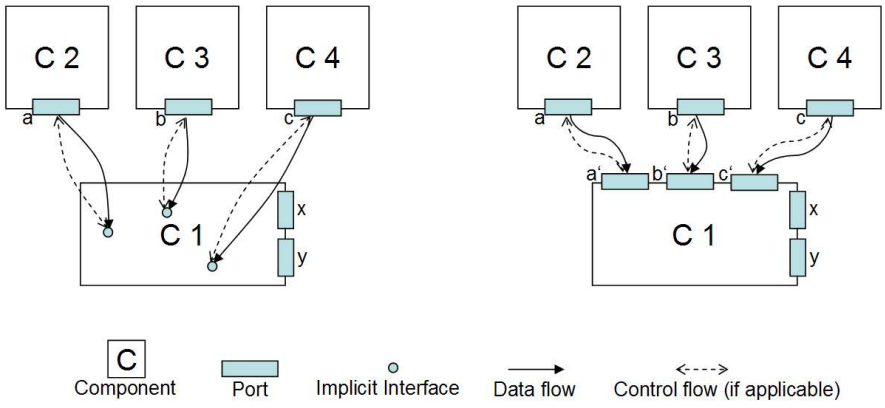

Figure 5.1: Implicit vs. explicit interfaces

Figure 5.1 depicts the difference between implicit interfaces and explicit ports. In the left part of the figure component C1 has two ports x any y that are described by the interface specification. To implement its service, C1 requires services from the components C2, C3 and C4. As these relations are

[1] A kludge to overcome this problem is the use of modeling and code analysis tools that make those implicit connections visible.

not covered by the interface specification, they represent implicit interfaces that are scattered all over the implementation of C1. Depending on the kind of service and the interaction type, either just data flow from the components C2, C3 and C4 is necessary to C1, or also control flow might be involved. In the right part of the figure, C1 again needs the services of C2, C3 and C4 to implement its services available via ports x and y. The interface specification now covers all interactions of C1, which are implemented via explicit ports. No implicit interfaces are used and the connections between the components can be easily identified without having to consider the implementation of C1. So the component interface level is sufficient to analyze the relations between C1 and the other components of the system.

5.3.7 Built-In Segmentation and Chunking

As already mentioned above, interfaces can represent segmentation boundaries for system comprehension. The extent to which this is possible depends on the characteristics of the interfaces. Technically, an interface can always be seen as a border line between a component and its environment. However, if an interface between two components is not well-defined, e.g., by supporting a priori unknown kinds of interactions, or if the border line between the components is not clear, as for example, in an object-oriented system (see section 5.4.1), then the conceptual segmentation lines become less clear. At least, they are not immediately apparent at the surface. A similar problem is a component interaction that involves hidden mechanisms or relies on component implementation details that are not contained in the interface specification. So additional conceptual structures are necessary to be able to understand the hidden or non-apparent mechanisms that are involved. Besides the technical structure built into the system by its designers via explicitly defined interfaces, additional conceptual structures may be required. This is the case if the structures provided by the interfaces do not match all those conceptual structures that are necessary to understand the system.

In the ideal case, all interactions of components occur via well-defined, fully specified interfaces and do not rely on non-apparent mechanisms. So the technical structures provided by the interfaces can be taken as a good basis for conceptual chunking and segmentation. This can be seen as *built-in segmentation and chunking*.

If the components of the system and conceptual structures that are required to understand the system do not match, this causes increased cognitive effort as both structures must be considered simultaneously and the required underlying concepts must be detected. For example, two conceptually unrelated kinds of functionality can be closely intertwined in a component instead of being separated into dedicated components. In this case the chunks that are needed for understanding just one kind of functionality must first be identified. This additional cognitive effort also means that comprehension errors can be introduced.

The advantage of explicit chunking and segmentation via component interfaces is that they represent fixed and explicit structures in the system. If additional mechanisms that are not apparent at the surface are required, they tend to be forgotten easily. They just exist in the conceptual world of the developer. This considerably complicates system understanding as the hidden structures must first be identified. People who maintain large systems, where components are considered just once every few months or even years, know this problem very well.

Built-in segmentation and chunking via interfaces can also reduce misunderstandings that are caused by different interpretations of the system structure by different people. Moreover, explicitly available structures support *external cognition* [War04], which reduces the working memory load as the structures are physically available and must not be kept in working memory while a cognitive task is performed.

See section 5.4.2 for a detailed discussion of how built-in segmentation can support the conceptual decoupling of components.

5.3.8 The Optimal Component Interface

Concluding from the previous discussion, a component interface should have the following properties:

Coherence: An interface should provide a meaningful abstraction, i.e., a semantically coherent mental model, so that it can be handled as a conceptual unit.

Decoupling: An interface should provide a high degree of conceptual decoupling from the rest of the system so that independent development and verification is possible, without knowledge about any other components. So an interface should provide a sufficient conceptual model (i.e., a mental model) of the component environment. If no sufficient conceptual model of the environment is available, independent development is not possible as knowledge about other components of the system is required.

Task-Level Inclusiveness: An interface should provide exactly those services that are required for the task at hand. This requires to avoid services not needed for the task, which just add accidental characteristics to the interface. Moreover, the services must be well-suited for the task at hand. They must neither be too fine-grained, nor too general as missing the optimal level of abstraction leads to accidental characteristics elsewhere in the system, for the users of the interface.

Component Simplification: An interface must provide a considerable simplification of the underlying component. Otherwise the introduction of the interface is a needless indirection that just makes the task of system understanding more complex. If the interface just reflects

the implementation this is an indication of bad design. If no properties of a component can be abstracted away the system is likely to be badly structured. In addition, a component interface should encapsulate relational aspects if the underlying relation is not needed outside the component.

Environment Simplification: An interface should provide a simplified and stable view of the environment to ease component implementation.

Completeness: An interface should be complete with regard to all relevant relational connections to other components and real-time entities outside the component. This means the interface must provide an explicit description of all interactions of the component. The relations must be fully described in the time and value domains and also provide a sufficient semantic description. So implicit interfaces can be avoided and the interface provides a sufficient description of all possible interactions.

Encapsulation Strength: An interface should not expose any of the component internals that are not covered by the interface specification. An example for a violation of this principle regarding the operational specification would be an error code that is not defined in the specification. An example for a violation of this principle regarding the meta-level specification would be to rely on concepts that are concerned with the implementation details of the component. Such violations of the meta-level specification are often especially hard to identify.

Stability: An interface should support temporal composability so that behavioral stability is guaranteed. If a component exhibits different behavior before and after the integration, this invalidates the conceptual models that were established at the component level. So additional models must be developed.

5.4 Seeing Connections

Acquiring a general understanding of a system and its constituent components (the *big picture*) is a frequent task in computer systems development and maintenance. As described in section 2.2.3, understanding means to see the relevant connections between related parts. In an embedded computer system there exist lots of connections between its constituent parts that are essential for correct understanding, e.g., data dependencies or shared resources.

A computer system where the relevant dependencies are apparent at the surface is easier and faster to understand than a computer system where the dependencies are hard to identify. Minimizing the number of relevant connections simplifies understanding. In this section it is discussed, how

system structure can be made explicit and how the number or relevant connections can be reduced.

5.4.1 Making Connections Explicit

There are endless possibilities of introducing relations between components that are not immediately apparent. Often, these dependencies are not introduced intentionally, but are rather accidental characteristics of the design. Such dependencies are sometimes not even recognized by the developers of the system. For example, it is often not clear whether a change in a piece of software affects other parts the system. For such systems the well-known saying *"never touch a running system"* applies. The connections between the parts are often not known until a problem occurs.

There are two requirements to make the connections between the parts of a system explicit:

1. Clear system structure
2. Explicit communication and coordination mechanisms - no hidden interactions

First of all it is important that a system has a clear structure. This means that for every part of a system it is clear where the part belongs to. While this condition is easy to fulfill for hardware components, for software it is often less clear where the parts that build up a system belong to. For example, when deep inheritance hierarchies are used to program a component, it is often not immediately obvious where the parts that really go into the system come from. They may be scattered around various files, directories, or even development tools. Consequently, when looking at a particular piece of code or configuration data, it might not be clear if this piece is actually used, and if it is used, what parts of the system make use of it.

As already mentioned in section 5.3.7, in the optimal case the conceptual structures that are used for the segmentation and chunking process for system understanding are reflected directly by the system structure[2]. This means that the relevant structures that are needed for comprehension can be easily identified by the person trying to understand a system. So a developer or maintenance engineer does not have to develop his or her own chunking and segmentation techniques, but can simply take the components as chunks.

The second requirement – explicit interaction mechanisms – means that all communication and interaction between the components of a system must be implemented through dedicated mechanisms. This prohibits interactions such as private data access, side-effects in service calls, making use of undocumented implementation details and similar mechanisms. If resource

[2]built-in segmentation and chunking

sharing causes relevant connections, these connections must also be made explicit or prohibited. A computer systems architecture shall offer a set of interaction and communication mechanisms so that there is no need for applications to by-pass those explicit interaction mechanisms. Furthermore, it must rule out relevant interactions that are caused by resource sharing.

If an interface specification covers all interactions of a component by a complete description of dedicated input and output ports, this makes all interactions explicit at the component interface level (see section 5.3.6). So the cognitive effort can be reduced considerably compared to a system where the component interface specification does not cover all possible interactions.

5.4.2 Minimizing Connections

To support comprehension, making the connections between related parts explicit is just one important goal. Another aspect is to reduce the connections between the constituent parts. According to relational complexity theory, a connection (relation) that has a high arity is always hard to understand. So even if we know that there exists a relation between, for example, five independent variables, their mutual influence may be hard to comprehend.

As already discussed in section 5.1.2, it is hard to ensure that a system only has essential characteristics. However, as a first development step, a designer should create a high-level model and then identify required high-level interrelationships. Probably multiple models must be tried to be able to minimize the number of dependencies. As this is pure design work, one can of course never be sure if there exists a better solution. At the level of a detailed implementation model it is often impossible to decide whether a dependency is accidental or essential.

For comprehension, just the different conceptual aspects of communication are relevant, not the rate of message exchange. For example, if one component of an application regularly sends a single state message to another component this is a very simple interaction, even if the rate of message exchange is high. However, if there is a high number of conceptually different messages, the interaction and thus the relations between the components require far more comprehension effort.

An important design principle to minimize connections is to keep a priori unrelated parts as separate from each other as possible. This can be done, for example, by hiding the existence of one application to a different application that runs on the same system. If this separation is supported by by a computer system architecture in the time and value domain, then no complexity is introduced by the integration of different applications into a single system.

A major factor for interdependencies between components is control flow. Control flow usually serves to propagate events and data updates within the system. If control flow between components can be removed, this reduces

the degree of coupling. An important design principle in this respect is *temporal decoupling* of components. This means that the temporal control of a component is fully performed by the component itself, no external control is possible. This includes, for example, explicit control signals, such as interrupts, and waiting for the occurrence of external events, such as the arrival of a message. Conditional evaluation of input data is another example where external control can be exerted on a component.

Conceptual decoupling of components is a design principle that ensures that reasoning about components is possible with no or just minimal knowledge about other components [RS07]. Conceptual decoupling does not mean conceptual independence, though. A component necessarily has conceptual relations to collaborating components. Decoupling just means that the details of this relation are not necessary for the component-internal viewpoint. So the knowledge about the component interface is sufficient to be able to develop the internal details of the component. Of course, this requires fully specified interfaces in the temporal and value domain, as well as a sufficient semantic specification. So besides the interface specification of the component under consideration, no further knowledge about any other components of the system is necessary, i.e., *thinking beyond the interface* of the component is not required [RE06]. This means that the conceptual relations between the components can be segmented at the interfaces between the components. So built-in segmentation and chunking (see section 5.3.7) is required for a high degree of conceptual decoupling. Moreover, conceptual decoupling of a component will usually be improved if temporal decoupling of the component is provided, as considering timing issues across an interface increases the conceptual coupling of a component. For example, if a system allows remote function calls or similar mechanisms, the timing properties of these functions must also be considered, as well as access conflicts with shared resources. All the knowledge that may be required from other components must be analyzed when reasoning about the behavior of a component. However, such an analysis may involve a large number of components to be able to fully understand the temporal properties of the system. This characteristic might be very familiar to developers of large object-oriented systems. Control flow analysis in such systems typically involves more than just a single component and is inherently relational. These relational properties are not only present in object-oriented systems. Similar characteristics can be found in most event-triggered systems where control flow across interfaces can occur.

The direct invocation of methods of other objects that is not coordinated globally may lead to access conflicts, processing or network overloads that must be considered during system design. This is an issue that has not yet been solved satisfactorily for object-oriented approaches. So object-oriented systems are, in general, not composable [Los05].

While a component developer is working on a component with a high degree of conceptual decoupling, e.g., during initial development, all relevant

aspects of the component are either fully described by the interface specification or by those parts of the component implementation that are already available. The largest chunk that may be required is the component itself. This considerably supports the analysis of all possible behavior of a component as no references to other components must be resolved. For example, if the component specification just says that a component receives a temperature sensor reading in the form of an integer value every 10 milliseconds and that this number represents the temperature of a vessel in degrees Celsius, this is all required knowledge. However, if the component specification says that a temperature object of type `<temp_object>` is sent every 10 milliseconds and `<temp_object>` is not part of the component specification, this requires reference to the specification of `<temp_object>`. In this case the component specification is not self-contained any more. This means that reasoning about a component requires significant knowledge about other parts of the system that are not just attributive properties of the component under consideration.

So although the object-oriented approach at first glance seems to naturally support the principles of conceptual chunking and segmentation, it inherently requires *thinking beyond the interface* as objects are typically interwoven with each other: An object may access various methods and data properties of other objects, which requires a lot of knowledge about these objects to be able to understand them. This aspect is often ignored when discussing the advantages of the object-oriented approach. No clear conceptual separation between objects is usually possible due to the existence of relational properties. So it seems that object-oriented approaches are more suitable for intra-component software development where tight interdependencies are less problematic than for the specification of conceptually decoupled, composable components that shall be developed and verified independently.

Section 6.2.3 discusses temporal firewalls as an example for an interface that supports a high degree of conceptual decoupling of components.

5.5 Exploiting Regularity

As already presented in section 3.2.1, regularity is synonymous to conceptual simplicity as regularities allow a compressed (simplified, chunked) representation of the underlying information. In real-time systems development there are two different kinds of regularity that can support comprehension: The structure of the system can exhibit some degree of regularity, as well as the behavior of the system over time. In this section the implications of both kinds of regularity are discussed.

5.5.1 Structural Regularity

Structural regularities, i.e., if the components of a system have a similar or even equal structure, help a lot when having to comprehend a system as regularities allow to re-use conceptual knowledge and comprehension strategies from one component of the system to other components. For example, every component can have the same diagnosis interface with support for the same set of operations. So we just have to learn one set of diagnostic operations and can then apply this knowledge to all components. If every component has a different diagnosis interface, each of these interfaces must be learned separately.

Similarities in structure, for example, that every component consists of just one additional level of sub-components, can be used to simplify the understanding process as for each component a similar comprehension strategy can be used to identify the relevant parts and their interconnections. If all components are completely different, a new strategy must be developed for each single component, with no re-use between components.

Regularities should be made explicit, e.g., by using the same names for the same parts in different components. So identifying the same or very similar structures becomes recognition of already known parts, instead of learning new structures.

A large system with a regular substructure may be simple to create and maintain, whereas even a relatively small system with no obvious regularities is perceived as being far more complicated. As regularity means simplicity, the number of different component types in a system should be limited. Nowadays, the development of embedded real-time systems still follows a very customized ECU[3]-focused, or at least application-focused approach, resulting in rather isolated applications either at the ECU or application level, with little reuse of components (either hardware or software) across diverse ECUs, applications, and even product families. So very diverse component types must be handled.

Architectures and design patterns support the creation of systems with a regular structure as design concepts can be re-used. A high degree of regularity is a desired property of a comprehensible real-time computer system. From this perspective, all components of a system should have the same high-level structure. Low-level differences (e.g., different application jobs) implement the necessary behavioral differences that justify the creation of different components.

Architectural fault-tolerance by the exact replication of components is an example for high structural regularity, making use of completely identical parts.

[3]Electronic Control Unit

5.5.2 Behavioral Regularity

Regularity of behavior means that over time a system shows the same or at least similar behavior. For example, a component can become activated every 50 milliseconds, read a number of sensor values, perform some computations according to these values, and then send a message to an actuator. A static (cyclically executed) schedule introduces a large amount of behavioral regularity into a system.

Regularity in behavior helps in a number of different aspects. It reduces the behavior of a component as the number of possible action sequences is limited. A component that has no cyclically recurring behavior can have a different order of its actions every time it is activated, and it can be activated at arbitrary points in time. For comprehension, this additional variety can increase the problem space dramatically. If the behavior is regular, the number of concepts and comprehension strategies needed to understand the behavior of the system is limited. If there is more variety in behavior, a larger number of comprehension strategies is needed.

Let's have a look at an example where a static scheduling table is used: First, job `A` is executed, then job `B` and finally job `C`. Job `C` requires input data from jobs `A` and `B`. In this case the schedule ensures that job `C` always has up-to-date input data. All other job sequences can be ignored. They would only have to be considered, if a scheduling strategy with more possible job sequences was used. Moreover, errors can be tracked to a specific sequence, so they are easier to reproduce than if the sequence of job executions that has led to the error was unknown.

If the message patterns that are produced and consumed by a component are highly regular, these regularities are very helpful to develop a conceptual model of the component. We can induce the simplest categories that are consistent with the observed message pattern. Hence, a simple conceptual model can be developed by a highly regular message pattern. This is especially helpful for diagnosis and monitoring purposes, where no details about the component internals are known. In addition, a simple conceptual model of a component simplifies the use of component services in other components.

Another advantage of behavioral regularity is that the future behavior of a component can be predicted. So deterministic models can be used. If a component does not behave as expected, this is an indication for a failure.

Moreover, the exact prediction of future behavior is helpful if we have to ensure that a receiver of messages does not get overloaded. It is known exactly when the sender will submit messages. This knowledge helps to avoid message queues, timeouts, and similar mechanisms that are required if the timing of events is not known. Similarly, access conflicts for shared resources can be avoided so that mutual exclusion constraints do not occur at the job level.

In a highly regular system, developers are unlikely to overlook an aspect

that does not occur very often, such as a rarely sent message. Similarly, if most jobs of a component are activated regularly and not just in rare-event scenarios, this gives confidence that the system works as expected.

5.6 System Structuring

This section describes how system structure can influence the complexity of development tasks. First, hierarchies, layering and partitioning are discussed in general, then job-based system structuring is analyzed.

5.6.1 Hierarchies

Hierarchies play an important role in human thinking. On the one hand, concepts are usually organized hierarchically, and on the other hand the world is, in many aspects, organized hierarchically. Hierarchies are also present in computer systems. Sometimes they are just there because they cannot be avoided, e.g., a computer consists of a number of electronic components. Sometimes, hierarchies are introduced explicitly for the purpose of structuring a system. It has been recognized in many different areas that appropriate use of hierarchies can make systems more understandable [Sim81, Sim95, Res03].

Hierarchal relations can be combined with abstraction so that simplified views of underlying components can be achieved. Sometimes, hierarchies just represent an organization between related parts, without any immediate simplification. The latter is used, for example, to create inheritance hierarchies in object-oriented programming.

The introduction of hierarchies is not always useful. If hierarchies are just created for technical reasons, e.g., to enable the re-use of program code, this does not necessarily mean that this improves understandability. The problem with large hierarchies with many relational aspects is that it takes considerable effort to understand the relationships.

An example for hierarchies of concepts are inheritance hierarchies in object-oriented programming [JL04]. There exist multiple different lines of arguments which claim that the extensive use of hierarchies in the style of object-oriented programming is highly complex: *Nonmonotonic* reasoning, i.e., reasoning where conclusions have to be withdrawn by counterevidence, is required for understanding object-oriented systems as default values defined in superclasses can be overridden by descendants. This *reasoning to consistency* [JL04] can be considered highly complex – see section 3.2.7. Moreover, object-oriented hierarchies are highly relational structures as dependencies can span multiple levels of abstraction. So the relational complexity of development tasks can be very high. The level of abstraction usually does not increase with the number of inheritance relationships, just the number of dependencies. Another problem is that not all problems can

be structured to result in a useful hierarchy. So the resulting relationships can sometimes be considered accidental characteristics.

A system should only be structured hierarchically, if relations between subsystems that are far away from each other in the hierarchy are minimized, and clusters between related subsystems can be formed. If there are many relations in a system that involve various different layers of a hierarchy, this takes considerable effort to understand, as many parts of the system are involved. The relational complexity of these structures is usually high. To overcome these problems, a system can be structured horizontally into layers that hide global hierarchical relationships – see section 5.6.2.

5.6.2 Layering

Layering is a technique where different levels of abstraction are used within a system. The details of lower levels are hidden from higher levels. Layering is frequently used for communication systems where high-level interactions are built on a number of low-level operations so that the low-level details are hidden from the application. A famous example for layering is the OSI model [Int84].

An *abstraction ladder* represents views of different levels of abstraction on the same object [Hay91]. For example, a cow called *Bessie* can be seen as a *cow*, more abstract as *livestock*, or even more abstract as a *farm asset*. A computer system can also be considered at different levels of abstraction: Figure 5.2 shows the interrelationship between simplicity and complexity, depending on the respective position on the abstraction ladder. The figure depicts the *Four Universe Model* of a computer [Avi82], which introduces four levels of abstraction when modeling a computer system: The physical level is concerned with the analog signals of the circuits. The digital logic level is concerned with binary logic values, which represents a higher level of abstraction. The information level combines the binary values into meaningful data structures, and the external level represents the services of the computer system as seen from an outside observer. Each abstraction level allows a reduced view on the lower layer by ignoring irrelevant detail, which is an important aspect for complexity management. However, within each abstraction level complexity can emerge, depending on the number of elements and their interrelationships [Kop08]. So it is important to avoid wide steps on the abstraction ladder in order to keep complexity at a manageable level. In the ideal case, a modeling task requires just a single level of abstraction. If modeling tasks require moving up and down the abstraction ladder, the complexity of the tasks increases. This is a strong argument to limit the entities that are required for a given development task to a single level of abstraction. For example, for the task of system integration, the concept of system-level components can be used (see section 5.7.4).

Layering can be helpful to reduce complexity for two main reasons:

Increase the level of abstraction: Layering can reduce the complexity

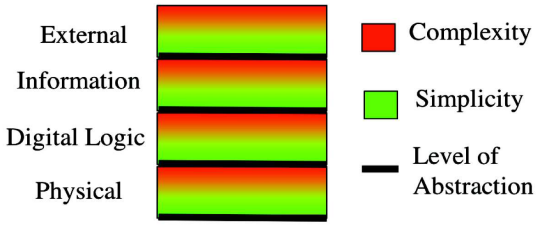

Figure 5.2: Abstraction ladder of the Four Universe Model (from [Kop08])

of tasks as it can increase the level of abstraction of tasks. For example, it is much easier for an application programmer to open an FTP transaction, than to directly access a communication controller that transfers Ethernet frames. All those low-level details, such as message encoding, or handling of communication errors, are irrelevant at the application level.

Hide differences: The introduction of an intermediate layer can hide differences between subsystems that would be different otherwise. For example, if a system consists of a number of similar but not identical components, or even some legacy components, an intermediate layer can hide the low-level differences so that all components can be treated alike. Similarly, device drivers make use of this principle of hiding different hardware implementations.

The introduction of an additional abstraction layer only makes sense for one of these two reasons. If a layer is introduced for a different purpose, it is very likely that the layer represents an indirection that impairs understanding.

If lower level layers are not fully hidden to the users of the higher level layers, or if there is considerable crosstalk between layers that do not directly sit upon each other this is also problematic for comprehension, as the relational aspects within the system are increased. To understand these relations, multiple steps on the abstraction ladder are required. Similarly, relational aspects in a system are also increased, if in a layered architecture the architectural services are not cleanly separated from the applications. This requires knowledge about the layers and the relationships between the layers.

Global hierarchical relationships can be hidden with a layered system structure so that just the immediate relations between layers are relevant. This helps to reduce relational complexity of development tasks.

5.6.3 Layering vs. Partitioning

Layering (also called *horizontal structuring*) can be used in combination with partitioning (*vertical structuring*). Partitioning alone may result in a large and flat system structure, so the introduction of layers may be useful.

However, the introduction of a layer must be considered carefully, so that conceptually coherent layers are created. Design concepts should not be scattered across various layers to avoid a large amount of relational aspects between layers. Pure layering does not result in an optimal system structure, because the encapsulation of components can be increased considerably with partitioning – both technically (see section 4.8.3) and conceptually (see sections 5.3.3, 5.3.6, and 5.3.7).

A good choice for the introduction of a layer in an integrated system is, for example, the conceptual difference between the application layer (consisting of various jobs of the same application) and the system layer (consisting of various applications). A good choice for the introduction of a partition in an integrated system is, for example, the natural border between applications, as each application is quite independent from other applications. This avoids a flat structure where different applications are mixed up at the same level of abstraction. Each application component (job) can also represent a partition if it is designed to operate nearly independent from the other jobs.

5.6.4 Job-Based Structure

An important level of abstraction for embedded systems modeling is the application level. Components introduce structure into a system. There exist two main approaches to structure application functionality into components – job-based and object-based structuring. Job-based structuring means that the application is decomposed into a number of jobs of equal level of abstraction, usually with standardized interfaces between the jobs [KOPS04, AUT06]. So at the job level a flat system-level structure is created (vertical structuring). An object-based (hierarchical) structure is used for systems that are developed according to an object-oriented design approach, see section 5.4.2.

An important advantage of a job-based system structure is that there exists just a single level of abstraction, so climbing up and down the abstraction ladder – which is typical for object-based structuring (see section 5.6.1)– is not required. As long as the application itself does not grow too large, this is the optimal solution. For today's embedded real-time systems this system structure seems most appropriate; unrelated functionality is separated into different DASs, so the resulting applications are not too large.

5.7 Component Integration Issues

In this section system and component characteristics are considered from the point of view of the system integrator. In the following discussion the term component refers to system-level component as defined in section 4.1.2.

A goal of any composable architecture must be to keep the complexity of component integration at the system level to a moderate and manageable

level. The system integrator shall not have to care about component implementation details, just assemble components to build up the system. The difficulty of this task largely depends on the characteristics of the components. More precisely, the degree of encapsulation of a component decides on the amount of knowledge that is needed about each component by the system integrator.

Moreover, it is explained how the effort of component integration can be minimized by avoiding design tasks during the component integration phase.

5.7.1 Attributive, Relational and Emergent Properties

At a given level of abstraction of a system we can differentiate between *attributive properties*, *relational properties* and *emergent properties*. Attributive and relational properties can be accessed directly and are usually also modeled directly. Emergent properties cannot be accessed directly but result from the interplay or at least from the existence of components or subsystems at a lower level of abstraction. So an emergent property represents a higher level of abstraction – the system level, where the components are integrated.

As the emergent properties can only be influenced indirectly – via the properties of the components – they are often hard to understand. A *relational property* conceptually belongs to the component level, just like an attributive property, but it only has a meaning with respect to other properties, i.e., a relation between the related properties must be established to be able to fully understand the property. The related properties can either belong to the same component or to other components.

An example for an emergent property is the load of the communication system of an integrated ensemble of components. An example for an attributive property is that a device supports a standard communication interface like USB or IEEE 1394. However, if a data sheet of a device states a serial RS232 interface, the user needs to adjust several relational properties, such as baud rate, parity mode, number of stop bits, and type of flow control between the two communication partners, which increases the complexity for setting up a working system.

It is important that the emergent properties can be deduced with just moderate effort from the characteristics of the components. If the interactions of the components result in *uncontrolled emergent properties*, the characteristics of the system level are hard to model, or can only be determined by simulation or prototype implementation. An example for an architecture with just controlled emergent properties is the time-triggered architecture [KB03].

Ideally, a component should have only minimal relational properties as this minimizes the relational complexity of tasks involving these attributes. Depending on the number of entities that are uniquely related to a relational

attribute, the arity and thus the relational complexity of the tasks increases.

5.7.2 Complexity of Component Integration

The overall essential functionality of a system can be considered constant. It cannot be removed by a good design. However, system design can influence the distribution of the functionality and how much accidental characteristics there are.

Often, there is a trade-off between interface complexity and implementation complexity, i.e., one can get a simpler interface if one is willing to pay implementation complexity for it [Ray03]. If a system has a certain functionality, it can be implemented either on one or on the other side of the interface. One of the basic questions of good design is where to locate the functionality. Of course, there is no solution to this question that is applicable to all kinds of systems.

The approach favored in this thesis is to implement all functionality within the components and leave no open functionality for the integration process. In other words, the integration of components should be a well-defined task, instead of a design task.

From the composability viewpoint, the complexity of component integration at the system level is an important aspect as it should increase just moderately when building up a system. When the complexity of component integration grows quadratically or even exponentially then the integration process of large systems is likely to fail.

Even when all the components are designed with very easy-to-understand interfaces, the integration process can be very complex, if the required functionality is not fully implemented within the components, or if the interfaces between the components do not match: It is thus design work that is left to be dealt with at the system level – by the system integrator. For example, wrappers around components may be needed in case of interface mismatches, or missing functionality might be implemented.

To avoid this propagation of design effort to the system level, the goal of the system architect should be to implement all functionality within the components, and provide matching interfaces so that the integration becomes a non-issue. This might now seem to be a platitude, however, most of today's computer systems are not developed according to this principle.

Another example for complexity of component integration is when unintended side-effects occur, e.g., caused by resource conflicts, such as temporary overloads of the communication system. In this case the principle of composability is not fulfilled. Such problems become apparent due to the integration of the components, so they are detected at a very late development stage, or probably even later in the field.

The current crisis of electronics systems development in the automotive industry can be attributed to ignoring complexity during component

integration: Every single component is easy to understand in isolation, but the emergent properties of the components are often extremely hard to understand, especially the timing behavior of event-triggered systems. This results in a high number of faults that are very hard to detect during system development and verification, and even harder when a vehicle fails in the field. It has happened to the author that a car regularly lost most of its power when driving on a motorway with a specific speed at temperatures around 5 degrees Celsius. Stopping and then restarting the engine helped to re-gain normal power. The reason for the error could not be found. Just with a software update months later that – according to the car manufacturer – had no specific fix for this particular problem, the error went away. It seems likely that some timing aspects that changed due to the update, had caused the problem.

5.7.3 Traditional Approaches

A typical approach to system design and integration is to deal with emergent properties, property mismatches or missing functionality at the system level by creating wrappers around the subsystems [Swa98]. Such an approach can be well-suited for the creation of systems mainly from legacy components, but it is fundamentally different from the concept of self-contained system-level components that is introduced in section 5.7.4: It does not avoid complexity at the system level, but tries to deal with various design problems at the system level, as part of the integration process.

Furthermore, temporal properties of components are usually changed due to the additional indirection of a wrapper. So the temporal properties that are established for the components are of no use at the system level. Hence, temporal composability of the components is not supported.

Moreover, a wrapper-based development approach does not result in a detailed architecture that can be used as a framework to easily implement a system from scratch: Too many aspects of system development are left undefined. No detailed framework is available that can provide a good guidance for the developers.

Another approach to complexity management is to use hierarchically structured software modules [PYJ04]. This solution has the disadvantage that most real-world systems can not use centralized control: Modeling a system with centralized control means to introduce dependencies where no such dependencies would be needed. Moreover, such an architecture does also not support temporal composability.

These traditional approaches make the integration process quite complex as lots of complexity caused by emergent properties, property mismatches, or missing functionality of the components must be handled by the system integrator.

5.7.4 Self-Contained System-Level Components

To be able to reduce the complexity of the integration process the concept of *self-contained system-level components* has been developed [Rum06]. Such components are self-contained so that they can be developed and verified in isolation and the integration is just minimal effort. Self-contained here means that no functionality is left to be implemented at the system level. This does not mean that the components have to be closed components, though. It must only be ensured that the SRLIFs of all components are connected to the respective SPLIFs of other components.

Such an approach is only possible if a strict two-level design approach as described in section 4.9.3 is used, and if the computer system architecture supports composability. So the components have matching interfaces and the properties that have been established at the component level also hold after the components are integrated.

Self-contained system-level components encapsulate all the functionality of the applications and avoid the propagation of (potentially complex) design tasks to the system level. The matching interfaces ensure that the component interfaces are clean boundary lines, without being blown up by connection systems. The additional indirection of the connection system usually leads to more comprehension effort regarding the interactions of components.

With a system-level component based approach, all components that must be used by the system integrator are on the same conceptual level, which reduces the cognitive load of the integration process. Of course, this is only possible if the component interfaces are fully specified and if no implicit interfaces are used.

Moreover, the distributed approach based on system-level components has an important advantage regarding fault-tolerance: The single points of failure that are introduced by hierarchic or centralized approaches as described in section 5.7.3 can be avoided.

5.7.5 System-Level Services

The system level is the aggregate level where the emergent services of the system-level components are realized. The system-level services that are required to integrate the components can be provided by the architecture, e.g., the communication service and clock synchronization. No controlled object interfaces are involved when just the LIFs of the system-level components are connected.

Such an architecture enables a separation of concerns: The system-level services, which are required for the integration, are provided by the architecture, the applications are only concerned with application functionality. The provision of the controlled object interface is the responsibility of the application.

5.7.6 Integration Levels

There exist two major approaches for the integration of components into a distributed real-time system. With a two-level approach as described in section 4.9.3, the integration of a component is always done in a single step, whereas in a layered architecture it is possible to integrate every layer separately, starting from the lowest layers and then moving upward to the higher layers.

The advantage of a pure two-level approach that strictly separates into system-level and component-level issues is that the integration is more or less a non-issue as the components all are designed to have matching interfaces and no open functionality or interface mismatches are left for the integration phase. All components that are used by the system integrator are at the same level of abstraction (system-level components), and all relevant aspects for the integration of the components have been fixed during the system design phase. The only task to be done is to verify the correct operation of the system, i.e., a system test.

In a layered architecture that does not follow a strict separation of system-level and component-level development, the aspects that are relevant for the integration are not fully specified during system-level design, as there are, by definition, more than two levels. So for each level a separate integration step can be done. This is how most of today's layered systems are integrated.

Table 5.1 shows an overview over the possible combinations of layered and single-level architectures with either an architecture that supports composability or with an architecture that does not support composability. Single-step means that all components can be integrated in a single step, whereas stepwise means that each component requires a separate integration step.

	layered	two-level approach
composability	layer-wise	single-step
no composability	layer-wise / stepwise	stepwise

Table 5.1: Overview of integration approaches

It is possible to combine a two-level approach with a layered architecture: If architectural services, such as fault-tolerant communication, are implemented in layers (transparent to the application components), this means that the system conceptually remains a two-level system regarding the integration process of the application. In this case the architectural layers should be generated automatically by the development tools. Similarly, a component can internally be structured to use layers (horizontal interfaces) and just present vertical interfaces to the system integrator. This also results in a single-step integration at the system level. If, however, such a component does not exhibit strong encapsulation regarding its internal structure during the integration process, e.g., by requiring additional configuration for

each layer, then the integration cannot be done in a single step.

5.8 Determinism

Determinism can be defined as follows: *The world is deterministic if and only if, given a specified way things are at time t, the way things go thereafter is fixed as a matter of natural law* [Hoe04]. Determinism is an essential factor that affects the comprehensibility of a system. In the area of embedded systems, a component (or model) is said to behave deterministically if, given the initial state t_0 and a sequence of future timed inputs, the outputs at any future instant t are entailed [Kop08].

5.8.1 Logical Reasoning

The identification of abstraction levels and the development of corresponding deterministic models, where the indeterminism of the world at the lower levels does have only a negligible effect, are at the root of scientific discovery and engineering practice [Kop08].

Deterministic models enable the exact prediction of future behavior, whereas non-deterministic models do not allow an exact prediction. Non-deterministic models require probabilistic reasoning, i.e., reasoning under uncertainty. As described in section 3.1.3 our minds are ill-equipped regarding probabilistic reasoning. So we should try to avoid the necessity for probabilistic reasoning in technical systems wherever possible. Furthermore, non-determinism requires a larger amount of mental models to describe the behavior of a system, which also accounts for increased complexity – see section 3.2.7. In other words, reasoning about non-deterministic systems is far more likely to lead to judgment errors than reasoning about deterministic systems.

Deterministic systems support the rational analysis of behavior, since determinism is a sufficient precondition for logical reasoning [Kop08]. When we reason about deterministic models, we can perform deductive tasks in which we perform very well, such as *modus ponens*[4]. Furthermore, it is easier abstract from deterministic models, as they exhibit less behavioral variety, which allows a more compressed representation. It is also more difficult to validate a non-deterministic system, since repeated test cases do not necessarily produce identical results [Kop08]. So it is hard, or sometimes even impossible to reproduce errors. Thus, the correctness of non-deterministic aspects is often left to be verified by different means, which usually involves human reasoning.

Non-determinism can be hidden within a system as a design error. For example, a programmer may not be aware that certain program constructs,

[4]The performance of other kinds of tasks, such as *modus tollens* is usually worse, but these kinds of tasks can be avoided and are usually not used very often.

such as the `select` statement in the Ada programming language [Int95] behave non-deterministically. These errors are often very hard to detect as they just become visible in certain (probably rare) scenarios, which might not be covered by tests. Thus, non-deterministic program constructs should always be avoided.

5.8.2 Consistency

In technical systems, determinism is often just present at certain levels of abstraction [Kop08]: The basic building block of computer systems, synchronous digital logic, is deterministic at the logic level. The non-determinism of the lower levels, such as voltage fluctuations at the physical level, that do not have an influence at the logic level, can be neglected. However, the determinism of the logic level is often sacrificed by the pressure to build high-performance processors, e.g., by implementing caching, pipelining and similar techniques. But also at the application level, non-determinism can arise due to inconsistencies of distributed data and due to improper handling of simultaneous events.

In order to achieve consistency in a distributed system, the proper handling of simultaneous events is very important [Kop08]: Mutual exclusion and the consistent ordering of messages are problems that are related to handling simultaneity. Whenever two events are perceived to have occurred simultaneously by a component of a distributed system, an additional rule must be put into effect to establish a uniform processing order of the simultaneous events. This additional rule must be applied consistently by all components of the distributed systems [Kop08].

Due to the denseness property of real time it is in theory impossible to ensure a system-wide consistent notion of simultaneity [Kop97, Kop08]. The sparse time model, which is discussed in detail in section 6.4, solves this problem by restricting the occurrence of relevant events that are in the sphere of control of the computer system. For events that are not in the sphere of control of the computer system, consensus or agreement algorithms [BDM93] must be used to re-establish consistency of distributed information.

5.9 Reducing the Problem Space

A reduction of the problem space is a widely accepted technique (e.g., suggested by cognitive load and mental model theories) to reduce the complexity of a task. In this section some generally applicable techniques are described.

5.9.1 Removing Irrelevant Information

The difficulty of reasoning tasks can be influenced severely by the presence of irrelevant information. If irrelevant information is processed, reasoning tasks

are more difficult than if no irrelevant information is present – see section 3.2.5. So not just the inherent characteristics of the underlying problem are relevant. It is also the way and the amount of information that is available, which influences the task performance. So often just skipping optional items in a design can help to increase understandability.

Abstraction is another important technique to reduce the amount of information. It is important to consider a system at the optimal level of abstraction. The optimal level depends on the task that must be performed. The term *level of abstraction* refers to a particular level of inclusiveness – the most useful level is the most abstract level at which the categories can mirror the structure of the problem domain [Ros78].

Interfaces should just present information that is required for a particular task. If very diverse tasks must be supported, the introduction of different interfaces is the only possibility to keep the amount of irrelevant information low. If the provision of different interfaces for different tasks cannot be implemented on the component side, it is possible to provide different views of a component with development tools.

5.9.2 Using Categories

Categorization is a very important aspect of understanding – see section 2.3. A system that makes good use of categories provides a good basis for comprehension.

Categorization of system or environment properties can reduce the problem space considerably. For example, if the environment can be modeled with just a few categories, this provides a more reduced view on the environment than if continuous dimensions must be used, or if many different categorical distinctions are required. Similarly, if the externally visible system state can be mapped to just a few categories, such as `startup`, `operational`, `shutdown` and `error`, the system state can be determined easier than if various detailed properties of the system must be analyzed. So categories can be used as a means for abstraction. They group related information into a single category and remove irrelevant diversity.

Another important use of categorizations is that similarities between parts of a system can be made explicit. For example, if two components use the same data type `vessel_temperature` we know that they both make use of the same semantics, such as a restriction to a particular range and rate of change[5]. If just a general integer type was used, these similarities would not be visible immediately, and the problem space for a developer would be larger until the similarities are discovered. Moreover, if similarities are directly reflected by the system this supports external cognition.

If different levels of abstraction must be used, a suitable framework of

[5]If the effort of specifying separate types is not feasible, or would result in a very large number of different types, at least the same names should be used for the same things.

categories has to be established for each level. Using categories at the wrong level of abstraction usually causes complexity, as a frequent switching of abstraction levels requires cognitive effort. So at each level, a consistent set of categories must be used.

5.9.3 Minimizing the Number of Model Interpretations

If there are many possible interpretations of a model, tasks are generally harder to perform than if there are less, or if there exists just a single interpretation – see section 3.2.7. The mental model theory says that problems are more difficult if they require reasoners to construct more mental models, due to the limited processing capacity of working memory. This is a very strong argument to reduce the number of possible mental models of a system. In general, this means to reduce a system just to essential characteristics, and to remove ambiguities and non-determinism – see section 5.8. So a highly deterministic, minimal system will be easier to understand than a system that exhibits a high degree of variation and non-determinism, which results in a large number of possible model interpretations.

For the development of real-time systems this means that predictability and behavioral regularity (section 5.5.2) are essential for creating systems with a low number of possible mental models. Whenever unpredictability and behavioral variety must be dealt with, this results in a high number of possible mental models of a system. At the application level, unpredictability is introduced, for example, by potential communication and component failures, a high degree of communication jitter, and non-deterministic observations of the times of event occurrences by different components.

Counting the possible states of a computer system can be done with *model checking* [CGP99]. However, model checking is not suitable to determine the cognitive complexity of tasks. One reason for this is that the number of states alone does not give a clear indication whether a system has good characteristics regarding comprehension, or not. A low number of overall states is a desirable property, but depending on the task that must be performed, just a limited number of states may be relevant. Issues that are relevant for understanding, such as chunking or segmentation, or providing different views of a component, cannot be modeled with the model checking tools that are available today. Furthermore, the number of states of the model may not correspond to the number of relevant states of a real-world computer system. For example, if the progression of time is modeled with a numeric counter in a computer system, this results in a very high number of states: Transitions between states that are effectively the same are counted multiple times if they span multiple time granules. Appropriate model checking techniques that do not have these drawbacks will have to be developed so that counting model states can be used as a cognitive complexity metric. However, it is yet unclear if it will ever be possible to create the required cognitive process models – see section 7.3.

5.10 Failure Handling

Appropriate handling of failures is an important aspect in the development of reliable embedded systems. In this section it is discussed how the handling of failures can influence the complexity of development tasks.

5.10.1 Stability of Models

The development of mental models of a system is inherent to system design. A deviation between the intended and the actual behavior of an artifact (a failure), leads to an upset of the model if the failure scenario is not part of the model, i.e., if only the fault-free system is modeled. To be able to understand the behavior of a system in case of a failure, a new model must be developed. As the development of new models means increased cognitive effort, we should aim at restricting the effects of a failure to a small scale, i.e., to contain the errors so that large parts of the system remain unaffected by the failure. It must be ensured that *the abstractions and models that are developed for understanding the system are stable even in case of failures* [Kop08]. This means that the system must be partitioned into independent fault containment regions to avoid error propagation to other parts of the system that are not directly affected by the fault.

The analysis of failures is simplified considerably if the effects of a failure are restricted to a small part of the system. If the whole system can be influenced by the propagation of errors then the identification of the initial cause becomes like finding a needle in a haystack.

5.10.2 Separation of Concerns

In a fault-free system no failure handling must be considered. This is the ideal case regarding system understandability as the system model can be much smaller and simpler than if various failure scenarios must be included. Unfortunately, completely fault-free systems cannot be achieved. However, it is possible to provide architectural fault-tolerance that can mask a large number of faults so that at the application level certain classes of failures can be ignored. So the models at the application level can be simplified considerably. In addition, this separation of concerns leads to more coherent application models – see section 6.3.3.

As an example, communication faults are a major cause for error handling in computer systems. However, this class of faults can be fully ignored if fault-tolerant, reliable communication is provided to an application. So applications can be modeled assuming an ideal (fault-free) communication environment.

5.11 Chapter Summary

Architectures and standards support the establishment of a well-defined conceptual framework for system development. This is only possible if the network of basic-level concepts can be identified easily. Hence, appropriate semantic descriptions of the core concepts are needed to make an architecture easily understandable for novices.

An architecture reduces the effort of the design process as the most fundamental decisions have already been made. So the design freedom is restricted, which reduces the size of the problem space. An architecture makes it possible to reuse comprehension techniques, concepts and whole subsystems across multiple components of a system.

Component interfaces not just serve as technical border lines between components, they can also support the psychological concepts of segmentation and conceptual chunking. Chunking is related to simplifications provided by the interface. Segmentation on the system level means to split the large system into smaller conceptual units. If these conceptual units correspond to the technical units (components) that build up the system, this system structure can be seen as built-in segmentation and chunking: The border lines between the components correspond to the conceptual segmentation lines, and the chunking is provided by the abstraction of the interface. Built-in segmentation and chunking supports the conceptual decoupling of components by minimizing relational component properties and by providing a conceptual structure that is easy to identify. External cognition is supported as the conceptual units that are needed for understanding are reflected by the system structure itself and do not just exist as mental representations.

Interfaces can reduce the relational complexity of various tasks at the component interface level if relations are encapsulated within a component. The elimination of relational component properties is particularly important for the system-level components that are used by the system integrator.

Interfaces can provide a simplification of the component if they encapsulate component-internal details. Abstraction of the environment is another important purpose of an interface. Component development can be simplified if the interface provides a reduced and stable view of the environment.

All component interactions should be defined explicitly in the component interface. This is especially important as understanding means seeing relevant connections. If the connections between components are not apparent at the surface, understandability suffers. Reducing the number of relevant connections between components is an important design principle. This can be achieved by a high degree of conceptual decoupling between components.

A comprehensible component interface should provide semantic coherence so that it can represent a single chunk at a high level of abstraction. Task-level inclusiveness is required to remove unnecessary details from the

interface on the one hand and to provide all necessary information on the other hand. So different views on a component can be supported best by the provision of different interfaces.

Implicit interfaces should be avoided as they require the consideration of the component implementation level. If all interactions occur through explicitly defined interfaces, the level of abstraction can be increased to the component interface level for all users of the component services.

Almost all embedded real-time systems have a high degree of interdependencies between subsystems. This is due to the characteristics of control applications that are typically performed by these systems. So data dependencies cannot be removed from these systems. The introduction of suitable interfaces and the provision of temporal decoupling can break up the system into nearly-independent subsystems. If these nearly independent subsystems have interfaces that support conceptual decoupling and provide a full conceptual description of the environment, independent development of components is possible. No additional information about other parts of the system or the environment is required. This is essential to reduce the cognitive load of component development.

The integration of components that are developed and verified in isolation can only proceed with minimal effort if the architecture supports temporal composability, if component interfaces match and if all required functionality is encapsulated within the components. So the integration process is just a well-defined task of interconnecting component interfaces, instead of a design task with unforeseeable complexity. Hence, the integration process is straightforward and connection systems can be avoided. Such an approach is possible with self-contained system-level components.

An architecture that supports composability enables the use of stable models across various development phases. This is a considerable advantage compared to an architecture that does not support composability, where different models of a component must be used, depending on whether a component is considered in isolation or as an integrated part of the system.

The development of a computer system requires modeling at different levels of abstraction. In order to reduce the complexity, each development task should just require a single level of abstraction. Tasks that require climbing up and down the abstraction ladder are inherently more complex.

Determinism is an essential factor that affects the comprehensibility of a computer system. Deterministic models enable the prediction of future behavior, which is a prerequisite for logical reasoning about behavior. In addition, it is easier to derive abstractions from deterministic models than from non-deterministic models. Non-deterministic models require probabilistic reasoning for which our minds are ill-equipped. Moreover, non-deterministic systems are more difficult to validate than deterministic systems.

The proper handling of simultaneous events is essential to achieve consistency in a distributed system. Simultaneous events must be processed in

the same order by all components to avoid inconsistencies. If inconsistencies of distributed data cannot be avoided, e.g., as the data sources are not in the sphere of control of the computer system, consensus or agreement algorithms must be used.

There are two kinds of regularities that can be exploited to simplify embedded real-time systems: The structure of the system can exhibit some degree of regularity, as well as the behavior of the system over time. Regular structures represent concepts that are easier to understand than structures with no regularities. Moreover, they support the re-use of concepts and comprehension strategies across components. Behavioral regularity, such as the cyclic execution of a static schedule, reduces the possible component behavior.

The problem space for component development and verification can be reduced if irrelevant information is removed wherever possible, and if the number of possible model interpretations is minimized. Determinism and consistency are very important to keep the number of models low. In addition, making use of categorization can increase the level of abstraction and makes relations between components explicit.

It is important that the abstractions and models that are developed for understanding a system are stable even in the case of failures. This can be achieved either by including the failure scenarios into the models, or by modeling just the fault-free case and providing appropriate systematic fault-tolerance mechanisms.

Chapter 6

DECOS Analysis

In this chapter the theoretical considerations of chapter 5 are used for an analysis of the DECOS computer system architecture. So the complexity management techniques can be considered in the context of a specific computer system architecture.

First, section 6.1 introduces the basic architectural concepts. Then the remaining sections discuss various aspects of these concepts in detail.

6.1 The DECOS Architecture

The DECOS integrated architecture [KOPS04, POT+05] is a framework for the development of distributed embedded real-time systems. It was developed by the EU Framework Program 6 *Dependable Components and Systems (DECOS)* research project.

6.1.1 Physical System Structure

The physical building blocks of the DECOS architecture are *clusters*, *physical networks*, *components* and *partitions*. In this thesis the term *node computer* is used instead of component, as component is defined in a more general sense than just a node computer – see section 4.1.2. A cluster is a distributed computer system that consists of a number of node computers that are interconnected with a physical network. Each node computer has its own dedicated hardware (processors, memory, communication interfaces, controlled object interfaces), and software (application programs, operating system, middleware).

6.1.2 Functional System Structure

In the DECOS architecture, the overall real-time computer system is divided into a set of encapsulated Distributed Application Subsystems (DASs),

each with dedicated computational and communication resources. There exist two different categories of DASs – safety-critical and non-safety-critical DASs.

The communication resources for the DASs are provided as *virtual networks* [OPK05b], which are implemented as overlay networks on top of a time-triggered physical network. In each virtual network a communication protocol tailored to the needs of the respective DAS is provided. The partitioning of the overall system into DASs with dedicated virtual networks ensures a strong encapsulation of each DAS, enabling a functionally independent operation of each DAS. This strategy supports the independent development of the applications as each DAS can become the responsibility of a corresponding supplier. Moreover, error containment between DASs is ensured as a message failure in one DAS cannot propagate to another DAS.

A DAS, such as a steer-by-wire system, is further decomposed into a number of jobs – see figure 6.1. These jobs interact with each other via the communication system provided as a virtual network. The interface between the jobs and the architecture is called the *platform interface*. Via this interface the architectural services are provided to the application jobs. The second interface type a job can have is the controlled object interface.

There are no direct interfaces between the jobs of a DAS, all communication is redirected via ports. Depending on the data flow direction, one can distinguish between input ports and output ports. For the safety-critical DASs all ports are state message ports. The application is never interrupted by incoming messages and can disseminate outgoing messages without relying on a control signal from the communication system. The points in time when the ports are accessed by the jobs are determined by the application. Message transmissions are performed according to the information push principle, message receptions are performed according to the information pull principle. The message transfer to other jobs is done autonomously by the communication system. For non-safety-critical DASs event message ports are also supported.

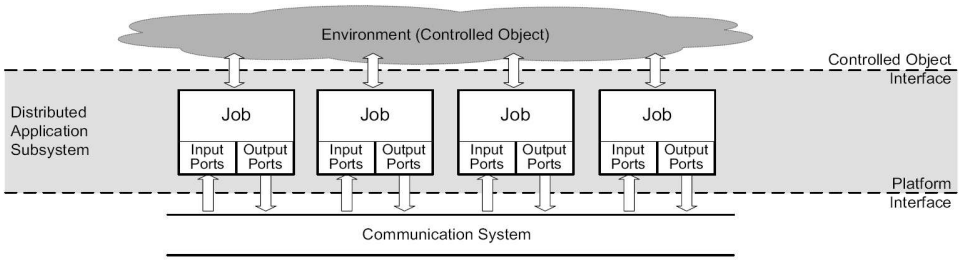

Figure 6.1: The structure of a DAS

6.1.3 Architectural Services

In the DECOS architecture, generic architectural services separate the application functionality from the underlying platform technology, which corresponds to the concept of *platform-based design* [SVM01]. So the application job behavior can be modeled independently from the underlying platform technology, i.e., independent from the implementation. Thus, highly portable application models can be created.

The architecture is built around four core services [KOPS04]

Deterministic and Timely Message Transport: This service enables the transport of state messages from the CNI of the sending component to the CNIs of the receiving components. The fault-tolerant message transport service is provided by a time-triggered communication system, such as the TTA [KB03]. Temporal firewall interfaces eliminate control error propagation and minimize the coupling between components.

Fault-Tolerant Clock Synchronization: As the operation of time-triggered systems requires a notion of global time, clock synchronization is a fundamental service [KO87]. This allows to establish a sparse time-base where the activity intervals form a globally synchronized action lattice.

Strong Fault Isolation: As the effects of a fault can propagate across fault containment region boundaries as erroneous data, error containment is required. The error detection mechanisms must be located in other fault-containment regions than the message sender.

Consistent Diagnosis of Failing Nodes: The membership service provides consistent information about the health state of nodes. It is based on a priori knowledge about the points in time of the time-triggered message send times. If a receiver does not receive a correct message at the expected points in time, the sender is considered erroneous.

Based on these core services, numerous high-level services can be provided, that can be tailored to the needs of the respective DAS [KOPS04]. As depicted in figure 6.2 these services are provided to the DASs via a *Platform Interface Layer (PIL)*:

Encapsulation Service: This service is responsible for spatial and temporal error containment at the node computer level. Error containment is provided between the safety-critical and the non safety-critical subsystem. Computational resources as well as communication bandwidth is allocated statically; control and information flow may only occur from the safety-critical to the non-safety-critical subsystem, but not the other way round. Error containment is also provided between the

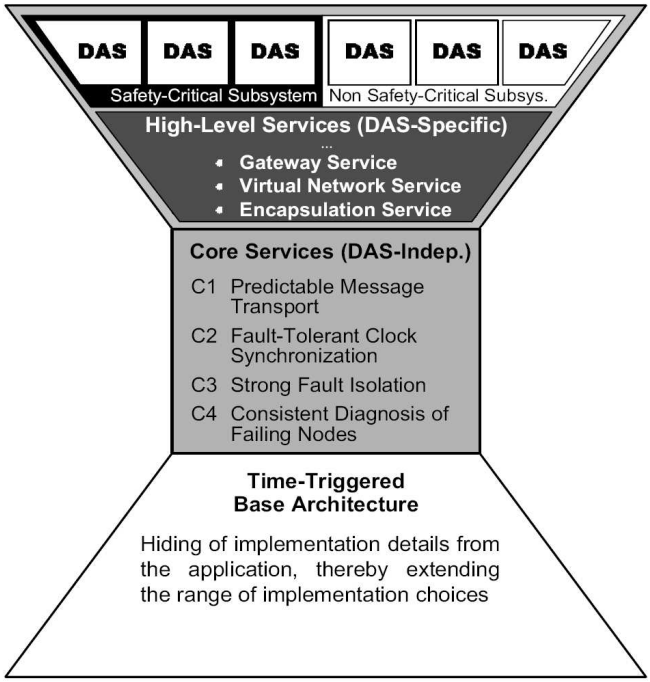

Figure 6.2: The DECOS architectural services (adapted from [OPK05b])

jobs at each subsystem by managing the access of each job to the shared resources. It is ensured that each job executes within a protected partition and the interactions between the jobs are restricted to the controlled message exchange via ports. Intellectual property protection is also provided by the encapsulation service.

Virtual Network Service: Virtual networks are provided as overlays on the time-triggered backbone network. They provide error containment at the network level. Each network has a separate namespace, which supports the independent development of DASs, and the integration of existing namespaces of legacy applications.

Hidden Gateway Service: Gateways are necessary to achieve communication between different subsystems. A hidden gateway performs the interconnection at the architectural level. So it is transparent to the jobs at the application level.

Fault-Tolerance Service: Architectural fault-tolerance is provided via a dedicated fault-tolerance layer. If the application supports replica determinism, redundant groups of jobs can be managed and failures can be masked by N-modular redundancy.

Diagnostic Service: The high-level diagnostic service comprises a diagnostic acquisition service and a diagnostic dissemination service. So it exceeds the node-level granularity of the core diagnostic service. The acquisition service monitors the port state of each job by evaluating

the incoming and outgoing messages against the interface specification. In addition, the fault-tolerance mechanisms of the fault-tolerance layer are also monitored. This is necessary as the masking of failures is transparent to the applications and errors would remain undetected. The dissemination service is responsible for forwarding the diagnostic information for analysis via a dedicated diagnostic DAS.

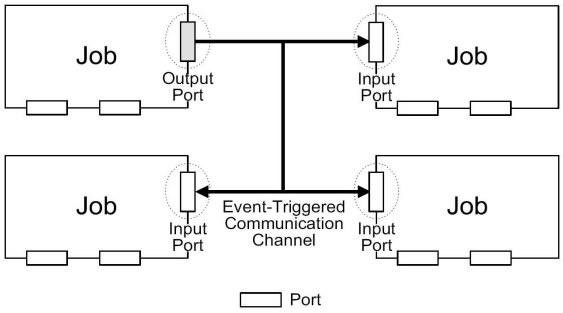

Figure 6.3: Jobs connected by an ETCC (from [KOPS04])

6.1.4 DASs and Virtual Networks

All communication between jobs in the DECOS architecture is achieved via virtual networks, which are overlay networks on top of a time-triggered physical network, e.g., based on TTP [TTT04] or Time-Triggered Ethernet [KAGS05]. Each virtual network supports a corresponding communication protocol that is tailored to the needs of the respective DAS regarding its functional, operational (temporal and syntactic) and meta-level properties (dependability, semantics of the interface data structures) [OPK05b, OP06].

A virtual network provides an independent namespace for each DAS [OPK05b]. All communication between a set of jobs is private within the DAS, so transmissions or receptions of messages can only occur between jobs of the DAS, unless a message is explicitly exported or imported by a virtual gateway [OPK05a]. So a virtual network provides encapsulation at the network level. No interference between different virtual networks is possible.

The provision of encapsulated virtual networks for each DAS supports service optimization [OPK05b]. This means that the communication requirements of different DAS, which can be quite diverse (e.g., when integrating legacy applications), can be supported ideally.

The virtual networks can be either time-triggered or event-triggered. For safety-critical DASs only time-triggered communication is supported due to the advantages regarding predictability, error detection, fault-tolerance and replica determinism. For non-safety-critical DASs either time-triggered or event-triggered communication can be used.

Event-Triggered Communication Channels (ETCCs) [KOPS04] are used to exchange event messages between the jobs of a DAS with event-triggered

ports. An ETCC implements a one-to-many communication relationship with exactly one output port and n input ports. So the job with the output port acts as the exclusive sender of the ETCC. Its messages can be received by all other connected jobs – see figure 6.3. An ETCC provides an elementary interface with unidirectional information and control flow and so it acts as an error-propagation boundary. Errors of the message consumers cannot propagate back to the producers and affect their operation. As dedicated ETCCs are provided for each sender job, there is no need for authentication mechanisms as masquerading failures cannot occur. In addition, the use of dedicated ETCCs prevents a faulty sender job from causing omission failures of correct jobs as bandwidth of other ETCCs cannot be acquired [KOPS04].

Multiple ETCCs can be combined to implement high-level event-triggered protocols that include services such as flow control or retransmissions in case of failures. Of course, then the advantages of elementary interfaces are lost.

An architectural gateway can either be a *virtual gateway* within a cluster, or a *physical gateway* that interconnects the physical networks of different clusters. A virtual gateway is always a gateway between two different DASs as each DAS has just a single virtual network. A physical gateway can be a gateway within a DAS, or between different DASs. Virtual gateways allow for the controlled coupling of the virtual networks of two or more DASs [OPK05a].

Virtual gateways resolve the differences in the operational specification of different DASs. A gateway can be introduced, for example, between a DAS with a time-triggered virtual network and a DAS with an event-triggered virtual network. So the gateway has to map the different temporal specifications by queuing messages that are received from the event-triggered virtual network so that they can be forwarded to the time-triggered network in a controlled fashion [KOPS04].

The introduction of virtual gateways is conceptually different to placing the jobs of the corresponding DASs into a single, common DAS. First, the export of information can be controlled as the virtual gateway selectively redirects the information. So it provides a reduced view on just that information that is really required. Second, virtual gateways prevent error propagation across DASs.

6.1.5 Development Process

DECOS employs a *model-based design* [OMG01], starting with a *platform-independent model (PIM)*, which is then transferred to a *platform-specific model (PSM)*. UML is used to describe both the PIM and the PSM. Moreover, UML-based meta-models is are provided for the description of both the PIM and PSM [TU 05]. These meta-models serve as constraints and guidance for system design, i.e., they are a formal representation of the architecture that can be used by design tools.

The development of a DECOS system starts with a PIM of each DAS – see figure 6.4. A PIM is a formal specification of structural, functional and meta-level properties of a system that abstracts away technical details. It is the starting point of system architecture conceptualization. When the functional and dependability properties of a system are clarified, the exact physical structure can be fixed, i.e., the functional elements (see section 6.1.2) must be mapped to the physical building blocks (see section 6.1.1).

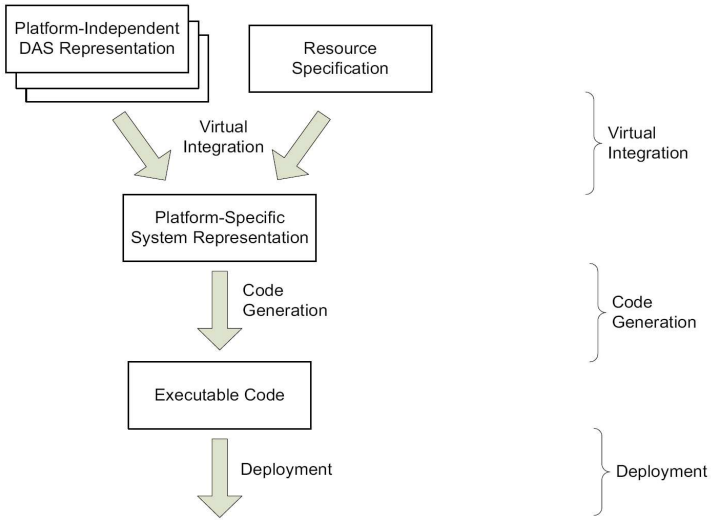

Figure 6.4: DECOS system development process (from [HOP06])

In the PIM, the structural units that are used for modeling are DASs and jobs. A focus is put on the specification of the linking interfaces, i.e., the ports between the jobs, and the specification of virtual gateways between DASs [HOP06]. These interface specifications capture the operational properties (syntax, temporal constraints, interface state) and meta-level properties (e.g., dependability) of the messages exchanged via the port.

The application functionality itself can be modeled with appropriate modeling tools, such as SCADE [Tec07]. For this purpose, the PIM can be imported into the tool. So a correct mapping of PIM and applications can be ensured. Modeling the applications can be achieved at the PIM level, as long as no platform-specific functionality is included in the applications. From the formal models of the applications, application program code can be automatically extracted with code-generators certified according to DO178B level A [RTC92]. Directly integrating C-code is also possible.

Simulation that uses the formally specified models can be done with tools such as Simulink [Mat04]. So even simulation is possible at the PIM level.

A *resource specification* describes the availability of resources, such as processor, memory, communication bandwidth, or special purpose hardware. It provides building blocks for system composition in a library [HOP06].

The high-level model of the PIM is transformed to a PSM with tool support, thereby taking into account resource constraints imposed by the

resource specification. This transformation is called *virtual integration* [GFL+02] as the integration is not performed on physical hardware, but operates on a virtual platform described by the resource specification. The purpose of the virtual integration is to find a feasible allocation of jobs to partitions and mappings of virtual networks to the time-triggered core network [HOP06]. Based on the specification of the available resources and on the resource requirements of the PIM, several checks can be performed that allow the identification of infeasible integration results at an early stage of the development.

The PSM extends the PIM regarding the allocation of jobs to partitions, the mapping of virtual networks to the time-triggered core network, and the parameterization of high-level services, which are selected and configured during the transformation [HOP06, HSS+07].

A problem for modeling hardware characteristics at a high level of abstraction is to establish a common view on the meaning of a particular characteristic. For example, just specifying a concept such as memory access time may not be sufficient, as for some hardware implementations the access time may be constant, while for other implementations the access time may depend on the location of the memory element. Such differences are especially problematic if diverse hardware is used. To achieve common semantics for resource primitives, the DECOS approach supports – besides a natural language description – links to *technical dictionaries* [HOP06]. These dictionaries are used to establish a common understanding of the resources and their characteristics. An example for such a dictionary is the Component Data Dictionary of the IEC 61360 standard.

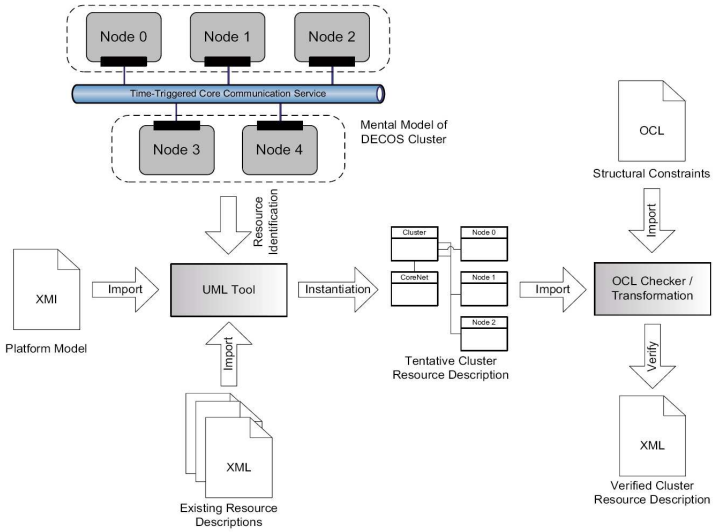

Figure 6.5: DECOS platform modeling workflow (from [HOP06])

The workflow of platform modeling is depicted in figure 6.5. The modeling process is supported by tools that provide the respective meta models and resource building blocks, and is guided by the constraints of the meta

model. In addition, an OCL checker ensures that additional constraints that can not be specified in the UML meta model are fulfilled by the cluster resource specification. So modeling errors can be identified early in the design workflow. The constraints restrict the structure and interconnection of the objects as well as the possible attribute values [HOP06]. So a verified cluster resource specification can be obtained, which is used for the subsequent virtual integration – see figure 6.4.

6.1.6 Diagnostic Strategy

In the DECOS architecture, a node computer represents a *Field Replaceable Unit (FRU)* with respect to hardware faults, and a job a FRU for software faults. To identify faulty subsystems, a diagnostic framework is part of the architecture. Due to the fact that the fault-tolerance functions of the architecture are transparent to the applications, the diagnosis of these services must be done at the architectural level.

The diagnostic service is provided at the network and application level [KOPS04]. The acquisition of diagnostic data is achieved via a dedicated virtual diagnostic network and the analysis of the data in a dedicated diagnostic DAS. So-called *Out-of-Norm assertions* [POK05] are checked against the interface state of the ports to indicate that the actual state deviates from the expected one. So a holistic view of the system can be established by operating on the distributed state. This allows to trace system anomalies to the responsible FRU.

The provision of a separate diagnostic DAS has the advantage that no probe effect can be introduced: At first, the diagnostic DAS cannot disturb the real-time functionality as it is fully encapsulated like every other DAS. Secondly, the purely virtual solution ensures that no additional hardware failures can be introduced, e.g., due to erroneous wiring or connector problems.

As the diagnostic DAS cannot interfere with the real-time functionality of other DASs, a certification of the diagnostic DAS to the highest criticality levels is not necessary.

6.2 State Message Ports

In the DECOS architecture, state message ports are required for the interfaces between the jobs of a safety-critical DAS. In this section the most important advantages of state messages and state message based interfaces regarding comprehensibility are discussed.

6.2.1 Conceptual Simplicity

State messages are a very simple means of communication between the components of a system, especially for control systems where lots of state information of various real-time entities must be exchanged. The simplicity of state messages can be explained with relational complexity theory: Processing state messages has lower relational complexity than processing event messages that describe the state changes of real-time entities. A state message is self-contained as it contains the full information about the state of the real-time entity. An event message just contains information about a state change. It is thus necessary to integrate the history of state changes to be able to reconstruct the full state of the real-time entity. In terms of relational complexity theory, a state message thus represents a unary relation. It requires no knowledge about the history of state changes and can be used immediately. An event message has a higher arity, depending on the steps required to reconstruct the state of the real-time entity.

It depends on the use case whether state or event information is required by the receiver of the message. If possible, a receiver should be designed to rely on state messages only to achieve the aforementioned simplification concerning the integration of past events. In general, control algorithms require the full state of real-time entities more often than event information, which is, conceptually, just incomplete state information.

6.2.2 Unification of Interfaces - Temporal Firewalls

The state message ports of DECOS jobs represent *temporal firewalls* [KN97], which are pure data sharing interfaces where control flow across the interface is avoided by design. So control errors cannot propagate across the interface.

As a temporal firewall just represents a collection of state messages, no complex interaction mechanisms are introduced. State message transfer is the only communication mechanism that is required in a safety-critical DAS. No additional interaction mechanisms must be considered. So all interactions are covered by a single mechanism.

An interface such as a temporal firewall, which supports periodic state messages with error detection at the receiver, represents an elementary interface. Composite interfaces are inherently more complex than elementary interfaces as the correct operation of the sender depends on the control signals from all receivers. E.g., an event message interface with error detection is a composite interface where even unidirectional data flow requires bidirectional control flow.

A temporal firewall allows for a high degree of temporal decoupling between components. Control flow analysis across component interfaces is not necessary. Independent timing is possible on both sides of the interface as long as it is ensured that the state messages are temporally accurate. This largely reduces the effort of component development as the consideration of

timing and control flow aspects is reduced to a single component.

6.2.3 Conceptual Decoupling

An important characteristic of state messages is that they support the *conceptual decoupling* of components – see section 5.4.2. Conceptual relations between components can involve semantic relations and temporal dependencies. As described above, state message interfaces support temporal decoupling as timing issues can be considered at a component-local scale.

The self-containedness on the semantic level is achieved due to the fact that a state message contains the full state of a real-time entity. So a state message port can be fully described by the port specification in the time, value and meta-level domains. No references to other component interfaces are necessary. This is a prerequisite for the independent development of components – see section 5.4.2.

An interface that makes use of more relational concepts than pure state messages does not provide the same high degree of conceptual decoupling. For example, if component `A` sends a notification message to component `B` which indicates that some data is ready to be transferred to component `B`, then component `B` requires knowledge about how to retrieve the data from component `A`. So a specification of the interface of component `B` without reference to the specification of the interface of component `A` is not possible. Thus, the conceptual coupling is higher than for a pure state message interface where the data is directly transferred as state information from the interface of component `A` to the interface of component `B`.

6.2.4 Interface State

In a DAS where only state message ports are used, every job exposes its declared state at its ports. This characteristic has the benefit that this part of the application state is easily accessible without the need for explicit queries via a separate monitoring interface. So the number of job interfaces can be minimized. The declared interface state can be used directly for analysis, e.g., by the diagnostic DAS or for error detection.

If the interface state is used for diagnosis, the analysis can be done at level of abstraction of the component interface. No low-level knowledge of component internals is required, so the abstraction of the interface holds even for diagnosis.

6.2.5 State Message Objects

The level of abstraction of pure state message ports is, in general, quite low. For many applications this is the optimal level of abstraction. However, sometimes a higher level of abstraction may be useful.

Additional abstraction layers must be added onto a set of state messages to enable built-in chunking. This can be achieved by using *state message objects*, which represent aggregate data structures consisting just of collections of related state messages, but represent a higher level concept. For example, a state message object `airspeed` may contain related data around the current airspeed of an airplane. The collection of data could comprise an agreed value collected from multiple sensors and also the single sensor readings. So a single state message object can hold multiple state messages that belong together, representing a single chunk. At the level of abstraction of the interface, a state message object is an atomic unit. This avoids that the interfaces are blown up with large collections of similar state messages. So the level of abstraction of state message interfaces can be increased and the structure of the data supports built-in chunking.

6.3 Architectural Service Provision

As described in section 6.1.3, the DECOS architecture provides a number of architectural services to the applications. In this section it is discussed, how the provision of the architectural services can help to reduce the complexity of various development tasks, compared to systems where these architectural services are not available.

6.3.1 Reduction of the Problem Space

The provision of the architectural services reduces the problem space for application development. For example, the system-wide notion of time can be used by an application. Hence, the application does not have to implement timers or similar mechanisms at the application level.In addition, a lot less application-specific fault-tolerance is required due to the fault-tolerance service of the architecture. The fault-tolerance layer provides fault-tolerant message transfer, so an application just needs to handle each fault-tolerant message send and receive operation like a normal message transfer operation. No additional functionality, such as retransmissions in case of errors or replica handling are required. This is all done transparently by the systematic fault-tolerance service. So these kinds of problems do not need to be considered as part of the application design.

6.3.2 Support for Determinism

It considerably helps to simplify the design problem at the application level, if the architectural services enable the creation of deterministic systems. Determinism allows to establish a consistent distributed state [MBSP02], which is essential to reduce the complexity of distributed real-time system design. The consistent distributed state simplifies error detection, state recovery and diagnosis. A consistent state is easy to achieve in a single-

node computer system, but in a distributed system message transmission times, different local clocks and lost messages are examples why consistency cannot be assumed in general.

The DECOS architecture supports deterministic and timely message transport, a consistent diagnosis of failing nodes and a sparse global time. Together with the fault-tolerance service these architectural services enable the development of highly deterministic applications.

In addition, the provision of a communication service with constant delay and minimum jitter supports control stability in real-time applications. Jitter brings an additional uncertainty into a control loop. This uncertainty of course requires additional comprehension effort.

6.3.3 Separation of Concerns

The applications are not blown up with those services that are provided by the architecture. Hence, they can implement pure application functionality, which allows to develop *conceptually coherent applications*.

For example, errors that are not application errors, such as the loss of a message due to a transient communication error caused by electromagnetic interference, are not the responsibility of the application. Handling such errors conceptually rather belongs to the message transport service than to the application. So the architectural fault-tolerance service helps to separate concerns and provides an ideal (simplified) environment for the application developers. Furthermore, applications are reusable more easily: Their essence, i.e., the basic domain-level and implementation concepts are easier to see and to understand if they are not bloated with conceptually unrelated fault-tolerance functionality.

If those services that are provided by the architecture were implemented at the application level, the application functionality and the service implementation would not be clearly separated. So developers and maintainers would have to separate the two mentally. This means increased cognitive load and additional effort to develop appropriate segmentation techniques. The architectural services provide natural, i.e., built-in, segmentation between generic, reusable services and application functionality.

6.3.4 Support for Structural Regularity

As application and architectural service functionality are cleanly separated from each other, they can be verified and re-used independently. In a system where there is no clean separation, increased design variety is likely, as it is the responsibility of the developers to implement the same functionality in the same way – especially if multiple developers create different parts of a system.

For example, systematic fault-tolerance supports a regular system struc-

ture leading to less design variety, as the fault-tolerance functionality provided by the architecture can be reused all over the system. Application-specific fault-tolerance is likely to be implemented differently in different parts of the system.

Moreover, if multiple developers create different parts of a system without systematic fault-tolerance, it is additional effort to ensure the correct function of the fault-tolerance functionality. Each developer may have a slightly different conceptualization of the fault-tolerance functions, so their correct interplay must be ensured. This additional verification effort is not required if the fault-tolerance service provided by the architecture is used.

6.3.5 Simplified Diagnosis

In current automotive systems, typically more than 50 percent of the program code in an ECU is related to diagnosis [PBC+02]. So by providing a diagnostic framework as an architectural service, a considerable reduction of the application size and thus the development effort can be saved.

As described in section 6.1.6, the diagnostic DAS allows the easy identification of faulty FRUs by providing a level of trust for each component. So maintenance engineers are supported in their decisions which FRUs need to be replaced. In addition, the error rate during a given time interval can be determined. This allows the tracking of transient errors and provides a basis for detecting wearout symptoms. Tracking these errors at the application level is only possible when the fault-tolerance mechanisms are also provided at the application level. However, as described above, this would make the development far more complex. Diagnosis at the architectural level can be added to a system without adding any complexity to application development. The holistic view of the system (e.g., by monitoring a subset of the distributed state of various applications) can only be achieved via architectural means. A holistic view is essential to be able to determine the health state of the overall system.

6.3.6 Stability of Models

The support for composability enables the stability of application models regarding the integration process. The architectural fault-tolerance service and the encapsulation service ensure fault containment and failure masking so that the application model remains stable even in the presence of faults.

6.4 Sparse Global Time

A sparse global time-base provides an abstraction of real time. In the DECOS architecture, the sparse global time is available to all components of the distributed system. In this section the advantages of a system with

a sparse global time-base are compared to a system where no sparse global time is available.

6.4.1 Natural Model of Time

A globally available notion of time most closely resembles our natural understanding of the concept of time, which is that of an omniscient outside observer with a single reference clock. This reference clock is used to timestamp all observations in a system, no matter which component observes an event. So if two or more components of a distributed system observe the same event, from the omniscient observers' viewpoint the event observation must get the same timestamp by all components.

If a system is based on a dense time-base, each component may assign different timestamps to the same observation as the local clocks of the components cannot be fully synchronous. So in effect, a system where there is no global time-base that allows to set identical timestamps to events by all components does not model real time very closely: Assigning different timestamps to one and the same event is just an accidental characteristic that is rooted in the technical implementation of the time sources. This has nothing to do with the model of real time we are all used to. So all effects that are rooted in these inaccuracies must be considered accidental characteristics that just complicate various development tasks.

The model of time provided by a sparse global time allows to observe all events with regard to a single, globally available time-base. From the viewpoint of the outside observer the single time source more closely resembles our natural concept of real time than various local clocks that are not fully synchronous. So we can reuse our knowledge about real time and do not have to consider different time sources.

6.4.2 Modeling Simultaneity

Simultaneity is an essential property when modeling actions of a computer system. Achieving simultaneity is especially difficult if large time delays are used so that inaccuracies between local timers can become significant. However, simultaneity cannot be modeled explicitly in distributed systems with a dense global time due to the inaccuracies of the local clocks. So simultaneous events must be coordinated globally to achieve nearly simultaneous actions by multiple components. This global coordination step represents an accidental characteristic as it is just required due to technical limitations. It introduces relational aspects into a system that are not required on a purely conceptual level.

Only in a distributed system with a sparse global time-base, simultaneity can be modeled explicitly, without the need for global coordination. The system-wide action lattice can be used to initiate simultaneous functions. This avoids relational aspects in a design, hence improving the decoupling

of components.

6.4.3 Consistency and Order

If a consistent interpretation of time and order is not guaranteed, this leads to ambiguity and non-determinism, which in turn requires additional comprehension effort when trying to analyze all possible system behavior. In addition, functionality to handle these accidental inconsistencies may be required, which leads to increased system size.

With a sparse global time-base it is possible to assign a significant event in the sphere of control of the computer system to the same global clock tick by all components. This characteristic is especially helpful when fault-tolerance is implemented by replication: Replicas just have to reach an agreement on the timestamps of events that are outside the sphere of control of the computer system, but can rely on a consistent interpretation within the system. This considerably helps to reduce the problem space of application development.

Not just if replication at the component level is used, consistency is useful. Also in non-replicated components, a replication of data values and timestamps is often required. For example, in an automotive computer system, multiple DASs may use the wheel speed for different purposes. In these cases, too, a consistent interpretation of time and order within the computer systems helps to reduce the possible behavior of the system and so keeps the number of possible system states low. The amount of agreement and coordination in a system with a global sparse time-base can be reduced significantly, compared to a system where no sparse global time-base is available. Less functionality must be modeled if inconsistencies and non-determinism can be ruled out by design.

6.4.4 Reduced Model of Time

In a system that uses an ε/Δ-sparse time-base, all events that are close together within the interval ε are supposed to happen at about the same time [Kop97]. So this model of time allows to ignore small variations of timing.

For example, if sending and receiving of messages is performed according to a sparse global time-base, small variations of message timing (e.g., caused by different propagation delay of the communication medium) can be ignored. So no small-scale concurrency issues must be considered for all those events that occur according to a sparse time-base.

Another example are small variations of application job runtimes (e.g., caused by varying input data that lead to different execution paths). If a system is designed so that finishing the execution of the job does not represent a significant event for the application, then these differing runtimes can simply be ignored. This can be achieved if just sending and receiving mes-

sages represent significant events and the results of the jobs are transferred to the collaborating jobs according to a sparse time-base.

Furthermore, computations that fit into a single global clock tick can be handled as if they would take no time at all. So for these short computations, runtimes can be fully ignored, except for the fact that it must be ensured they are short enough to fit into the interval.

6.4.5 Availability of a Model of Time

A model of sparse global time is a concept that can be used for system-wide modeling of timing aspects. So it contributes to low relational complexity with regard to timing issues: Timestamps that refer to the sparse global time-base are valid system-wide, so absolute times can be modeled easily. This avoids the establishment of causal relationships that are introduced solely for the progression of control flow. Control flow across interfaces can be avoided in most cases if the progression of global time can trigger system actions. So the global time-base allows for globally coordinated functions with low relational complexity as each component can operate temporally decoupled from the other components – see section 6.4.6. Moreover, time-outs can be avoided, as durations can be specified by referencing two instants of the global time.

In a system where there is no notion of a global time-base, it is not possible to generate globally valid timestamps for events. So time can only be referenced according to local clocks. Regarding the interactions of components, timing issues usually have to be modeled relative to events, which results in the introduction of causal event chains resulting in high relational complexity (see section 6.7.2). The causal event chains must be followed to be able to understand the system behavior as absolute points in time cannot be referenced. For example, to analyze the timing and the input parameter values of a function call, its causal predecessors that can call the function must be determined. Such causal chains can be very long, making a full analysis based on human understanding unfeasible. So formal analysis techniques and tools support may be required.

Another problem in systems with no global time-base is that timeouts must be used to specify durations. However, the introduction of timeouts can add relational aspect to a system. Multiple simultaneous timers conceptually is like having multiple clocks. As already mentioned in section 3.2.2, parallel aspects that are not completely independent of each other tend to be hard to understand. If the effect of a timeout is not fully constrained to an isolated aspect, relational aspects are introduced. So in general, timeouts should be used with care.

6.4.6 Temporal Decoupling

The availability of a sparse global time-base supports the temporal decoupling of components. If a component can become activated according to the progression of real time, this means that a component can implement its own control strategy. Temporal decoupling does not mean temporal independence. Temporal decoupling just means that control flow is not necessary to propagate data changes across interfaces. The interfacing partners have some freedom to access shared data items. The global timing properties, such as the temporal accuracy of a real-time image, must be covered by the interface specification. As long as the temporal constraints of the interface specification are fulfilled, a component is free to decide on the points in time when data is read or written.

If components are temporally decoupled from each other, e.g., by the usage of temporal firewall interfaces, this means that no control flow and no temporal dependencies between components must be considered. Due to the autonomous control, each component can be considered a nearly-independent system with no relational properties resulting from control flow propagation (see section 6.5.3).

6.5 System Structure

In this section the implications of the DECOS system structure with regard to comprehensibility are discussed. First, the structure of a single DAS and then the integration of multiple DASs is considered.

6.5.1 Simple Application Structure

Jobs are the only units of structuring an application. This leads to a very regular structure as there is just a single granularity of structural elements: All jobs represent the same abstraction level. Hence, no climbing up and down the abstraction ladder (see section 5.6.2) is necessary when modeling the job behavior (i.e., when designing the job interfaces), or when integrating the jobs of a DAS.

Moreover, no structural relations, such as containment or inheritance between the jobs must be considered. This means that at the purely structural level the relational complexity is just minimal. The only dependencies that can be introduced are functional ones – by exchanging messages.

The definition of time- or event-triggered ports is the only mechanism of the architecture to specify interactions between components. This represents just a very minimal – hence easy to understand – set of concepts that are required to understand the interactions between components.

Due to the fact that all communication is redirected through ports, this makes all interactions between the jobs explicit. There are no implicit inter-

faces. Hidden dependencies or unintended side-effects that are not visible at the job interfaces can not occur. This makes it possible to analyze a job at the interface level, without having to consider its implementation.

For a safety-critical DAS where just state message ports are used, a high level of decoupling between the jobs is achieved so that they can represent nearly independent components at the application level. The job interfaces serve as conceptual segmentation boundaries and provide conceptual chunking of jobs. So when reasoning about a job, the job interface specification provides a sufficient representation of both the job and the environment – no thinking beyond the interface is necessary.

Of course, fully specified interfaces are only possible if the job interface specifications are at the level of abstraction of the application. If the ports are just used to tunnel high-level protocols that require additional knowledge that is not contained in the specification, this makes understanding of interactions and dependencies more complex again – see section 5.3.8. For safety-critical DASs such mechanisms are not allowed. For the non safety-critical DASs it is the responsibility of the designers to avoid overly complex solutions. An architecture can just provide easy-to-understand mechanisms, but not avoid their circumvention by application designers. So for safety-critical DASs the job interfaces must be fully specified in the time and value domains and also provide a sufficient semantic specification. For non safety-critical DASs this requirement can be relaxed as the errors that can be introduced due to the additional design freedom do not have severe consequences. So opaque data structures can be transmitted where the possible contents are not fully specified at the port level.

The communication mechanisms that are provided directly by the architecture are at a very low level of abstraction. High-level protocols must be implemented at the DAS-level, or tailored to a DAS as a high-level architectural service.

As discussed above, this is possible for event-triggered applications just as on any other computing platform, by tunneling a high-level event-triggered protocol via the event message ports. For state message based communication of safety-critical DASs, it is not so easy to increase the level of abstraction of the communication. However, due to the nature of safety-critical applications, and due to the fact that the transport of state information is inherent to most control applications this is not a severe problem. In some cases the introduction of state message objects (at the application level) as suggested in section 6.2.5 can be beneficial, though.

6.5.2 Straightforward DAS Integration

The encapsulation service and the virtual network service provide a framework so that a DAS can be completely isolated from other DAS. Each DAS is logically fully independent from the other DASs that are executed on the same system. This way the advantages of a federated system are fully

preserved by the shift to an integrated system: A priori independent applications logically remain fully independent from the other systems that share the same physical hardware. No side-effects, dependencies or coupling is introduced due to the integration. The only form of coupling that is possible are explicitly modeled (i.e., intended) interactions via virtual gateways to achieve service integration.

The selective redirection of data via virtual gateways is defined at the beginning of the development process of each DAS. This allows for independent development of the DASs by separate vendors and ensures matching interfaces for seamless integration at the end of the development process. So no design tasks are left for the integration process, such as introducing glue functionality to resolve interface mismatches.

The selective forwarding of information from one DAS into another DAS with a virtual gateway has considerable advantages regarding complexity management: First, the selective data forwarding makes the real interactions explicit as just the information that is really needed in the other DASs is passed on. Second, the reduction of the amount of information to just the relevant interactions reduces the cognitive load when analyzing the DASs as the problem space becomes smaller. Third, a gateway supports the introduction of additional abstractions, instead of simply forwarding information from one DAS into another. This can rise the level of abstraction of the interactions and reduce the relational aspects – see section 5.3.2. Fourth, the gateway approach supports the conceptual decoupling of a DAS, because for understanding a DAS, only the jobs of the DAS with their interactions and the specification of the virtual gateways must be considered.

An important characteristic of the DECOS architecture is the possibility to tailor virtual networks according to the needs of the application (service optimization). So there is no need to use an architectural framework that represents a compromise which does not fulfill all application requirements fully. This is very likely to happen in an architecture where the communication systems of each subsystem are not fully independent of each other and where tailoring of the communication services on the DAS level is not possible. Moreover, service optimization allows to provide the communication at the optimal level of abstraction for each DAS.

6.5.3 Near-Independence of Components

Near-independence of subsystems is a design principle of time-triggered systems that is achieved in the DECOS architecture by the combination of the following design concepts:

Encapsulation Service: An encapsulation of components avoids unintended interference between unrelated functionality.

Temporal Firewalls: Autonomous control is supported by the introduction of temporal firewalls as linking interfaces between components.

Sparse Global Time: A sparse global time-base enables the establishment of a globally consistent view on the temporal characteristics of real-time images.

Temporal firewalls together with a sparse global time-base enable the autonomous control of components. Autonomous control means that control flow is restricted to the component level. No control flow across component interfaces is allowed. This is a fundamental requirement to achieve high degree of temporal decoupling between components, but still allows controlled data exchange. As long as the shared data is temporally accurate, each component can access the data autonomously. The sparse global time-base enables a common (i.e., consistent) view on the temporal properties of the temporal firewall.

The DECOS architecture provides an encapsulation service for DASs so that different applications can run completely independent of each other. At the network level, the encapsulation between the communication of different DASs is provided by the virtual network service. So each DAS can be considered a nearly-independent component of the overall computer system.

Within a time-triggered DAS, encapsulation is also possible, but it cannot be as strict as between different DASs as the jobs of a DAS must collaborate. Closely related functionality that needs a high degree of coupling can be clustered into jobs. So the coupling between different jobs can be minimized at the functional level. In addition, jobs can represent units of error containment within a DAS, e.g., if replicated jobs are used. So erroneous information from a faulty job cannot propagate to other jobs. Due to the encapsulation and the high degree of decoupling, jobs can also be considered nearly-independent components of a DAS. The degree of independence of jobs is of course lower than that of different DASs.

Due to the tight control flow coupling, the jobs of an event-triggered DAS do not represent nearly-independent components.

6.6 Development Process

In this section the characteristics of the DECOS development process are described. First, the advantages of the two-level application design approach are presented. Then, model-based application development and hardware platform modeling are discussed.

6.6.1 Two-Level Application Design

Regarding application design, the DECOS architecture supports a two-level design: At the global level[1] the interfaces between the jobs of a DAS and the virtual gateways to other DASs are specified. At the job-local level the

[1] Global here means the whole DAS, not the whole system

constraints of the interface specification enable the isolated development of the jobs.

A two-level design approach strictly separates between the interface design of the jobs and the design at the job level. The behavior of the jobs is already fully described by the interface specifications, so the design at the job level is only concerned with fulfilling the requirements and constraints that were established by the interaction design.

Due to the explicit interaction design, no hidden dependencies or implicit interfaces are created. The interfaces that are developed during the global design step represent stable building blocks (i.e., stable intermediate forms) for job development.

An advantage of a strict two-level design approach is that a clear separation of concerns is possible: Both global aspects and job-local aspects are designed explicitly and can also be verified at a separate level.

At the global level the designers can fully concentrate on the interactions between the jobs at a level where the job internals are not relevant. So the interaction design of an application can be made in an ideal environment, where just the application model itself can be focused. At that level a conceptual interface model for each job can be developed. As the designers can abstract away component internals they are not distracted by the component implementation aspects.

The separation into the global design and the job-local design does not mean that there are just two steps that are always performed just once, one after the other. Job implementation may lead to new insights and a better understanding of the problem. So it can of course happen that during the design of a job a developer recognizes that a certain global constraint is unfeasible. For example that an update rate of a message can not be achieved, or that an additional real-time image that is available only in another job is necessary. So the global design must again be changed to accommodate these requirements. However, in a two-level design, an explicit change of the global design is necessary. So the developers immediately recognize that their changes have a global effect. If there is no clear separation between job-local and global design decisions, job-local details can more easily propagate to the system level without being recognized.

6.6.2 Model-Based Application Design

Creating an application means to develop a (usually high-level) conceptual model and then derive a (usually more detailed) technical model that can be used for implementation. This model building is pure design work that must be done by the developers. So at this stage of development, human understanding is especially important. This is where model-based design can be especially helpful, as it allows to increase the level of abstraction of the applications.

The generic architectural services provide a *platform model*, which is an abstraction of the underlying hardware platform. This platform abstraction provides a simplified design environment, i.e., a reduction of the problem space. The applications can be developed independently from details of the underlying implementation. Due to this separation from the underlying platform, which results in pure high-level application functionality, the platform-independent model (PIM) of an application is easy to understand and reuse.

Modeling of the application at a high level of abstraction, i.e., at the PIM level enables the creation of a mental model of the application without being influenced (distracted, overloaded) by implementation details. Moreover, developers are not tempted to start with the implementation (i.e., modeling at the implementation level) before they have developed a complete model (i.e., before they have fully understood what they have to do). So the application modeling process is simplified by the high level of abstraction.

The PIM (a technical model) can serve as a basis for a high-level mental model of an application. It is much harder to derive a high-level mental model from an implementation-level technical model, such as a PSM, or – even worse – from application code. For example, the DECOS architectural membership service can be described more easily at the PIM level than on the implementation level. For the application at the PIM level it is irrelevant how a membership vector is created. Its existence and its semantics are all that is relevant.

Another advantage of the PIM is that it is available for later analysis of the system, e.g., for certification or maintenance tasks. So a mental model of the application can be derived more easily as a suitable high-level technical model is available. If there is no explicit high-level model, then the implementation is the only technical model that is available for analysis. In that case the extraction of a high-level mental model, which is required for comprehension, is far more effort.

The virtual integration enables the detection of design faults early in the design process, before the DASs are deployed on real hardware. So a purely functional integration is possible, where problems caused by hardware or compilers cannot occur. At this stage of development, simulations can already show the behavior of the future application. So the integration process is reduced to a virtual resource allocation problem. Physical hardware problems, such cabling or connector faults, that typically cause problems in the early stages of development of embedded systems, cannot occur. As every embedded system developer knows, it is often hard to decide whether a system failure is caused by a hardware fault or by a software fault, especially if the software has not yet run successfully. So the developers can get some confidence in their applications before the deployment to hardware.

6.6.3 Hardware Platform Modeling

As described in section 6.1.5 the DECOS modeling process for the hardware platform provides a library with resource primitives. So a compositional definition of node computers and complete clusters is possible. The designers need not start from scratch but can build on predefined elements that just need to be correctly composed to create the required resource specification. The predefined resource primitives and hardware elements (including full definitions of node computers) provide a standardized framework for conceptually guided application development.

The provision of pre-defined building blocks, such as the pre-defined hardware elements or the architectural meta models, has the advantage that these structures can be created by experts. So these building blocks can be reused by system developers that may not have such a high level of expertise in hardware design. This can improve the quality of the system as the results produced by experts usually excel those of less experienced people (see section 2.4). For example, many design errors can be avoided by the specification of meta models and constraints against which the models of the developers can be checked. Moreover, the difficult parts of the resource specification can be delegated to experts, such as hardware suppliers, and then be included into the system model by the system integrator. E.g., a sensor can be modeled by an expert with extensive knowledge in this domain [HOP06].

Semantic descriptions are essential to establish a common understanding of the hardware building blocks among different people involved in the development. The technical dictionary links and the natural language descriptions support a sufficient specification of semantics.

Due to the provision of automatic checkers, the mental load of verifying the constraints, which mainly represent relational properties of modeling objects, is not burdened onto the developers. In addition, external cognition is supported by the modeling tool and the provision of a meta model. Hence, designers can experiment easily with different configurations and settings, without having to consider all possible implications of their changes.

6.7 Time- vs. Event-Triggered Control

In the DECOS architecture both the time-triggered as well as the event-triggered control paradigm are supported. As already explained in section 4.9, the actions of an event-triggered system are in fact reactions that are triggered by events, whereas in a time-triggered system the progression of real time triggers the actions of the system. This is the major difference between the two control paradigms that also has considerable influence on the conceptual structuring and thus on the analysis of the behavior of a DAS.

At a first glance one might think that not every problem is suitable for a time-triggered solution, which has a cyclically recurring behavior, and that the time-triggered model of thinking is not "natural" as we are more used to think in cause-effect relationships. This may be true to some extent, however, it is not the whole story. In this section the major differences between the two paradigms that affect comprehension are discussed. We will see that it depends on the problem and on the application characteristics whether a time-triggered or an event-triggered approach is likely to be more complex.

6.7.1 Conceptual Structuring

In a time-triggered DAS, significant events that are in the sphere of control of the computer system are triggered only by the progression of real time. The instants when these events occur are determined a priori. So these points in time represent important information that can be used to get an understanding of the behavior of the system. The trigger times represent the focal points for conceptual structuring. So the time axis represents the basis for modeling and understanding the behavior of a system – every action of the system is related directly to an instant of the time-line, see figure 6.6. At each a priori defined instant a_i, b_i, c_i, and d_i a given application job A, B, C, and D is executed.

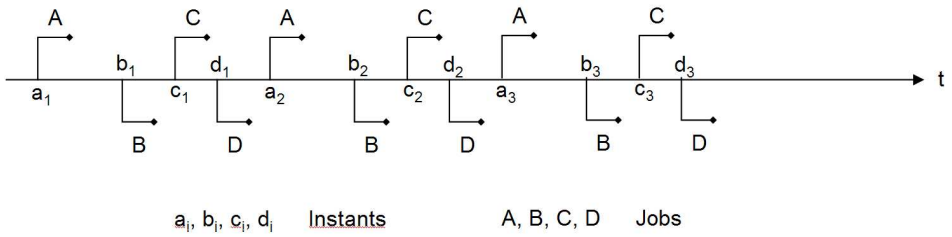

Figure 6.6: Job activations triggered by the progression of time

In an event-triggered DAS it is the occurrence of events that provides a basis for conceptual structuring, not the time-line. The time of occurrence of events is not known a priori and may vary considerably, depending on the dynamics of the environment. So no information about the order of events and about the order of possible reactions of the system can be taken as given. Hence, a structuring of the actions of the system along the time axis is not possible. Instead, causal event chains must be followed to be able to determine the initial cause of a reaction, or the possible effects of an event – see section 6.7.2.

As already mentioned in section 2.1.2, there exists evidence that temporal problems are mentally represented like spatial problems. Hence, the time-line based action structure of the time-triggered approach seems to be advantageous compared to the event-triggered approach. The event-based structure cannot be represented easily as a spatial diagram.

6.7.2 Modeling Cause and Effect

Cause and effect is a well-known relation that can frequently be found in the real world. So we can easily model a system according to this principle. When an event-triggered DAS is designed, cause and effect relationships are usually modeled via event handlers. A developer can define appropriate handling functions for each significant event. The event handlers can in turn generate new events that cause the invocation of further event handlers, resulting in event chains. So causal event chains can be a very helpful structuring technique. As long as the event chains remain short and the event handlers just have low relational aspects to other handlers, the system behavior can be understood easily. For each event it is just necessary to follow its event handlers. However, if the chains become long, include branching or cyclic structures, the relational aspects between the event handlers increase. This leads to high relational complexity when analyzing the possible effects of an event, or when trying to determine the cause of a system reaction. The possible length of the causal event chains depends on the system structure and on the degree of coupling between the components. If data value changes can trigger remote events and control flow across interfaces is possible, the whole system may be affected by a single event. So a high degree of coupling between components is typical for event-triggered systems.

To keep causal event chains short, boundaries must be provided that break up the chains and avoid the propagation of events all over a system. However, the developers must explicitly design their systems in a way that avoids long event chains. Moreover, causal relations may sometimes be hidden within an application so that they are not immediately recognized. So it may be very hard to ensure short causal event chains in a large event-triggered application.

In a time-triggered DAS, cause and effect is just modeled at a very small scope – only within a single job. For example, when an incoming message is read from a time-triggered state message port, specific functionality may be executed, depending on the message data. There are no long causal event chains as the application jobs of a DAS are usually small units that communicate with one another via temporal firewalls. The propagation of information about an event from one job to the next is done via state messages. So an important advantage of time-triggered DASs is that there is no global control flow across job interfaces. The temporal firewalls break up the global control flow, keeping the causal event chains very short. This considerably supports comprehension as long and net-like causal relationships tend to be very hard to understand (see section 3.2.4).

If there are no long causal event chains between a significant event and an action of the system, the step backward to the triggering event is very short. The reactions are always immediate. Long delays caused by event queuing or long causal event chains do not occur. This supports understanding the system behavior in simulation environments or when analyzing the run-time behavior of a system.

It can be considered a disadvantage of the time-triggered approach that a direct modeling of causal relationships across component interfaces is not possible. This means that creating such relationships requires more effort than in an event-triggered system as the additional intermediate mechanism of a time-triggered state message is required. This involves the introduction of the concept of a regular message transfer. However, this characteristic can only be considered a disadvantage for very small systems. As discussed above, the advantage of short causal event chains for improved understanding outweighs the additional design effort in most large systems that would otherwise require long and net-like causal event chains.

6.7.3 Segmentation Boundaries

In an event-triggered DAS the job interfaces do not coincide with the boundaries of control flow. Control flow across component interfaces is is an inherent principle of event-driven control. This coupling via the control flow leads to a high conceptual coupling between the jobs, so "thinking beyond the job interface" is required to analyzing cause and effect relationships. Thus, no built-in segmentation along the job interfaces is provided.

In a purely time-triggered DAS, time-triggered state messages are used as a means for the transport of information across component interfaces. As described in section 6.2.3, these state messages represent an abstraction provided at the component interface that improves the temporal and conceptual decoupling of the components. So conceptual segmentation boundaries are provided by the component interfaces and each component can be analyzed and implemented almost independently from the rest of the system.

6.7.4 Behavioral Variety

Time-triggered DASs have a repetitive nature. A job is activated according to an a priori created scheduling table, reads the same input ports, performs its calculations, and then writes its set of output messages to the output ports – usually at about equidistant points in time. This static, globally synchronized action lattice results in a high degree of behavioral regularity. A component does not change its temporal behavior, and similar functionality is performed each time the job is activated. So the variations in behavior are restricted to the value domain and do not affect the temporal properties of the jobs. Thus, just a limited number of comprehension strategies is needed, compared to a system with varying temporal behavior.

In an event-triggered DAS, actions are triggered by events within the system or by events in the environment. This results in inherent dependencies of the component behavior on the dynamics of the event occurrences and on the characteristics of the input data[2]. The jobs are typically called in different orders. Due to this increased behavioral variety of event-triggered

[2]The input data may trigger new events.

DASs, a larger number of comprehension strategies is required than for a comparable time-triggered DAS.

6.7.5 Size of the Problem Space

In an event-triggered DAS the increased behavioral variety results in a larger problem space for system development. The unpredictability of the dynamics of the environment are directly reflected by the system behavior, which results in arbitrary sequences of job activations. For example, first job A may be activated before job B, and the next time the jobs may be executed in the reverse order. This increased variability must be considered when designing a DAS. Mechanisms may be required that make the jobs independent of the order and frequency of execution. So the non-determinism of the environment must be handled inside the system.

In a time-triggered DAS the order of job execution results from the message dependencies between the jobs. For example, if job A sends a message to job B then job A is executed before job B. The static job schedule ensures that the job sequence never changes at run-time. So resource conflicts and concurrency problems are avoided by design. Moreover, in a time-triggered DAS the relevant real-time entities in the environment are polled regularly for their state. So the dynamics of the environment do not influence the temporal behavior of the system. This reduced system behavior results in a considerably reduced problem space for system development. The unpredictability and non-determinism of the environment is not handled inside the system, but abstracted away at the interfaces.

6.7.6 Hybrid Applications

As explained in the previous subsections, the time-triggered and event-triggered approach of modeling differ considerably. We need to think in different ways about an application, depending on whether it follows the time-triggered or the event-triggered paradigm. This means that mixing up the two approaches in a single system requires a combination of both ways of thinking. If aspects of both paradigms are intertwined in the same application, or if mutual influences between time-triggered and event-triggered application systems cannot be ruled out, this makes system development and analysis very hard. Moreover, many advantages of the time-triggered approach, such as a high degree of conceptual decoupling between components, do not exist in hybrid systems.

In a hybrid integrated system that supports time-triggered as well as event-triggered control, it is thus essential to clearly separate the time-triggered from the event-triggered functionality. In the DECOS architecture, this separation is supported as a control strategy can be established per DAS, and mutual interferences between different DASs are ruled out by the encapsulation service. Moreover, for safety-critical (time-triggered)

DASs, no dependencies on messages from non safety-critical DASs are allowed.

6.8 Chapter Summary

Time-triggered systems in general and the time-triggered DASs of the DECOS integrated architecture in particular provide various mechanisms that support system comprehension.

State messages are used as the only means for communication between application jobs. They are self-contained and enable the development of components with low relational complexity. The dynamics of the environment are abstracted by periodic state message readings, enabling a stable and simplified view of the relevant real-time entities.

Temporal decoupling between time-triggered components is achieved by the provision of a sparse global time-base, together with temporal firewalls that prohibit control flow across the interfaces. So the components on either side of the interface can operate according to their individual schedules.

For event-triggered DASs, control flow across job interfaces is typical, which results in long and often net-like causal event chains. Oversimplification of the event chains and ignoring relevant side-effects is more likely when analyzing event-triggered than time-triggered DASs.

As the conceptual structuring of time-triggered and event-triggered DASs differs considerably, the clear separation of event-triggered and time-triggered functionality at the DAS level is essential to ensure that the advantages of the time-triggered approach regarding comprehensibility are maintained.

The encapsulation service ensures that no unintended interactions between different jobs can occur. The restriction of all interactions between jobs to message exchange via well-defined ports makes all interactions explicit and avoids implicit interfaces. So hidden dependencies are avoided by design.

Temporal decoupling, a full specification of all input and output ports, and the provision of the encapsulation service are the key factors for a high degree of conceptual decoupling between the jobs of a time-triggered DAS. Time-triggered DASs provide built-in conceptual segmentation at the job interfaces.

The conceptual decoupling of time-triggered components allows to reduce the scope of consideration during component development to the component itself. In the DECOS architecture, each DAS can be considered a nearly-independent subsystem of the integrated system. Similarly, each job can be considered a nearly-independent subsystem of a time-triggered DAS. Of course, the degree of decoupling between the jobs of a DAS is lower than that between different DASs.

The provision of the port concept in just two basic forms – with time-triggered and event-triggered control – reduces the number of architectural communication mechanisms to a minimum. This reduces the amount of learning effort and also the cognitive effort when analyzing an system.

The logical system structure does not contain deep hierarchical relationships or dependencies. The system is structured vertically into a safety-critical and a non-safety-critical subsystem and – at the same level of abstraction – into various DASs. The only hierarchical relationship is that various DASs are further subdivided into jobs. However, at the job level this hierarchical subdivision is not visible any more due to the introduction of virtual gateways. So for any application development task, there is just a single level of abstraction regarding structural units.

The provision of a dedicated virtual networks for each DAS allows for service optimization and supports an optimal level of abstraction of the communication services for each DAS.

The architecture provides various architectural services that serve as a validated and stable baseline that reduce the problem space for application development and enable a high degree of structural regularity. The architectural services provide a simplified environment for the applications as they support the transparent masking of various classes of failures and enable determinism at the application level. So coherent applications that are easy to understand and reuse can be developed. Furthermore, a clean separation of concerns between application functionality and generic architectural services is ensured.

The support for composability ensures stable application models regarding the integration process. The architectural fault-tolerance and encapsulation services ensure application model stability even in case of faults.

The sparse global time-base provides an abstraction of real time that more closely resembles our natural understanding of time than the use of various local clocks that are not perfectly synchronized. Moreover, a sparse global time allows to model simultaneity and supports determinism. It also represents a reduced model of time that avoids small-scale concurrency problems. The availability of a model of time simplifies the implementation of globally coordinated functionality and avoids the introduction of timeouts and similar mechanisms. The globally synchronized, system-wide action lattice induced by the sparse time-base provides behavioral regularity of the system.

The DECOS architecture follows a strictly model-based approach which supports a high level of abstraction for the modeling of applications in the PIM, abstracting from the details of the underlying platform. The two-level design methodology ensures composability and enables the independent development of jobs or whole DASs by different suppliers. So the job interfaces provide stable building blocks for component-based application development.

Model-based development is also supported for the hardware platform where pre-defined building blocks created by experts are provided.

The models can be verified according to architectural meta models and pre-defined constraints. So design errors can be detected during the modeling process. The PIM is transferred to a PSM with tools support to relieve the developers from resource allocation and schedule generation tasks.

Chapter 7

Conclusions

This section first summarizes the most important aspects of the dissertation, then outlines the scientific contribution and finally describes open issues that need further research.

7.1 Summary

Managing the complexity of computer system design has always been subject to research in computer science. The increase of the abstraction level from the early programming languages to today's modeling environments has alleviated some of those problems. However, temporal composability and independent development of components is not possible by only increasing the level of abstraction. A major problem are the inherent relationships between components. Just as almost no single concept can be fully described in isolation, the same problem occurs during system development. When we develop a single component of a distributed application, we always need some knowledge about the other components in the system.

The only promising strategy to avoid complex development tasks is the decomposition of a large system into nearly-independent components that can be developed almost independently. The components must be composable so that their integration into the system can proceed with minimal effort.

In this thesis self-contained system-level components are suggested as a means to provide nearly-independent components that can be integrated with just minimal effort. A two-level design approach enables the design of fully specified component interfaces where mismatches can not occur. All application functionality is implemented by the system-level components, so no design effort is left at the system level. A composable architecture with a two-level design methodology ensures that the properties of the components also hold when the components are integrated. So behavioral stability of the components is guaranteed and uncontrolled emergent properties can be avoided. In addition, a single level of abstraction of components reduces the

complexity of the modeling and integration tasks.

Full conceptual independence between components is not possible due to the inherent relationships between collaborating components. However, conceptual decoupling is the key for independent development of components. If there are lots of conceptual interdependencies between components, comprehensibility is impaired significantly as it is not possible to consider a component independently of other components. The characteristics and the placement of the interfaces decides on the degree of conceptual decoupling that can be achieved. State message interfaces serving as temporal firewalls support a high degree of conceptual decoupling as state messages are self-contained and can fully describe a real-time entity. In addition, temporal decoupling is enabled as temporal firewalls avoid control flow across interfaces. The provision of a sparse global time-base that can trigger component functions supports the temporal decoupling between components as the temporal control can remain component-local. This means that the control flow is broken up at the component interfaces, which avoids long causal event chains. Such event chains are a major factor for relevant relations between components. So state message interfaces allow for simple system structuring by minimizing relational component properties, hence limiting the scope of consideration for component development to a single component.

A globally available notion of time most closely resembles our natural understanding of the concept of time, which is that of an omniscient observer. It avoids the introduction of the concept of multiple parallel local clocks that are not perfectly synchronized. A sparse global time-base further simplifies system development as globally coordinated actions of the system and simultaneity can be modeled easily. In addition, a consistent interpretation of time and order is guaranteed for all events that are in the sphere of control of the computer system. Furthermore, the abstraction of a sparse time-base represents a reduced model of time where small variations in timing can be ignored.

Segmentation at the system level means to split the large system into smaller conceptual units. If these conceptual units correspond to the technical units (components) that build up the system this structure can be seen as built-in segmentation and chunking: The border lines between the components, i.e., the interfaces, correspond to the conceptual segmentation lines; chunking is provided by the abstraction of the interfaces. So built-in segmentation and chunking supports the conceptual decoupling of the components by minimizing relational properties. Moreover, the conceptual structure is easy to identify, which supports external cognition: The conceptual units that are needed for understanding are also reflected by the system structure and do not just exist as mental representations. The component-oriented approach presented in this thesis supports built-in conceptual chunking and segmentation at the component interfaces. All interactions between components are redirected through ports, which makes them explicit, hence avoiding implicit interfaces.

The provision of systematic fault-tolerance can help to reduce the complexity of various development tasks. Application-specific fault-tolerance can be minimized, so the design problem is simplified considerably. Moreover, architectural fault-tolerance enables a separation of concerns: Applications contain only application functionality, whereas the fault-tolerance functions are provided by the architecture. So conceptually coherent applications can be developed. In addition, the architectural fault-tolerance leads to increased structural regularities that support comprehension.

An architecture that supports composability enables the use of stable models across various development phases. This is a considerable advantage compared to an architecture that does not support composability, where different models of a component must be used, depending on whether a component is considered in isolation or as an integrated part of the system.

In time-triggered systems, the instants when actions are triggered represent the focal points for conceptual structuring of system behavior. This keeps causal event chains short. In event-triggered systems it is the occurrence of events that provides a basis for conceptual structuring, not the time-line. Causal event chains resulting from interconnected event handlers are typical for event-triggered systems. Long and net-like causal event chains result in complex causality and high relational complexity when analyzing the possible effects of an event, or when determining the causes of a system reaction.

In addition, the behavioral regularity of time-triggered systems, which results from the cyclic operation, requires just a low number of comprehension strategies. Event-triggered systems exhibit far more behavioral variety and unpredictability which increases the problem space for system analysis considerably.

Determinism enables deductive reasoning, which is essential for the rational analysis of behavior. Hence, computer system architectures must support the development of highly deterministic applications. The DECOS integrated architecture supports determinism at the application level, induced by the sparse global time-base, architectural fault-tolerance, deterministic and timely message transport, the encapsulation service, strong fault isolation, and by the consistent diagnosis of failing nodes.

7.2 Contributions

The introduction of hierarchical relationships and abstraction is the usual approach to reduce the complexity of development tasks. This is the concept of divide and conquer, which is not new, of course. However, the traditional approaches such as structured or object-oriented development do not provide a high level of conceptual decoupling between the subsystems. So independent development of components is not possible as component developers always need some knowledge about the other components of the system.

The approach presented in this thesis is more than just traditional divide and conquer, though. The component model supports a high degree of conceptual decoupling of components so that the component interface specification is sufficient for all reasoning about the component. A theoretical basis for conceptually dividing a large system with many interdependencies into decoupled nearly-independent components with well-defined interfaces is introduced. The characteristics of component interfaces are analyzed in detail with regard to comprehensibility issues, resulting in a guideline for component interface design. In addition, various techniques for architectural complexity management are presented, accompanied by cognitive support theories.

In the computer science literature, interdisciplinary approaches to complexity management can hardly be found, and to the knowledge of the author there exist none in the area of embedded real-time systems. So the introduction of complexity management techniques for embedded real-time systems with regard to psychological theories represents a unique contribution of this work, especially the analysis of conceptual chunking and segmentation support by interfaces.

In addition, architectural real-time systems concepts are analyzed with regard to their support for complexity management. The effects of the architectural concepts on comprehensibility have never been fully analyzed, so lots of knowledge was left implicit. The cognitive support theories provided by this work make the advantages of specific architectural concepts explicit. So a theoretical framework is available that justifies the claims that some of the architectural concepts can really support comprehension.

7.3 Open Issues

To be able to make statements about the complexity of more specific development and maintenance tasks than for general system comprehension, cognitive process models of the tasks must be developed. However, it is yet unclear if it will ever be possible to determine exact cognitive process models that are generally applicable. As human reasoning heavily depends on the knowledge base in long-term memory, which can be different for every single person, the cognitive processes may vary considerably. For example, an experienced developer may use a more efficient chunking and segmentation strategy for a given problem than a novice.

Furthermore, empirical evaluations of the predictions made in this work will have to be done. However, this is beyond the scope of this thesis as large-scale experimental evaluations with real-world applications are necessary. Most problems that can be alleviated with the complexity management techniques proposed in this work can only be observed in large real-world systems. The extraction of small-scale experiments with the same characteristics seems unfeasible.

Bibliography

[Aer91] Aeronautical Radio, Inc., 2551 Riva Road, Annapolis, Maryland 21401. *ARINC Specification 651: Design Guide for Integrated Modular Avionics*, November 1991.

[AGWA04] Tracy Packiam Alloway, Susan E. Gathercole, Catherine Willis, and Anne-Marie Adams. A structural analysis of working memory and related cognitive skills in young children. *Journal of Experimental Child Psychology*, 87:85–106, 2004.

[AIS77] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language*. Oxford University Press, 1977.

[AKC90] Mohan Ahuja, Ajay D. Kshemkalyani, and Timothy Carlson. A basic unit of computation in distributed systems. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 12–19, Paris, France, 1990.

[Ale64] Christopher Alexander. *Notes on the Synthesis of Form*. Harvard University Press, 1964.

[AUT06] AUTOSAR GbR. *AUTOSAR – Technical Overview V2.0.1*, June 2006.

[Avi82] Algirdas Avizienis. The four-universe information system model for the study of fault-tolerance. In *Proceedings of the 12th Annual International Symposium on Fault-Tolerant Computing (FTCS-12)*, pages 6–13, Santa Monica, California, June 1982. IEEE.

[AW95] Christopher Ahlberg and Erik Wistrand. IVEE: An information visualization & exploration environment. In *Proceedings of the 1995 IEEE Symposium on Information Visualization*, pages 66–73, Atlanta, Georgia, USA, 1995. IEEE Computer Society Press.

[Bad98] Alan Baddeley. Recent developments in working memory. *Current Opinion in Neurobiology*, 8:234–238, 1998.

[Bad00] Alan Baddeley. The episodic buffer: a new component of working memory? *Trends in Cognitive Sciences*, 4(11):417–423, November 2000.

[Bar32] Frederick Charles Bartlett. *Remembering: A study in experimental and social psychology.* Cambridge University Press, London, 1932.

[Bar87] Howard Barringer. Up and down the temporal way. *The Computer Journal*, 30(2):134–148, 1987.

[BB87] Ronald M. Baecker and William A. S. Buxton, editors. *Readings in Human-Computer Interaction: An Interdisciplinary Approach.* Morgan-Kaufmann Publishers, Los Altos, CA, 1987.

[BBH+95] Cornelia Boldyreff, Elizabeth L. Burd, R. M. Hather, Richard E. Mortimer, Malcolm Munro, and E. J. Younger. The ames approach to application understanding: a case study. In *Proceedings of the International Conference on Software Maintenance*, pages 182–191. IEEE Computer Society Press, 1995.

[BCC+03] Ed Brinksma, Geoff Coulson, Ivica Crnkovic, Andy Evans, Sebastien Gerard, Susanne Graf, Holger Hermanns, Bengt Jonsson, Anders Ravn, Philippe Schnoebelen, Francois Terrier, Angelika Votintseva, and Jean-Marc Jezequel. Component-based design and integration platforms. Technical report, Verimag, Grenoble, France, 2003.

[BDM93] Michael Barborak, Anton Dahbura, and Miroslav Malek. The consenus problem in fault-tolerant computing. *ACM Computing Srurveys (CSUR)*, 25(2):171–220, 1993.

[Ber81] Jacques Bertin. *Graphics and Graphic Information Processing.* Walter De Gruyter, Berlin, New York, 1981.

[Ber83] Jacques Bertin. *The Semiology of Graphics (english translation).* The University of Wisconsin Press, 1983.

[BEW95] Richard A. Becker, Stephen G. Eick, and Allan R. Wilks. Visualizing network data. *IEEE Transactions on Visualization and Computer Graphics*, 1(1):16–28, March 1995.

[BGA56] Jerome Bruner, Jacqueline Goodnow, and George Austin. *A study of thinking.* Wiley, New York, 1956.

[BH74] Alan Baddeley and Graeme Hitch. Working memory. In Bower G.A., editor, *The Psychology of Learning and Motivation*, volume 8, pages 47–89. Academic Press, New York, 1974.

[Bir02] Damian Patrick Birney. *The Measurement of Task Complexity and Cognitive Ability: Relational Complexity in Adult Reasoning.* PhD thesis, School of Psychology, University of Queensland, St. Lucia, Queensland, Australia, March 2002.

[BK01a] Sarita Bassil and Rudolf K. Keller. A qualitative and quantitative evaluation of software visualization tools. In *Proceedings of the Workshop on Software Visualization*, pages 33–37, Toronto, Ontario, May 2001.

[BK01b] Sarita Bassil and Rudolf K. Keller. Software visualization tools: Survey and analysis. In *Ninth International Workshop on Program Comprehension (IWPC'01)*, pages 7–17. IEEE Computer Society Press, 2001.

[BL76] Laszlo A. Belady and Meir M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976.

[BMW93] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas Webster. The concept assignment problem in program understanding. In *Proceedings of the 15th international conference on Software Engineering*, pages 482–498, Baltimore, Maryland, United States, 1993. IEEE Computer Society Press.

[Bro58] Donald Eric Broadbent. *Perception and communication*. Pergamon, London, 1958.

[Bro83] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18:543–554, 1983.

[Bro87] Frederick P. Brooks. No silver bullet: essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

[BS03] Benjamin B. Bederson and Ben Shneiderman, editors. *The Craft of Information Visualization. Readings and Reflections.* The Morgan Kaufmann Series in Interactive Technologies. Morgan Kaufmann Publishers, San Francisco, CA, 2003.

[CDK94] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, 2 edition, 1994.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[Chi00] Michelene T. H. Chi. Misunderstanding emergent processes as causal. Presented at the American Educational Research Association annual conference, New Orleans, April 24–28, 2000.

[Chr07] Gerhard Chroust. Coping with system complexity: Identifying dichotomic architectural alternatives. *ERCIM NEWS*, (68):55–56, January 2007.

[CMN83] Stuart K. Card, Thomas P. Moran, and Allan Newell. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1983.

[Con05] FlexRay Consortium. Flexray communication system protocol specification version 2.1 revision a. Technical report, FlexRay Consortium, 2005.

[Cor89] Tom A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[Cow01] Nelson Cowan. The magical number 4 in short-term memory: a reconsideration of mental storage capacity. *Behavioral and Brain Sciences*, 24(1):87–114, 2001.

[CV03] Nick Chater and Paul Vitany. Simplicity: A unifying principle in cognitive science? *Trends in Cognitive Sciences*, 7(1):19–22, 2003.

[dAH01] Luca de Alfaro and Thomas A. Henzinger. Interface theories for component-based design. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, pages 148–165, London, UK, 2001. Springer-Verlag.

[EBK03] Wilfried Elmenreich, Günther Bauer, and Hermann Kopetz. The time-triggered paradigm. In *Proceedings of the Workshop on Time-Triggered and Real-Time Communication*, Manno, Switzerland, Dec. 2003.

[Ech03] Echeoln Corporation, San Jose, CA, USA. *Neuron C Programmer's Guide, Revision 7*, 2003.

[EF04] Jonathan St. B. T. Evans and Aidan Feeney. The role of prior belief in reasoning. In Jacqueline P. Leighton and Robert J. Sternberg, editors, *The Nature of Reasoning*, chapter 4, pages 78–102. Cambridge University Press, 2004.

[FCS01] Paul J. Feltovich, Richard L. Coulson, and Rand J. Spiro. Learners' (mis) understandings of important and difficult concepts: A challenge for smart machines in education. In K. D. Forbus and P. J. Feltovich, editors, *Smart Machines in Education: The Coming Revolution in Educational Technology*, pages 349–375. AAAI Press, 2001.

[Fel03] Jacob Feldman. The simplicity principle in human concept learning. *Current Directions in Psychological Science*, 12(6):227–232, 2003.

[FFL07] Fabrizio Fabbrini, Mario Fusani, and Giuseppe Lami. Software travels in the fast lane: Good news or bad? *ERCIM NEWS*, (68):54–55, January 2007.

[FH03] Lance Fortnow and Steve Homer. A short history of computational complexity. *Bulletin of the EATCS*, 80:95–133, 2003.

[FRR04] Paul J. Feltovich and Hoffmann Robert R. Keeping it too simple: How the reductive tendency affects cognitive engineering. *IEEE Intelligent Systems*, 3(19):90–94, 2004.

[GeA04] Miguel Goulao and Fernando Brito e Abreu. Software components evaluation: an overview. In *Proceedings of CAPSI'04*, Lisbon, Portugal, 2004.

[GFL+02] Paolo Giusto, Alberto Ferrari, Luciano Lavagno, Jean-Yves Brunel, Eliane Fourgeau, and Alberto Sangiovanni-Vincentelli. Automotive virtual integration platforms: Why's, what's and how's. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'02)*, pages 370–378, Washington D.C., USA, 2002.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[GIJ+03] Marie-Claude Gaudel, Valérie Issarny, Cliff Jones, Hermann Kopetz, Eric Marsden, Nick Moffat, Michael Paulitsch, David Powell, Brian Randell, Alexander Romanovsky, Robert Stroud, and François Taiani. Final version of dsos conceptual model. Technical Report 45/2002, Vienna University of Technology, 2003.

[Gil04] Kenneth J. Gilhooly. Working memory and reasoning. In Jacqueline P. Leighton and Robert J. Sternberg, editors, *The Nature of Reasoning*, chapter 3, pages 49–77. Cambridge University Press, 2004.

[Gir04] Vittorio Girotto. Task understanding. In Jacqueline P. Leighton and Robert J. Sternberg, editors, *The Nature of Reasoning*, chapter 5, pages 103–125. Cambridge University Press, 2004.

[Gri03] Klaus Grimm. Software technology in an automotive company—major challenges. In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, pages 498–503. IEEE Computer Society, 2003.

[Hay91] Samuel Ichiye Hayakawa. *Language In Thought and Action*. Harcourt, 1991.

[HBMB05] Graeme S. Halford, Rosemary Baker, Julie E. McCredden, and John D. Bain. How many variables can humans process? *Psychological Science*, 16(1):70–76, January 2005.

[HGK04] Imed Hammouda, Olcay Guldogan, and Kai Koskimies. Tool-supported customization of uml class diagrams for learning complex system models. In *Proceedings of the 12th IEEE Interna-*

tional Workshop on Program Comprehension (IWPC'04), pages 24–33, Bari, Italy, June 2004. IEEE Computer Society Press.

[Hoe04] Carl Hoefer. Causality and determinism: Tension, or outright conflict. *Revista de Filosofia*, 29(2):99–225, 2004.

[Hol95] John H. Holland. Can there be a unified theory of complex adaptive systems? In Harold J. Morowitz and Jerome L. Singer, editors, *The Mind, The Brain, and Complex Adaptive Systems*, volume XXII of *Santa Fe Institute Studies in the Sciences of Complexity*, pages 45–50. Addison-Wesley, 1995.

[HOP06] Bernhard Huber, Roman Obermaisser, and Philipp Peti. Mda-based development in the decos integrated architecture – modeling the hardware platform. In *Proceedings of the 9th IEEE International Symposium on Object and component-oriented Real-time distributed Computing*, Gyeongju, South Korea, April 2006. IEEE.

[HS96] Brian Henderson-Sellers. *Object-oriented metrics : measures of complexity*. Prentice-Hall, Upper Saddle River, New Jersey, 1996.

[HSF+04] Harald Heinecke, Klaus-Peter Schnelle, Helmut Fennel, Jürgen Bortolazzi, Lennart Lundh, Jean Leflour, Jean-Luc Maté, Kenji Nishikawa, and Thomas Scharnhorst. AUTomotive Open System ARchitecture—An Industry-Wide Initiative to Manage the Complexity of Emerging Automotive E/E-Architectures. In *Convergence International Congress*, Detroit, MI, USA, October 2004. Convergence Transportation Electronics Association.

[HSP04] Cindy E. Hmelo-Silver and Merav Green Pfeffer. Comparing expert and novice understanding of a complex system from the perspective of structures, behaviors, and functions. *Cognitive Science*, 28(1), 2004.

[HSS+07] Wolfgang Herzner, Rupert Schlick, Martin Schlager, Bernhard Leiner, Bernhard Huber, Andras Balogh, György Csertan, Alain LaGunnec, Thierry LeSergent, Neeraj Suri, and Shariful Islam. Model-based development of distributed embedded real-time systems with the decos tool-chain. In *Proceedings of the SAE 2007 AeroTech Congress & Exhibition*, Los Angeles, CA, USA, September 2007. SAE.

[HWP97] Graeme S. Halford, William H. Wilson, and Steven Phillips. Abstraction: Nature, costs, and benefits. *International Journal of Educational Research*, pages 21–35, 1997.

[HWP98] Graeme S. Halford, William H. Wilson, and Steven Phillips. Processing capacity defined by relational complexity: Implications for comparative, developmental, and cognitive psychology. *Behavioral and Brain Sciences*, 21:803–831, 1998.

[Int84] International Organization for Standardization. *Open Systems Interconnection – Basic Reference Model, ISO 7498:1984*, 1984.

[Int95] Intermetrics Inc. *Ada Reference Manual – Language and Standard Libraries, ISO/IEC 8652:1995(E)*, 1995.

[Jac01] Michael J. Jacobson. Problem solving, cognition, and complex systems: Differences between experts and novices. *Complexity*, 6(3):41–49, 2001.

[JL04] Philip N. Johnson-Laird. Mental models and reasoning. In Jacqueline P. Leighton and Robert J. Sternberg, editors, *The Nature of Reasoning*, chapter 7, pages 169–204. Cambridge University Press, 2004.

[KAGS05] Hermann Kopetz, Astrit Ademaj, Petr Grillinger, and Klaus Steinhammer. The time-triggered ethernet (TTE) design. In *Proceedings of the 8th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, pages 22–33, Seattle, Washington, May 2005.

[Kan03] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison Wesley, 2003.

[KB98] Rick Kazman and Marcus Burth. Assessing architectural complexity. In *2nd Euromicro Conference on Software Maintenance and Reengineering*, pages 104–112, Florence, Italy, March 1998. IEEE Computer Society.

[KB03] Hermann Kopetz and Günther Bauer. The time-triggered architecture. In *Proceedings of the IEEE*, volume 91, pages 112–126, January 2003.

[Kel03] Ronald T. Kellogg. *Cognitive Psychology*. Sage Publications, Inc., Thousand Oaks, California, USA, second edition, 2003.

[KN95] Hermann Kopetz and Roman Nossal. The cluster compiler – a tool for the design of time-triggered real-timesystems. *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems, June 1995, La Jolla, California, USA*, 6 1995.

[KN97] Hermann Kopetz and Roman Nossal. Temporal firewalls in large distributed real-time systems. In *Proceedings of the 6th IEEE Workshop on Future Trends of Distributed Computing Systems*

(FTDCS '97), pages 310–315, Tunis, Tunisia, 1997. IEEE Computer Society.

[KO87] Hermann Kopetz and Wilhelm Ochsenreiter. Clock synchronization in distributed real-time systems. *IEEE Transactions on Computers*, 36(8):933–940, 1987.

[Kop92] Hermann Kopetz. Sparse time versus dense time in distributed real-time systems. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 460–467, Yokohama, Japan, 6 1992.

[Kop97] Herman Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*, volume 395 of *Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, Boston, 1997.

[Kop98] Hermann Kopetz. The time-triggered model of computation. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS98)*, pages 168–177, Madrid, Spain, Dec. 1998.

[Kop99] Hermann Kopetz. Elementary versus composite interfaces in distributed real-time systems. *ISADS '99, March 1999, Tokyo, Japan*, 3 1999.

[Kop00] Hermann Kopetz. Composability in the time-triggered architecture. *SAE International Congress and Exhibition (2000-01-1382), Detroit, MI, USA*, March 2000.

[Kop03] Hermann Kopetz. Fault containment and error detection in the time-triggered architecture. In *Proceedings of the Sixth International Symposium on Autonomous Decentralized Systems (ISDAS 2003)*, pages 139–146, April 2003.

[Kop08] Hermann Kopetz. The complexity challenge in embedded system design. In *Proceedings of the 11th IEEE International Symposium on Object and component-oriented Real-time distributed Computing*, pages 3–12, Orlando, FL, USA, May 2008.

[KOPS04] Hermann Kopetz, Roman Obermaisser, Phillip Peti, and Neeraj Suri. From a federated to an integrated architecture for dependable real-time embedded systems. Technical report, Vienna University of Technology, Austria; Darmstadt University of Technology, Germany, August 2004.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, February 1978.

[KS03a] Hermann Kopet and Neeraj Suri. On the limits of the precise specification of component interfaces. In *Proceedings of the ninth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'03F)*, pages 26 – 27, Anacapri, Italy, October 2003.

[KS03b] Hermann Kopetz and Neeraj Suri. Compositional design of rt systems: A conceptual basis for specificatiom of linking interfaces. *6th IEEE International Symposium on Object-Oriented Real-Time Computing (ISORC'03)*, May 2003.

[KT07] Tevye R. Krynski and Joshua B. Tenenbaum. The role of causality in judgment under uncertainty. *Journal of Experimental Psychology: General*, 136(3):430–450, 2007.

[Lab73] William Labov. The boundaries of words and their meanings. In Charles James Nice Bailey, editor, *New ways of analyzing variations in English*, pages 340–373. Georgetown University Press, Washington, D.C., USA, 1973.

[Lak87] George Lakoff. *Women, Fire, and Dangerous Things - What Categories Reveal About the Mind*. University Press Chicago, 1987.

[Lap92] Jean-Claude Laprie, editor. *Dependability: Basic Concepts and Terminology in English, French, German, Italian and Japanese*. Springer Verlag, Vienna, 1992.

[Lea94] Doug Lea. Design patterns for avionics control systems. Technical report, SUNY Oswego & NY CASE Center, DSSA Adage Project ADAGE-OSW-94-01, 1994.

[Lei04] Jacqueline P. Leighton. Defining and describing reasoning. In Jacqueline P. Leighton and Robert J. Sternberg, editors, *The Nature of Reasoning*, chapter 1, pages 3–11. Cambridge University Press, 2004.

[LM99] Stephen Laurence and Eric Margolis. Concepts and cognitive science. In Eric Margolis and Stephen Laurence, editors, *Concepts: Core Readings*. MIT Press, July 1999.

[Log88] Gordon D. Logan. Toward an instance theory of automatization. *Psychological Review*, 95(4):492–527, 1988.

[Los05] Thomas Losert. *Extending CORBA for Hard Real-Time Systems*. PhD thesis, Vienna University of Technology, May 2005.

[LY01] Yang Li and Hongji Yang. Simplicity: A key engineering concept for program understanding. In *Proceedings of the 9th International Workshop on Program Comprehension (IWPC'01)*, pages 98–107. IEEE Computer Society Press, 2001.

[Mat04] The MathWorks. Simulink – simulation and model-based design, version 6. Technical report, The MathWorks Inc., Natick, MA, USA, 2004.

[MB98] Thomas J. McCabe and Charles W. Butler. Design complexity measurement and testing. *Communications of the ACM*, 32(12):1415–1425, 1998.

[MBSP02] Rainhard Maier, Günther Bauer, Georg Stöger, and Stefan Poledna. The time-triggered architecture – a consistent computing platform. *IEEE Micro*, 22(4):36–45, July/August 2002.

[MG03] Sarah Mittlefehldt and Tina Grotzer. Using metacognition to facilitate the transfer of causal models in learning density and pressure. Presented at the National Association of Research in Science Teaching (NARST) Conference, Philadelphia, PA, March 23–26, 2003.

[MGP60] George A. Miller, Eugene Galanter, and Karl Pribram. *Plans and the Structure of Behavior*. Number 32. Holt, Rinehart and Winston, New York, 1960.

[Mil56] George A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 63:81–97, 1956.

[MS99] Akira Miyake and Priti Shah, editors. *Models of Working Memory – Mechanisms of Active Maintenance and Executive Control*. Cambridge University Press, 1999.

[MTG04] Barnaby Marsh, Peter M. Todd, and Gerd Gigerenzer. Cognitive heuristics: Reasoning the fast and frugal way. In Jacqueline P. Leighton and Robert J. Sternberg, editors, *The Nature of Reasoning*, chapter 10, pages 273–287. Cambridge University Press, 2004.

[Nos97] Roman Nossal. *An Interface-Focused Methodology for the Development of Time-Triggered Real-Time Systems Considering Organizational Constraints*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstrasse 3/3/182-1, 1040 Vienna, Austria, 1997.

[NP89] Gregoire Nicolis and Ilya Prigogine. *Exploring Complexity*. W. H. Freeman and Company, New York, 1989.

[Obj03] Object Management Group (OMG). OMG Unified Modeling Language Specification. Technical report, OMG, 2003.

[OMG01] OMG. Model driven architecture: A technical perspective. Technical Report ab/2001-02-01/04, Object Management Group, February 2001.

[OP06] Roman Obermaisser and Philipp Peti. Realization of virtual networks in the decos integrated architecture. In *Proceedings of the 14th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS 2006)*, pages 8–15, Island of Rhodes, Greece, April 2006.

[OPK05a] Roman Obermaisser, Phillip Peti, and Hermann Kopetz. Virtual gateways in the decos integrated architecture. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, pages 134a–134a, Denver, Colorado, April 2005.

[OPK05b] Roman Obermaisser, Phillip Peti, and Hermann Kopetz. Virtual networks in an integrated time-triggered architecture. In *Proceedings of the 10th IEEE International Workshop on Object-Oriented Raal-Time Dependable Systems (WORDS'05)*, pages 241–253, Sedona, Arizona, February 2005.

[Par72] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[PBC+02] Claudia Picardi, Rosanna Bray, Fulvio Cascio, Luca Console, Philippe Dague, Oskar Dressler, David Millet, Berndt Rehfus, Peter Struss, and Christian Valle. Idd: Integrating diagnosis in the design of automotive systems. In *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI 2002)*, pages 628–632, Lyon, France, July 2002.

[PG00] David N. Perkins and Tina A. Grotzer. Models and moves: Focusing on dimensions of causal complexity to achieve deeper scientific understanding. Presented at the American Educational Research Association annual conference, New Orleans, April 24–28, 2000.

[POK05] Phillip Peti, Roman Obermaisser, and Hermann Kopetz. Out-of-norm assertions. In *Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2005)*, pages 280–291, San Francisco, CA, USA, March 2005.

[Pol96] Stefan Poledna. *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*. Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[POT+05] Phillip Peti, Roman Obermaisser, F. Tagliabo, A. Marino, and S. Cerchio. An integrated architecture for future car generations. In *Proceedings of the 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, pages 2–13, Seattle, Washington, USA, May 2005.

[PYJ04] Anthony Phillips, Diana Yanakiev, and Fangjun Jiang. Managing complexity in large-scale control system design. In *Proceedings of the 2004 American Control Conference*, volume 5, pages 4698–4703, Boston, MA, USA, June 2004. AACC.

[QKG05] Jose Quesada, Walter Kintsch, and Emilio Gomez. Complex problem solving: a field in search of a definition. *Theoretical Issues in Ergonomic Science*, 6(1):5–33, 2005.

[Ras00] Jef Raskin. *The Humane Interface*. Addison-Wesley, July 2000.

[Ray03] Eric S. Raymond. *The Art of UNIX Programming*. Addison-Wesley, 2003.

[RE06] Bernhard Rumpler and Wilfried Elmenreich. Considerations on the complexity of embedded real-time system design tasks. In *Proceedings of the IEEE International Conference on Computational Cybernetics 2006 (ICCC'06)*, pages 55–60, Tallinn, Estonia, August 2006. IEEE.

[Rea90] James Reason. *Human Error*. Cambridge University Press, 1990.

[Rec91] Eberhardt Rechtin. *Systems Architecting: creating and building complex systems*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[Ree99] Wayne Reeves. *Learner-Centered Design: A Cognitive View of Managing Complexity in Product, Information, and Environmental Design*. Sage Publications, 1999.

[Rei01] Daniel Reisberg. *Cognition*. W. W. Norton & Company, Inc., New York, second edition, January 2001.

[Res03] Mitchel Resnick. Thinking like a tree (and other forms of ecological thinking). *International Journal of Computers for Mathematical Learning*, 8(1):43–62, January 2003.

[Rie96] Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison Wesley, 1996.

[RK06] Bernhard Rumpler and Hermann Kopetz. Design comprehension of time-triggered real-time systems. In *Proceedings of the Junior Scientist Conference 2006 (JSC'06)*, pages 63–64, 2006.

[Rob04] Maxwell J. Roberts. Heuristics and reasoning i: Making deduction simple. In Jacquelind P. Leighton and Robert J. Sternberg, editors, *The Nature of Reasoning*, chapter 9, pages 234–272. Cambridge University Press, 2004.

[Ros78] Elanor Rosch. Principles of categorization. In Allan Collins and Edward Smith, editors, *Readings in Cognitive Science, a perspective from Psychology and Artificial Intelligence*, pages 312–322. Morgan Kaufmann, 1978.

[RS07] Bernhard Rumpler and Martin Schlager. Segmentation and conceptual chunking in embedded real-time system design. In *Proceedings of the ERCIM/DECOS Workshop on Dependable Embedded Systems at the 33rd EUROMICRO Confrence on Software Engineering and Advanced Applications (SEAA 2007) and the 10th EUROMICRO Conference on Digital Systems Design (DSD 2007)*, Lübeck, Germany, August 2007.

[RTC92] RTCA Inc., Washington D.C., USA. *DO-178B/ED-12B Software Considerations in Airobrne Systems and Equipment Certification*, 1992.

[Rum05] Bernhard Rumpler. Concept comprehension in computer science - a literature survey. Technical Report 31/2005, Vienna University of Technology, Institute of Computer Engineering, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2005.

[Rum06] Bernhard Rumpler. Complexity management for composable real-time systems. In *Proceedings of the 9th IEEE International Symposium on Object and component-oriented Real-time distributed Computing (ISORC)*, Gyeongju, South Korea, April 2006. IEEE.

[Rus99] John Rushby. Partitioning in avionics architectures: Requirements, mechanisms, and assurance. Technical report, Computer Science Laboratory, SRI International, 1999.

[RW98] Mitchel Resnick and Uri Wilensky. Diving into complexity: Developing probabilistic decentralized thinking through role-playing activities. *Journal of the Learning Sciences*, 7(2):153–172, 1998.

[RW02] Vaclav Rajlich and Norman Wilde. The role of concepts in program comprehension. In *Proceedings of the 10th International Workshop on Program Comprehension (IWPC'02)*. IEEE Computer Society Press, 2002.

[RWW07] Bernhard Rumpler, Roland Wolfig, and Carsten Weich. High speed and high dependability communication for automotive electronics. In *SAE Transactions Journal of Passenger Cars: Electronic and Electrical Systems*, volume 115, pages 365–369, USA, 2007. American Technical Publishers Ltd.

[Saw06] R. Keith Sawyer, editor. *The Cambridge Handbook of The Learning Sciences*. Cambridge University Press, 2006.

[SC94] John Sweller and Paul Chandler. Why some material is difficult to learn. *Cognition and Instruction*, 12(3):185–233, 1994.

[Sim81] Herbert Alexander Simon. *The Sciences of the Artificial*. MIT Press, Cambridge, Massachusetts, 1981.

[Sim95] Herbert A. Simon. Near decomposability and complexity. In Harold J. Morowitz and Jerome L. Singer, editors, *The Mind, The Brain, and Complex Adaptive Systems*, volume XXII of *Santa Fe Institute Studies in the Sciences of Complexity*, pages 25–43. Addison-Wesley, 1995.

[Spe01] Robert Spence. *Information Visualization*. Addison-Wesley, 2001.

[SS99] Manas Saksena and Bran Selic. Real-time software design – state of the art and future challenges. *IEEE Canadian Review*, 32:5–8, 1999.

[Ste04] Robert J. Sternberg. What do we know about the nature of reasoning? In Jacqueline P. Leighton and Robert J. Sternberg, editors, *The Nature of Reasoning*, chapter 16, pages 443–455. Cambridge University Press, 2004.

[SVM01] Alberto Sangiovanni-Vintecelli and Grant Martin. Platform-based design and software design methodology for embedded systemss. *IEEE Design & Test of Computers*, 18(6):23–33, 2001.

[Swa98] Delbert L. Swanson. Evolving avionics systems from federated to distributed architectures. In *Proceedings of the 17th Digital Avionics Systems Conference (DASC)*, volume 1, pages D26/1–D26/8. AIAA/IEEE/SAE, 1998.

[Tec07] Esterel Technologies. Scade language primer. Technical report, Esterel Technologies SA, Elancourt, France, 2007.

[TK74] Amos Tversky and Daniel Kahneman. Judgment under uncertainty: Heuristics and biases. *Science*, 185(4157):1124–1131, September 1974.

[TTT04] TTTech. TTP: Time-triggered protocol TTP/C high-level specification document, protocol version 1.1. Technical Report D-032-S-10-028, TTTech Computertechnik AG, Vienna, Austria, January 2004.

[TTT07a] TTTech Automotive GmbH, Vienna, Austria. *TTX-Build – The Node Design Tool for FlexRay Clusters*, 1.5.80 edition, October 2007.

[TTT07b] TTTech Automotive GmbH, Vienna, Austria. *TTX-Plan – The FlexRay Cluster Design Tool*, 1.5.70 edition, August 2007.

[TTT07c] TTTech Computertechnik AG, Vienna, Austria. *TTP-Build – The Node Design Tool for the Time-Triggered Protocol TTP*, January 2007.

[TTT07d] TTTech Computertechnik AG, Vienna, Austria. *TTP-Plan – The Cluster Design Tool for the Time-Triggered Protocol TTP*, January 2007.

[TU 05] TU Darmstadt. DECOS Deliverable D 1.3.1 – Specification of an interactive PSM development tool, February 2005.

[Tuf83] Edward R. Tufte. *The Visual Display of Quantitative Information.* Graphics Press, 1983.

[Tuf91] Edward R. Tufte. *Envisioning Information.* Graphics Press, 1991.

[vMV95] Anneliese von Mayrhauser and A. Marie Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

[Wal02] Andrew Walenstein. Theory-based analysis of cognitive support in software comprehension tools. In *Proceedings of the 10th International Workshop on Program Comprehension*, pages 75–84. IEEE Computer Society, 2002.

[War04] Colin Ware. *Information Visualization - Perception for Design.* The Morgan Kaufmann Series in Interactive Technologies. Morgan Kaufmann, San Francisco, CA, second edition, 2004.

[WR99] Uri Wilensky and Mitchel Resnick. Thinking in levels: A dynamic systems approach to making sense of the world. *Journal of Science Education and Technology*, 8(1):3–18, 1999.

[YC79] Edward Yourdon and Larry L. Constantine. *Structured Design:Fundamentals of a Discipline of Computer Program and System Design.* Prentice-Hall, 1979.

[Zha98] Jianjun Zhao. On assessing the complexity of software architectures. In *Third International Workshop in Software Architecture*, pages 163–166, Orlando, Florida, 1998. ACM Press.

Curriculum Vitae

Bernhard Rumpler

October 9th, 1977	Born in Neunkirchen, Austria
September 1984 – June 1988	Elementary school in Gloggnitz
September 1988 – June 1996	Comprehensive Secondary School ("Bundesrealgymnasium") in Neunkirchen
October 1996 – May 1997	Military service
October 1997 – May 2002	Studies of computer science at the Vienna University of Technology Graduation with distinction
May 27th, 2002	Master's degree ("Diplom-Ingenieur")
November 2002 – April 2003	Austrian state approved ski instructor training
September 2001 – ongoing	Different positions in the software development department at TTTech Computertechnik AG, Vienna; currently working as a project manager for automotive software projects
September 2002 – ongoing	PhD studies at the Institute of Computer Engineering of the Vienna University of Technology, supported by the FIT-IT program of the Austrian Federal Ministry for Transport, Innovation, and Technology