TIK-Schriftenreihe Nr. 39

Hans Otto Trutmann

Separate Connection and Functionality is the Pivot in Embedded System Design

A dissertation submitted to the
Swiss Federal Institute of Technology ETH Zürich
for the degree of Doctor of Technical Sciences

Diss. ETH No. 13891

Prof. Dr. Albert Kündig, examiner
Prof. Dr. Matjaž Colnarič, co-examiner

Examination date: October 23, 2000

© 2000 Institut für Technische Informatik und Kommunikationsnetze TIK, ETH Zürich

ISBN 3-906469-09-3

to J. H. W. & M. E.

Preface

The present publication marks the end of my time at the Computer Engineering and Networks Laboratory TIK of ETH Zürich. Beginning in 1989, for eight years my main focus has been the control system for the Hybrid III car. Apart from theoretical and conceptual questions, this has proven a comprehensive task with many practical aspects. It offered the chance for trying newly found concepts in a demanding real-world application. This thesis expands and generalizes these ideas.

I thank everybody who helped me, especially my thesis supervisor Prof. Albert Kündig, who gave me the opportunity to work on this project. The laboratory under his prudent guidance provided the necessary freedom in the search for my own solutions. This freedom added a strong sentiment of responsibility for their functioning; it also augmented my pleasure when they actually worked.

I also thank my co-advisor Prof. Matjaž Colnarič for his interest and remarks.

Nebojša Jelača deserves my thanks for his continuing efforts in hardware development and his patience also in difficult times, Dr. Hugo Fierz for his friendship and for the intense cooperation and many interesting discussions about all facets of embedded systems; the CIP method – developed by him and Hansruedi Müller – was one of the prerequisites for my work, and possibly also for the success of Hybrid III. Valuable contributions were made by students, among them Matthias Manhart, Patrik Reali, Hanspeter Schmid and Andreas Erne. Aside from all academic considerations, the touchstone for the envisaged solutions has always been their applicability to real-world problems; therefore I must also thank the Hybrid III team for having kept me in touch with the industrial reality.

I thank all my family and friends for making my life interesting and enjoyable. Most of all, I thank my wife Veronica Bürgler and my daughter Eva for being a source of pleasure and for teaching me more than I ever learned elsewhere.

Zürich, October 2000

Abstract

The goal in the development of embedded real-time systems is a solution that satisfies the initial requirements, although this is not enough. There is also a need for appropriate descriptions of the many different aspects of a system, for connecting these representations among each other and with the implementation itself, for easing maintenance and for dealing with organizational chores. All this makes the solution appear as a by-product of a well-organized development process. This development process profits from a clear separation of concerns, which can be achieved with an explicit interconnection model, the architectural pivot specification. Its relative stability allows development to progress independently behind stable syntactic and semantic interfaces.

The three basic activities in control applications, input, processing and output, are dealt with individually within separate problem areas. A problem-oriented approach postulates the use of suitable methods and tools for solving each of these issues, thus embodying different formal representations and explicitly written code on different levels of abstraction. Unison is achieved through generators yielding code with defined generic properties that favor smooth and efficient implementation. The functional problem comprises all the behavioral aspects of a control system, using shared phenomena to interact with the controlled processes in the environment. Solutions to this problem omit any details not pertaining to the relation between the course of events in the environment and the inner states of a model. The connection problem is concerned with information transport from and to the environment. Its solution is based on regular sequential structures.

During development, the common high-level source baseline permits generating implementations for various modes of operation without modifying the abstract descriptions. The approach eliminates manual programming when code alterations are needed to adapt to changes in the hardware environment. It thus allows testing of proposed solutions against numerical models, as well as adapting resources according to emerging requirements any time during the development process. It supports decentralized development work and simplifies target implementations without complicated scheduling, without interrupts, and without the overhead of elaborate operating systems. The code is statically deterministic, and the performance of entire implementations is analyzed exhaustively to assure the desired real-time properties.

Zusammenfassung

Das Ziel bei der Entwicklung eingebetteter Echtzeitsysteme ist eine der Anforderungsspezifikation genügende Lösung; das reicht jedoch noch nicht aus. Es braucht auch den verschiedenen Aspekten angemessene Beschreibungen, Verbindungen dieser Beschreibungen untereinander und zur Implementation, Unterstützung beim Unterhalt, bei der Weiterentwicklung und beim organisatorischen Ablauf. All das lässt die eigentliche Lösung dann eher als die Begleiterscheinung eines gut organisierten Entwicklungsprozesses erscheinen. Dieser Prozess profitiert von klar voneinander abgegrenzten Problembereichen; eine Abgrenzung, die mit der architektonischen Pivot-Beschreibung erzielt wird. Hinter solch syntaktisch und semantisch stabilen Schnittstellen kann innerhalb einzelner Problembereiche selbständig weiterentwickelt werden.

Die drei grundlegenden Aktivitäten in Steueranwendungen, Eingabe, Verarbeitung und Ausgabe, werden gesondert in eigenen Problembereichen behandelt. Ein problembezogener Ansatz postuliert den Gebrauch geeigneter Methoden und Werkzeuge für jedes dieser Gebiete, so dass es verschiedene formale Darstellungen neben explizit geschriebenem Programmtext auf mehreren Abstraktionsstufen gibt. Diese werden mittels Generatoren vereinigt, welche Code mit definierten generischen Eigenschaften liefern, der sich einfach und effizient implementieren lässt. Das funktionale Problem umfasst das Systemverhalten; die Koppelung des Steuersystems mit den zu steuernden Prozessen in der Umgebung läuft über Phänomene, welche von beiden geteilt werden, so dass Lösungen dieses Problems nur die Wechselbeziehung von Ereignisfolgen in der Umgebung mit dem inneren Zustand des Modells beinhalten. Beim Verbindungsproblem geht es um den Transport von Information vom und zum kontrollierten Prozess. Lösungen basieren hier auf sequentiellen Strukturen.

Aus den gemeinsamen abstrakten Beschreibungen lassen sich Implementationen für verschiedenste Betriebsarten generieren, was manuelle Eingriffe beim Austausch von Systemkomponenten überflüssig macht. Die erarbeiteten Lösungen können an numerischen Anlagemodellen ausprobiert werden. Der Ansatz unterstützt die dezentrale Entwicklung von Systemen und vereinfacht die Installation auf dem Zielsystem: keine komplizierte Ablaufplanung, keine Unterbrechungen und kein aufwendiges Betriebssystem. Der erzeugte Code ist statisch deterministisch, und das Zeitverhalten von ganzen Implementationen wird umfassend geprüft, um das gewünschte Echtzeitverhalten nachzuweisen.

Contents

1 Introduction **1**
1.1 Development Concepts 6
1.1.1 Separation of Concerns 6
1.1.2 Development Emphasis 7
1.2 The Toolbox 10
1.3 Overview 11

2 Embedded Systems **15**
2.1 Requirements 17
2.1.1 Functional Requirements 18
2.1.2 Temporal Requirements 19
2.1.3 Accuracy Requirements 24
2.1.4 Implementation Requirements 25
2.1.5 Communication Requirements 26
2.1.6 Reliability Requirements 28
2.2 Development 30
2.2.1 Building Principles 31
2.2.2 Architecture 32
2.2.3 Language Issues 34
2.2.4 Tools 36
2.2.5 Modes of Operation 37
2.2.6 Simulation 38
2.2.7 Performance Verification 39
2.3 Deployment Techniques 40
2.3.1 Deployment 40
2.3.2 Code Generation 42
2.3.3 Resource Management 43
2.3.4 Communication 46

3 Problem Decomposition **49**
3.1 Architectures 50
3.2 Separation of Concerns 51
3.2.1 Functional Problem Frame 54

3.2.2 Connection Problem Frame 55
3.2.3 Classification 57
3.3 Abstraction 58
3.4 Practical Development Suites 60

4 Architecture 61
4.1 Operational Architecture 62
4.2 Implementation Architecture 64
4.3 Component Architecture 66
4.3.1 Specification 68
4.3.2 Components 68
4.3.3 Connectors 71
4.3.4 The Pivot 73

5 Functionality 75
5.1 State Machine Models 76
5.1.1 Finite State Machines 76
5.1.2 Hierarchy 78
5.1.3 Cooperation 78
5.1.4 State Machine Extension 78
5.1.5 Semantics 79
5.2 Communicating Interacting Processes CIP 80
5.3 CIP Constructive Elements 82
5.3.1 Process 82
5.3.2 Synchronous and Asynchronous Cooperation 83
5.3.3 Behavioral Structuring 86
5.3.4 Extensions 88
5.4 CIP Implementation Units 90
5.5 Working with CIP Models 92
5.5.1 Functional Models within a Context 92
5.5.2 Initialization 95
5.5.3 An Eye on Interaction 95
5.5.4 Behavioral Patterns 96

6 Connection 97
6.1 Device Access 99
6.1.1 Representation of Continuous Signals 101
6.1.2 Encoding of Discrete Values 102
6.2 Data Acquisition 103
6.3 Input Data Distribution 108
6.4 Event Extraction 111
6.5 Output 112
6.6 Communication 114

7 Implementation **115**
7.1 Code 117
7.1.1 Functional Code 117
7.1.2 Connection Code 120
7.2 Deployment 121
7.2.1 Scheduling 122
7.2.2 Processing Scheduler 123
7.2.3 Implementation Variants 124
7.3 Dynamic Reconfiguration 128
7.3.1 Static Approach 128
7.3.2 Reconfiguration Strategy 130
7.3.3 Example 131

8 Hybrid III **135**
8.1 Hybrid Vehicles 135
8.1.1 Hybrid Concepts 136
8.1.2 Drive Systems and Energy Storage Devices 137
8.2 The Project Hybrid III 138
8.2.1 How it Works 140
8.2.2 Results 140
8.3 Hybrid III Components 141

9 Hybrid III Control **145**
9.1 Concepts 147
9.1.1 Processors 148
9.1.2 Communication 150
9.1.3 Input and Output 152
9.1.4 Bus Systems 154
9.1.5 Mechanical and Electrical Installation 154
9.1.6 Code 155
9.2 Methods and Tools 156
9.2.1 Working Environment 157
9.2.2 Configuration and Modeling 158
9.3 Hardware Implementation 160
9.3.1 Control System 160
9.3.2 Peripherals 162
9.3.3 Additional Hardware 162
9.4 Software Implementation 164
9.4.1 Communication 164
9.4.2 System Software and Connection 165
9.4.3 Functional Code 170

10 Conclusions **171**
10.1 Complete Solutions 173
10.2 Implementation 174
10.3 Reuse 175
10.4 Suggestions for Further Work 175

A Specification **177**
A.1 Architectural Specification 178
A.1.1 Specification Language 178
A.1.2 Productions 179
A.2 Input and Output Specification 183
A.2.1 The Generator 'acc' 183
A.2.2 Specification Language 183
A.2.3 Productions 184
A.3 Event Extraction Specification 189
A.3.1 The Generator 'evt' 189
A.3.2 Specification Language 190
A.3.3 Productions 190
A.4 Code Generator Program Structure 193
A.4.1 Data Structures 194
A.4.2 APs (Access Procedures) 197
A.4.3 IPs (Initialization Procedures) 198
A.4.4 Variables 198
A.4.5 Access Distribution 199
B Syntax **203**
B.1 Architecture 203
B.2 Input and Output 204
B.3 Event 205
C Glossary **207**
References **221**
List of Figures **230**
List of Tables **233**
Index **235**

1 Introduction

In our daily life we are surrounded by complex technical structures, most of the time without worrying about their safety and dependability. We are secure in the knowledge that buildings, bridges and means of transportation are based on the principles of sound engineering. In recent years we have exposed ourselves increasingly, and with the same trust, to the influences of complex computer programs, although we would be well advised to exercise more caution in this area. Contrary to disciplines like mechanical or electrical engineering, software engineering is in its infancy. It still lacks generally accepted ways and principles that would provide a reference frame for evaluating new solutions, and a wide gap remains between practitioners whose interest is to build and use computer controlled systems, and computer scientists who study the mathematics needed to analyze systems and programs [Par97]. Because the scientists who study programming understand little about the problems of engineering, they fail to explain how their proposed concepts could be put to use in real-world applications. If a solution demonstrated on toy problems does not scale, or if it neglects all the accessories needed to make this solution work, its possibilities are lost on practitioners. The products being built reflect this situation; they are developed by people whose actual job is not programming, but who need to program in order to carry out some other task; therefore, these products often present a major source of problems for those who depend on them. So, the undeniable advantages such as enhanced functionality, flexibility and cheaper production of computer controls come at a great cost and risk. On the one hand, an increasing number of serious incidents, often accompanied by significant human and financial losses, can be traced back to computer failures [Risks]. On the other hand, software development makes up an increasing share of the overall costs. If we are unable to develop a true engineering discipline with universal criteria in this area, then at least we have to ask how the existing methods can be put to effective use: what is the best possible way to develop systems that are reliable and cost-advantageous?

Since the time when electronic systems were introduced, applications have continuously become more complicated. This in turn entailed more demanding requirements with regard to the quality of such solutions and, simultaneously, to the quality of the development methods themselves. Unfortunately it seems as if all progress in this area is swallowed up by the growing complexity of the applications and by user demand that increases at an equal pace. Add to this the fact that software fails differently from mechanical systems. Mechanical systems and their ability to continue functioning to a certain extent even in the case of failure are in stark contrast to the infamous erroneous bit that unexpectedly paralyzes an entire system. Redundancy and the ability to gracefully degrade its functions were inherent in earlier controls in addition to the specified functionality and significantly contributed to the safety of a system. In the change over to computer solutions these features must be replicated or replaced entirely by different mechanisms. Regrettably this also does away with the previous separation between normal operation and the operation in the presence of failures. To make matters worse, it is not uncommon that the efforts required to manage these deviations far exceed those of the control task in normal operation.

The newly gained vast expressive power and flexibility in turn propagate more extensive requirements and result in complex implementations that are hard to control. While the individual developers themselves have not become much wiser during this process, the accumulated knowledge surrounding these issues grew immensely at the same time. For this reason, many developments nowadays require the commitment of numerous specialists in a process in which all team members pitch in to bring it to fruition. Cooperation, this oldest of approaches in resolving complex problems is reaching its limits, and the search for alternatives is getting more attention. The insight that in some areas tools already now have become indispensable has replaced the initial hopes of merely increasing productivity with the help of efficient tools. Contrary to humans, computer programs do become smarter in the course of time, as programs are continuously expanded and help in systematizing the collective knowledge.

Therefore, it is obvious that automation, i.e., the comprehensive use of programs, can offer solutions for some issues in the development process. However, the question is what parts of the process can be automated, and how can these parts reliably be tied together? The state of technological development today does not suggest that in the near future there will be computer systems, which can offer creative and intelligent solutions themselves. The performance of a system will still be limited to following simple rules and to applying them to large amounts of data, and it is still the developer who formulates the goals and the rules that are necessary to reach these goals.

Formal methods promised a way out of this situation. Not only should the requirements be specified formally, but it was even hoped to obtain the eventually required proofs of correctness and performance, as well as the executable code in the form of different, semantically equivalent projections from one and the same representation. Yet, despite the efforts of the past, formal methods only partially succeeded in making the jump from the universities to practical applications. Today most control systems based on computers are still programmed manually. There are several, partially contradictory reasons for this.

Academic arguments called for formal specifications, because only these could be analyzed, transformed and verified automatically. From this point of view the precision and completeness that are mandatory in the use of a formal method were considered to be advantages alone for a long time. The emphasis was placed on efforts to newly define the entire development process and to quickly obtain flawless systems as a perfect whole. It has in fact been shown numerous times that such monolithic solutions do work when they are applied to small sample situations in an ideal environment.

Users from industry have various objections. These are not computer scientists but engineers whose work must result in a product with the desired quality, completed within an acceptable period of time. The methods and tools they use must be reconcilable with the practical aspects of the solutions and must be integrated well into the development process. Wherever formal methods are available and offer solutions for partial areas, they remain isolated and can only be combined through extensive manual efforts. In other cases the underlying formalism remains all too visible and easily exceeds a programmer's mathematical knowledge. It is also not uncommon for formal methods to simply fail as soon as the tasks at hand exceed a certain size and complexity. In certain cases this is due to the fact that the 'formalism' behind commercially available tools lacks a solid mathematical basis, and that the resulting problems often only surface in the course of large projects. However, the same can happen with formally complete descriptions when they are applied to large problems, causing overloads in the form of what is known as 'state space explosion'. Finally, the industry has a hard time understanding the call for discarding the fairly successful development processes they use today in favor of something new. Thus, instead of the desired integration, researchers and practitioners seem to have drifted further apart, as the great number of conferences with an emphasis on purely academic topics suggests [Rei97].

All this must be seen in the larger context of planning and developing entire systems that is dominated by the question: what must the development process be like so that high-quality and competitive products are obtained as a, more or

less, secondary effect? With the exception of control system development (which is treated in more detail in chapter 2) each participating discipline now has established methods that contribute to reaching this goal [Dea92]. The areas that deal with these methods such as deadlines, costs, energy consumption, size, the use of a certain architecture or product line according to a company policy, the reuse of components and – increasingly important – short development periods, still are poorly linked. Instead of contributing to a solution of the control system problems they rather are the cause of more demands and additional complications in its realization.

Among the more significant reasons for the hesitant acceptance of formal methods are the limits set to the developer's understanding by a formal description itself. The study of a formal specification alone does not easily offer insights into the behavior of a system. In the scientific community, occupied with small example problems that are easier to oversee, this fact did not attract much attention. Only recently have the fundamental differences between the various representations of the system to be built entered the discussion: the mental image a developer has on the one hand, and its formal representation on the other hand [LCF97]. Related issues are the comparison of mathematics and computer applications [Sai96], or the attempts to unite the essence of specifications with their adequate presentation [ZaJ97]. Even the proponents of complete, executable specifications emphasize the importance of informal comments and descriptions of the environment [Hoa96]. These new trends may be related to a dampening of the initial enthusiasm when it comes to real world systems. Because only small problems can be verified formally until now, developers of complex systems would be left with a scarcely comprehensible representation of their problem – and no verification [Sch97].

The fact that the development of a new machine and its controls cannot but start out informally is frequently overlooked. The analysis phase produces texts, drawings, diagrams, etc., in other words a number of documents in a form that best describes the facts. The machine that is obtained at the end of the development process, on the other hand, contains many formal elements; in the case of a control system this comprises the code of programmable devices. Somewhere between the informal beginning and the formal ending there is at least one transition to formality hidden in the development process. This discontinuity is not to be confused with existing gaps between different subsequent formal presentations with different semantics, since development *always* starts out with a description that lacks a clearly defined semantics! It becomes more complicated because there is not just one but several descriptions that represent the same system or parts of the same system in different ways. In the course of the conversion, incompleteness and contradictions must be found and rectified.

From this point of view, the discussion for or against formal methods actually addresses the point in time when this transition should take place in the course of the project. Either it occurs towards the end, when finally an executable program is to be written, or the transition is made at the beginning. Obviously it requires just as much careful work to meet the demands of the semantics of a formal model on a higher level as those of a programming language. The question is, what is more rewarding? Even if it were practicable to provide a complete formal description of a system specification – with the hypothetical possibility of generating full implementations automatically – such a formal representation is only useful if the developers still understand what it is about. Examples of formalism that are suitable for producing executable code (e.g., Petri nets) or for verifying certain properties (VDM, Z) abound, but unfortunately they tend to obscure the issue at hand. Therefore they alone are of questionable use in a complicated iterative development process.

'…the formalization required when using any one given language for formal specifications cannot be expected to yield any particular advantage, but rather will be a burden on the programmers' (P. Naur [Nau92]).

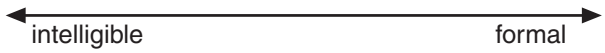

Figure 1: Comprehensibility vs. formality

Does this mean that there is only the choice between the lesser of two evils? More recent approaches attempt to combine existing technologies and to thus obtain the best of both worlds [Web97]. Formal methods are only employed in areas in which they are helpful and can be used cost-effectively. Such areas can be selected functions that require special proof of safety (this also includes the manual verification of individual components), or methods integrated in tools with which individual parts of an application are built. These tools are not only expected to provide complete and correct results, but must allow the user to understand and find his way around the model. The initial intentions must be clearly recognizable at all times. However, solutions that conjugate different technologies also have disadvantages, a major cause of problems being the combination of formal and non-formal representations. What starts out as a precise arrangement of separate problem areas frequently leads to a situation where the individual models increasingly separate or become incompatible where they overlap. Changes in one location cannot automatically be propagated to the entire project. This scenario is compounded if different versions come into play, and rigorous procedures are necessary in order to maintain consistency across the entire project and throughout the complete project duration.

1.1 Development Concepts

The development of embedded systems is complicated, because the considerable difficulties associated with customary non real-time software development are compounded by additional challenges, namely real-time response, reactivity, if worst comes to worst heterogeneous system structures, fault tolerance or even explicit proofs of safety. To date there is no single methodology that unites these concerns, there is not even a way to reconcile the results that can be obtained along the way from different methods.

A complex development is broken up into separate phases, where each phase deals with a certain problem area. Regardless of the method employed in a certain phase, it will require corrections, advancing in loops with small changes applied in each cycle. It is especially this iterative process that makes it difficult to preserve the consistency of all existing representations in an ongoing project, since often a correction on a low level (e.g., in the implementation) also results in modifications to the requirement specification at the top level. These issues can swamp the developers with a flood of new questions and severely distract from working on the actual problem solution.

1.1.1 Separation of Concerns

Structuring has always been the method of choice for dealing with comprehensive, complicated tasks by detecting and isolating individual areas that are decoupled from each other. One structure that seems ideal for embedded systems is the separation of the functional response of a control system from its actual connection to the process in the environment that is to be controlled (Fig. 2). Images of elements of the respective other side together with stable protocols at the interface of these areas must ensure that the parts remain compatible during the development process and ultimately in operation.

In many projects, however, all that remains is good intentions, since each possible approach is under pressure from all different directions, from the process features in the environment (that respond differently than anticipated) to the desired behavior as stated in the requirements (the specification was wrong or it was interpreted incorrectly from the beginning), to the characteristics of the hardware (the processor performance proves to be insufficient under the circumstances). All this violates the original concepts, up to the point where it becomes impossible to discern the envisaged separation in a final solution.

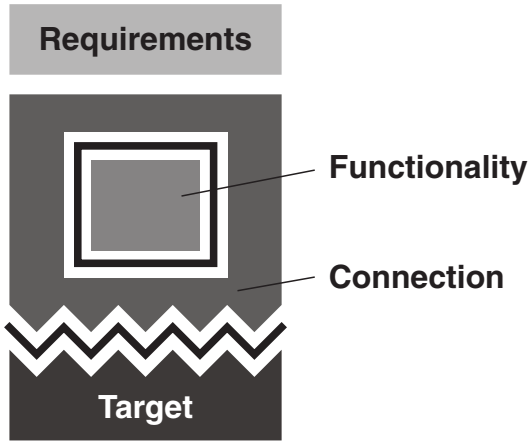

Figure 2: Separation of concerns

So, on the one hand, there is a development that moves, with or without repetitions, through different phases whereby different aspects are important. On the other hand, there are problem areas in addition to this time sequence that are isolated from each other and that ought to be addressed independently. The multi-layer structure determines what tools and methods should be used, especially since these tools often are truly useful only in a certain time phase or for a partial area. This would not be such an issue if it were possible to obtain flexibility in the application as well as consistency across the entire project, using a temporal and, with regard to the many different areas, transparent representation of the facts. Even with (or maybe even because of) generously designed efforts to obtain uniformity (as for example in UML [OMG99], or CDIF [EIA98]) this still does not seem to be possible.

1.1.2 Development Emphasis

As if the discrepancies regarding the problem areas were not enough, there are also significant differences in how to proceed. Today's customary approaches to development are situated between two extremes: either the emphasis is on the implementation, or functional aspects are considered more important.

Emphasis on Implementation. If the emphasis is placed on the implementation (Fig. 3), hardware characteristics dictate how to proceed, at the same time presuming that functionality is the smaller problem. The concepts on which this approach is based often require a separation of different spheres, too, and the task is approached accordingly. However, in the course of the development the limits between the solution that is found and the way in which it is realized become increasingly blurry. Often the original requirement specification disappears completely from the field of vision and thus is no longer of use during subsequent revisions.

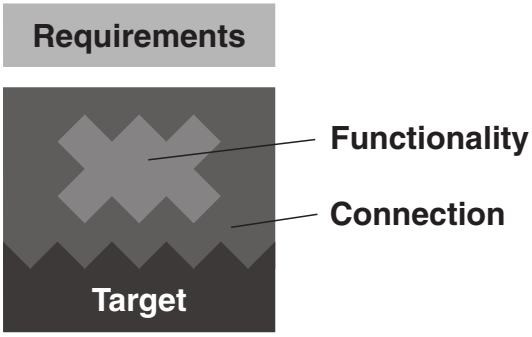

Figure 3: Emphasis on implementation

Many smaller applications have been successful with this concept, especially in the area of programmable logic controllers that has its roots in the replacement of relay controls. In the meantime these systems have become more complex, too, and quite often must be integrated into larger environments. The original way of programming these devices (with ladder diagrams) did not allow the use of structured techniques, and until integration was sought by defining a standard, several diverse programming methods were already in widespread use [IEC93]. In the variants now permitted by the standard, the emphasis unfortunately is on the syntax of these languages whereas it is often ambiguous about its semantics. Implementation differences and the lack of a comprehensible problem separation result in a growing number of cost- and time-intensive incidents, where initial operation phases are long drawn-out and cause damage because of functional deficiencies.

Based on the requirements, an example of a draft development cycle of such systems can be as follows, whereby the loop in the development process is concentrated exclusively in item 4.

1. prepare a functional concept,
2. implement the interfaces,
3. implement the functionality,
4. install and test the implementation in the target system.

Emphasis on Functionality. The other extreme of possibilities calls for an approach that largely abstracts from the implementation process and concentrates on functionality (Fig. 4). Especially academic solutions often are satisfied with input and output vectors instead of dealing with a true environment and driving actual interfaces. This is also due to the fact that the elements at the interface between hardware and software are very difficult to describe formally. It becomes even more difficult for many of these peripheral elements when the ways in which they must be operated come into play. For example, the acquisition of a single value may call for multiple accesses to a device, each with its

own timing requirement, which in an implementation may suggest the use of interrupts. Hence it becomes clear that with regard to the overall task the envisaged abstraction proves to be merely an incomplete description.

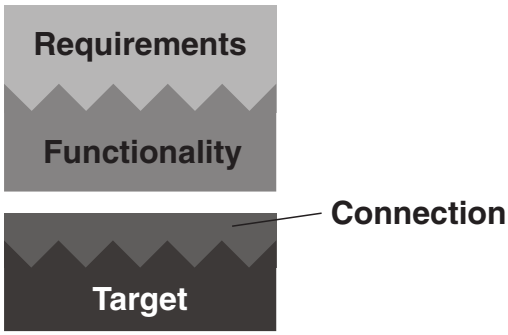

Figure 4: Emphasis on functionality

Since this approach pushes the development forward independently of the computer architecture and the peripheral devices for as long as possible, it results in problems that become obvious only in the later phase of implementation. For example, changing processor loads in the target system can have a negative impact on the response of controller algorithms or even cause instabilities in the control loops. In these cases, the goal may still be achieved with additional resources in the form of more or faster hardware. This possibility, however, remains barred in developments for consumer goods where economy is of main concern.

In the following example of a draft cycle for an adequately complicated system, a process model is being used. It shows that not even successful testing with the model protects from undesirable surprises when moving to the target system.

1. prepare a process model,
2. simplify the model until it can be used for the control design,
3. prepare the control structure,
4. design the algorithms of the control and fine-tune them iteratively until they seem acceptable when compared to the reduced model,
5. test the algorithms with the help of the process model. This phase may result in changes to the algorithms, the reduced model or even the process model,
6. integrate the implementation into the target system and test the hardware and software of the control system together with the environment.

This approach contains several development loops in items 4, 5 and 6, whereby especially the final step can be very cost- and labor-intensive. Oftentimes, serious problems do not show up until tests are performed in the final installation; these can be simple coding errors, unexpected influences of the computer

structure on the dynamic response, or new findings in general that did not surface during the testing process with the model, since the models are usually based on deficient abstractions that may hide relevant properties. The implementations in the testing environment and in the final system are very different and thus are hard to compare anyhow. If, after some fine-tuning, it becomes necessary to switch from the testing environment back to the model, the change is likely defective and requires extensive manual adjustments. In practical applications this step is hardly ever taken, so that the two types of implementation increasingly diverge. Such a set-up can still provide valuable services during the development phase since the participants are familiar with the subject. However, in subsequent expansions the knowledge of existing analogies will be lost, which is why the model will no longer be available for these processes either.

This technique has some other disadvantages. When determining the control structure in item 3, the assumption is that a top-down division of the overall problem into independent parts is feasible. This also implies that changes to a single module do not have any consequences for some other module. The assumption often does not hold and this is one reason why true testing only begins after the system is completely coded. Anticipated changes in the implementation have an effect on the entire structure and therefore are not only expensive with regard to cost and labor as described, but they also endanger the stability of the entire design.

1.2 The Toolbox

There is no doubt that embedded systems are complex. If the methods and tools used in their development are also complicated, they accentuate the problem instead of contributing to the solution. The wealth of the different ways in software development indicates that in software practically everything can be implemented, sold, and even applied; the decades-old prophecies of an immediate collapse of the software technology have not yet come true. This may change. Although it is probable that in consumer goods time to market and feature count continue to be the driving forces, in other areas software correctness will become important (again?). This mainly concerns equipment that is part of the infrastructure we rely on in our interactions with the environment: transportation, energy, health care, financial matters, etc. The growing proportion of software in these areas has been predicted to boost modeling techniques, code verification and analysis [JaR99]. The expectation is that the number of faults in embedded systems will decrease, once analysis of huge pro-

grams becomes feasible through progress in these techniques and the increased performance of future computers. All is not lost even before these wishes become reality, because, in addition to being a prerequisite of formal verification, models also serve another purpose. A model based on a formal foundation helps in clarifying concepts through simplicity and uniformity, which is a major contribution to keeping the number of possible mistakes at a minimum. Therefore, the presented approach is based on three simple principles:

- consistent separation of two spheres: functionality on the one hand, and its connection to the processes in the environment on the other hand,
- specification and modeling on a high level of abstraction; this concerns not only system behavior and response, but also the embedding into a hardware environment,
- replacement of manual coding through efficient, generated code with deterministic properties that can be statically analyzed; a prerequisite for lean solutions.

1.3 Overview

This thesis presents nothing new. Most of the accommodated techniques have been known for decades. The aim was to offer a coherent solution for the perceived need of practitioners that incorporates and reconciles methods and tools for each development stage. In an attempt to separate the wheat from the chaff in today's abundant propositions of how to tackle the different aspects of real-time systems, the emphasis has been on identifying a required minimum set of such techniques. It is considered essential that a clear separation of concerns can be achieved, which encourages limitations of a tool's power to what is needed. Because these tools are closely related parts of a whole, their teamwork in mutual consideration assists in making implementations simpler and helps cut off a whole welter of consequences that otherwise must be dealt with. From the point of view of the developers, the different aspects of a project present themselves on a high level of abstraction, whereas in the implementation this approach does away with dynamic scheduling, resource contention, priority inversion, sporadic tasks, all of them objects of their own disciplines in computer science. At the same time it improves on the issues of predictability and efficient use of processor resources.

In chapter 2, embedded systems are presented in more detail and an attempt is made to define *minimum requirements,* taking into consideration all the different aspects. These requirements lead to claims that must serve as guidelines

when the combination of methods shall lead to efficient and lean solutions. The economic use of resources goes hand in hand with implementations that are as small as possible, i.e., not cluttered with puffed-up additions not pertaining to the actual solution itself. During development, on the other hand, a high level of abstraction must be envisaged, with representations of the various problem areas in terms of the respective problem itself. All development phases benefit, if the principles applied in different problem areas are based on the same grounds and share more than the syntax at an interface. Aside from introducing different views on architecture, chapter 3 describes two kinds of problem decomposition. One is concerned with *layers of abstraction* dividing specifications and models from the implementation. The other one deals with the *separation of two fundamental concerns* and a way to guarantee the interplay of the particular models. The correspondence does not only exist on a high abstraction layer, it is equally true for the code that is produced from such models. Considerable advantages for the implementation result, if this code conforms to certain restrictions instead of just being a collection of independently schedulable black boxes. The braces that hold these spheres together is the *pivot,* the component architecture, which is presented in chapter 4, whereas chapters 5 and 6 are devoted to the methods used to tackle the problems within each sphere. *Functionality* is modeled in the graphic environment of CIP Tool [CIP00]. Its expressive power is appropriate to what is needed in embedded system development, and the code produced by the tool is efficient and deterministic. The *connection* with a real hardware environment is achieved on an equally high level. Here, specifications are translated into code that complements the functional code. In chapter 7, the *implementation* is considered further and the obtained code blocks and their calling conventions examined. The properties of this code are the prerequisite for implementations that excel in simplicity and have deterministic timing behavior. Appendices A and B present the *specification languages* and tools used in the solution of the connection problem.

Original contributions center on the architectural view, the connection problem, and the implementation. All these aspects are dealt with on a high level of abstraction, to this end specification languages were defined for the component architecture, as well as for low-level data manipulation associated with input and output. Code generators provide an automatic link from the high level descriptions to implementations. The implementations for their part benefit from the strictly controlled properties of the generated code. As a side-effect, worst case execution paths need not be extracted painfully from the final code, but are provided by the generators directly.

The claim of the presented approach is its usefulness in solving the *entire* problem that the development of complex embedded systems presents. To demonstrate this capability it was put through its paces in a demanding application, namely the project Hybrid III. In chapters 8 and 9 its system structure and control system is described at length. Hybrid III attempts to improve fuel efficiency and to reduce emissions of private vehicles in the entourage of the perceived need for less polluting traffic in the industrial capitals at the end of the first automotive century. The idea consists not so much in developing a completely new kind of vehicle, but mainly in pushing the limits of current drive technology to the extremes. This project was the touchstone for many of the concepts presented and it showed all the attributes of a real-world situation. It started out with vague specifications that were changing continuously, it involved several institutions with many people from various fields and of different expertise. Control system technology had to stand back in view of the 'real' problems the mechanical and electrical engineers were facing, they even chose to ignore it completely as long as it functioned correctly. It would have been inadequate, to say the least, if the methods we proposed had only satisfied theoretically. Project management, various operation modes with frequent changes from one to the other, and the coordination of many contributors, brought about additional hardships to the routine issues of a hard real-time system. In the end, a functional system had to be delivered. It has been our ambition to reach this goal as a side effect of a consistent development setup that emerges from a combination of selected methods, no hacking involved.

In an attempt to maintain the train of thought throughout the chapters, the explanation of recurring, often-used terms is subsumed in a glossary.

2 Embedded Systems

Fast technological advances in the field of hardware accompanied by decreasing costs resulted in the current trend towards computer supported controls for almost all industrial processes and many devices in daily life. These are electronic systems that fulfill a limited number of predetermined tasks. Since these tasks usually remain unchanged over the course of the entire life of a device, such systems cannot be programmed freely like, for example, a workstation. The applications differ widely and reach from radios and washing machines to safety related tasks in airplanes and medical devices. Computers that are components of a larger system whose primary purpose is not computation, are called embedded systems; they control their environment and are connected to it via sensors and actuators [Kün87]. Aside from their much more specific application, the major difference to a workstation is indeed to be found in the environment. The environment of embedded systems is not intelligent and is controlled by the computer and not vice versa (Fig. 5). This is also the reason why incorrect operation of an embedded system can have implications for the safety of its users and their surroundings [Sto96]. Since the consequences of failure can be severe, these machines must behave correctly without intervention also in the presence of faults, which implies that they handle errors all by themselves and recover from unexpected digressions.

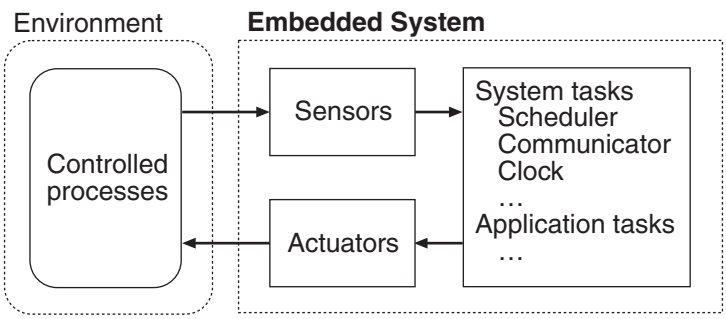

Figure 5: Embedded system

Many embedded systems are reactive, i.e., in a continuous loop such a system first tests a number of inputs for changes, then processes them to produce a number of output values. These systems are integral parts of a feedback loop with the external world. Embedded computers need not be small, or even all that real-time, indicating that the terms 'embedded' and 'real-time' are independent of each other.

The processes in the environment are parallel and determined by physical, chemical or other principles that in turn depend on the respective application. Continuous information is translated to its discrete representation at the interface between environment and control system, and even time is represented by a discrete approximation. Since the peripheral hardware is able to convert many input values simultaneously, there still is a high degree of parallelism at this point. In most customary control applications, a single processor then reads the data individually and processes them sequentially. Distributed control systems, on the other hand, work in parallel with several processing nodes. This makes it easier to adjust the required computer performance to the control issue at hand and facilitates separate development. It also is advantageous with regard to maintenance and the reuse of components. Either way, conventional solutions contain quite a few parallel processes for which it is very difficult to guarantee that they cooperate smoothly, and that the timing requirements are met under all conditions. Efficient strategies for cooperation and communication within the system have a key role in these applications.

Often the calculations that a control system performs are time-critical. It is in these cases that embedded computers are real-time systems with stipulated response times to certain sensor signals. The reactions must not only be functionally correct, but they must happen at the correct time if such a requirement is postulated (temporal correctness). The scale of the desired real-time response varies very much and can range from seconds to µs. Hard real-time cases are marked by a specification of a maximum response time that must never be exceeded. Soft real-time, on the other hand, allows for a defined statistical distribution of response times for each and every action, always at some required heavy level of processor activity. Timing requirements are quantifiable and measurable, and they either are postulated directly as part of a system's specification, or they may emerge as a consequence of other stipulations. For example, the desired positioning speed and accuracy of a machine tool dictates the minimal sampling rate for the measured quantity 'position'. Aside from satisfying timing requirements, which is mandatory in hard real-time systems, it is also desirable to obtain a high utilization of the resources that are present in an installation. If a high production volume is planned, reducing the indispensable resources to a minimum may become a distinctive requirement in itself.

Currently, developers often seem to be satisfied with *any* feasible result that fulfills these requirements, although it would be possible and preferable to find optimal solutions for both the timing and performance demands. This becomes possible when all related elements have deterministic behavior that can be analyzed in advance [XuP93]. Fortunately, hard real-time requirements usually are restricted to parts of a system, which in turn helps to achieve acceptable computer utilization. As outlined, failures in real-time systems do not necessarily imply incorrect results from computation, they also refer to (and presently more often) violations of temporal requirements. Attempts to counter this type of failure involve generous hardware resources that reduce the average performance of real-time systems compared to that of non-real-time systems. This implies that a system cannot be chosen on the basis of its average performance, if latency or worst case performance is the key issue.

Work on embedded systems is characterized by cross-development, i.e., the machine where the development takes place is not identical with the final target. The advantages offered by a powerful workstation environment with sophisticated tools and network facilities for cooperation in a team are outweighed by the difficulty of gaining access to programs and data once they are installed and running in their targets. This is notably true in the case of distributed systems, which are virtually impossible to observe and debug during operation [Dan97].

2.1 Requirements

The one advantage in the development of embedded systems is that the characteristics of the application and its operating environment are more precisely known than that of general purpose machines. Using this information, it is possible to fine-tune real-time systems accurately for optimum performance.

Aside from the requirements raised by a client specification that describes what a system should do once it reaches completion, an additional collection of various requirements is stipulated by the characteristics of a real-time system's principal constituents (Fig. 5). Most of the demands are dictated by the processes in the environment and they also depend on how external information is conditioned for use by the digital embedded system. This system itself consists of a possibly large number of processors, equipped with local resources for computing and storing data, and interconnected by a real-time communication network that provides bounded communication delays. The real-time operating systems that manage the use of processors and resources range from simple executives to complicated aggregates that handle preemptable tasks and resolve

arising conflicts in resource usage. Performance must be traded for reliability in fault tolerant systems that require additional management of redundant hardware and software components.

There are two fundamentally different paradigms that are referred to in the characterization of real-time systems. In *event-triggered* real-time systems, any system activity is initiated in response to the occurrence of a particular event in the system environment, whereas in *time-triggered* systems, activities are initiated at predefined instants of the globally synchronized time [PaD93]. It is a common misconception that the paradigm best suited to describe one specific view of a system (usually the implementation) must extend to the whole and include all the different viewpoints from the abstract model to the target implementation. In fact, the best way of dealing with functionality on a high level of abstraction is in stark contrast with how this functionality should be implemented, especially in cases where predictable behavior must be guaranteed. Because in the event-triggered approach the expiration of time can be treated as a sequence of time events, and in time-triggered systems discrete events are merely deferred by some predefined amount of time, the two ways of dealing with the problem are really equivalent. This is why there is a choice of the different representations for each level of abstraction.

The following compilation attempts to highlight the essential requirements and to draw up a number of claims that are raised against any suggestions of how integral embedded system development should look like. Somewhat contradicting demands are inevitable, hence there is no way to a solution without compromise. Nevertheless, these guidelines provide valuable assistance in discriminating essential properties from those that can be more easily sacrificed.

2.1.1 Functional Requirements

The objects in the environment exhibit a specific behavior according to their physical, chemical etc. properties and their interdependencies. It is the goal of a control system to bring about specific interactions between these objects. This interaction can be realized by means of the reactive behavior of a computer program, in an analogous way that it could be achieved pneumatically or mechanically. The essence of a customer's requirements is just what this behavior should be. Unfortunately, and for a variety of reasons, the conceptions available at this stage are incomplete, often contradictory and do not provide enough detail to serve as a basis for further reasoning [LCF97]. In order to be able to continue from here, a new comprehensive picture of the system to be built must be created using a notation that permits the customer to understand

what it is about and which can at the same time serve as a foundation of further progress. The means by which a system's functionality is described must therefore be based on some kind of formal model, using textual and graphic representations for defining systems, processes, their behavior, and their interrelations. Notations exist in many different variations: formal description languages that were created for use in a special application domain (such as SDL for the telecommunications industry), state-based notations especially suited for reactive systems (of which Statecharts is a prominent, but unfortunately only partially formal example), or strictly formal languages, which are applied when the emphasis is on formal verification (such as Petri Nets, VDM or Z). In any case, it would be most desirable that a high level description orients itself along the lines of the informal requirements. These come in a variety of representations including graphs, pseudocode snippets and natural language. Hardly ever do they coincide with the structures of formal models such as Petri Nets or algorithmic languages, instead they very often already show the characteristics of automata with a finite number of states.

What is the most suitable paradigm for these representations? On the one hand, the internal behavior of embedded systems, where any change of state in the environment calls for a reaction, makes them a perfect match for the event-driven model, on the other hand, an increasing number of these systems incorporate control loops that require periodic execution irrespective of the course of events in the environment. These parts of a system suggest the application of the time-triggered paradigm, but, because the reaction to asynchronous events presents the most demanding functional issue, it seems appropriate to choose a description based on events for the entire problem area. The passing of time is then treated the same as all the other (sporadic) events originating in the environment.

Claim 1: The problem at hand must have a comprehensive representation at a high level of abstraction. This representation must accompany the system and its developers through all development stages and allow for seamless transitions into implementations. The number of additional preconditions imposed on the implementation must be minimal.

2.1.2 Temporal Requirements

Temporal requirements are raised by the properties of the real-time processes in the environment. A distinction has to be made between two different kinds of temporal demands. One is concerned with the amount of time that is allowed to elapse until the system reacts to a *discrete event* in the environment. The

other kind of temporal requirement is associated with external processes that are part of periodically executing *control loops.* Examples of the first type, which may require either hard or soft real-time response, are the reactions to the push of a button, or to the event of a continuous signal exceeding some predefined limit. Hard real-time response is indispensable for the cyclical execution of control algorithms that belong to the second category. In addition to the usual maximum reaction time, an upper bound on input jitter is imposed for control loops. In absolute numbers, the spectrum ranges from man-machine interfaces with modest calls for maximum response times in the range of 40 ms (instantaneous for a human observer), to the demanding control loops in solid state switching for electrical drive applications, which are among the fastest within the realm of today's control systems. For digital signal processing, continuous signals must be quantized at some discrete interval after which they appear to the computer as a sequence of digital values, i.e., values that are limited to discrete steps in amplitude. This quantization adds random noise to the signal with a uniform distribution among $\mp 1/2\ LSB$, LSB being the least significant bit, corresponding to the distance between adjacent quantization levels.

The time interval and the accuracy (or resolution) of this transition must be chosen according to the properties of the controlled process, with shorter periods resulting in better closed loop characteristics. Within limits it is possible, however, to vary the sampling time and adapt the control laws such that the characteristics remain acceptable. As a rule of thumb, sampling intervals for a signal must amount to approximately $1/10$th the process' characteristic time constant d^{rise} in Fig. 6 [Kop97, Unb87].

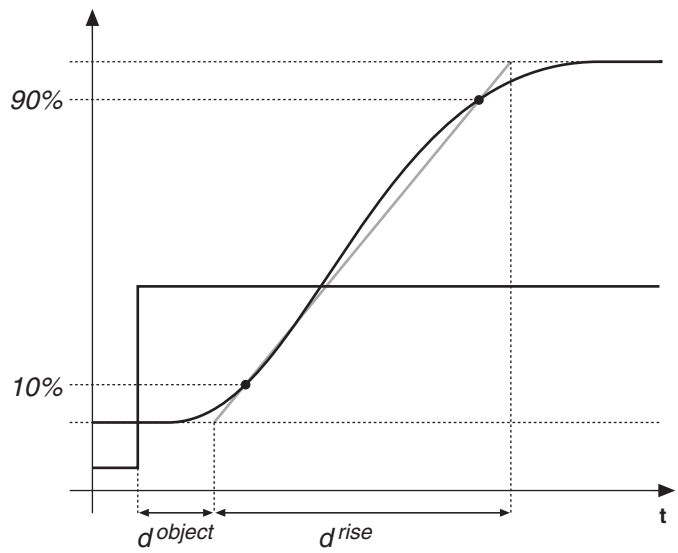

Figure 6: Process step response

Sampling. Together with quantization, digital signal processing depends heavily on sampling. The signals in the environment evolving continuously in time must be represented by just a set of their values. According to the sampling theorem, the sampling rate must be greater than twice the signal's highest frequency component, if the properties of the signal shall remain unaltered through this process [Cat69]. Data under observation must be band-limited to a frequency less than a critical frequency, otherwise aliasing and severe nonlinear distortion result. Because the sampling rates are chosen in accordance with the interesting part of a signal's spectrum, low-pass filters (aptly called anti-aliasing filters) must be used on the analog side of the conversion to precondition the input signal [Smi97].

In practical applications, the characteristics of the available non-ideal filters introduce selective attenuation and phase shifts (Fig. 7). Therefore, finding feasible combinations of sampling rates and filter characteristics involves a trade-off between filter complexity and sampling speed, and it always results in the need of higher sampling rates than postulated by the sampling theorem. The effect of using too low a sampling interval in closed loop control is first an increase in settling time and in extreme cases instability.

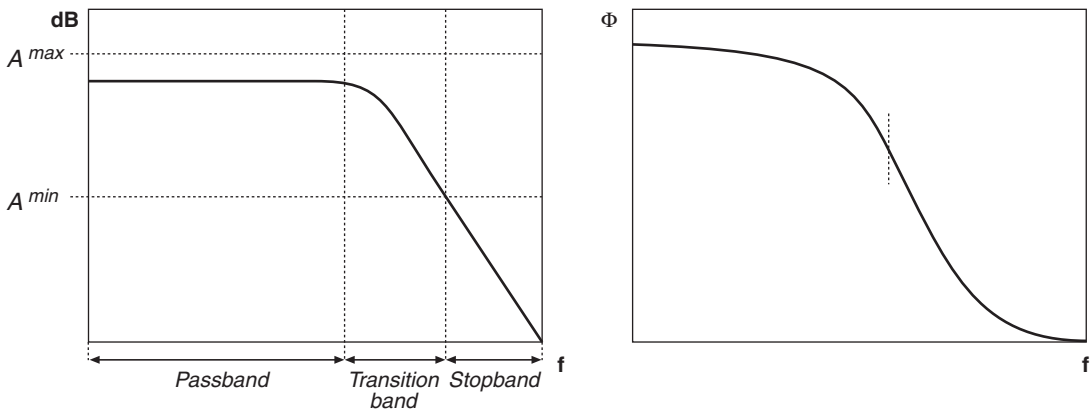

Figure 7: Filter magnitude and phase vs. frequency response

Jitter. Implementations of control algorithms have an additional temporal requirement. It is not caused by the processes in the environment, but rather by the way continuous information is fed into the digital system. When discrete control algorithms are based on a fixed sampling rate for data capture, and this frequency is subject to jitter (which may be caused by scheduling issues, or interfering interrupts), these algorithms may suffer from stability problems. The reason is that input signals experience a measuring error proportional to the temporal variation of the instants when the samples are taken (Fig. 8). Especially signals with a steep gradient introduce disturbing excursions of the

read values that may eventually cause the control algorithm to fail. The worst case occurs when a signal is sampled alternatively at the beginning and at the end of the sampling period.

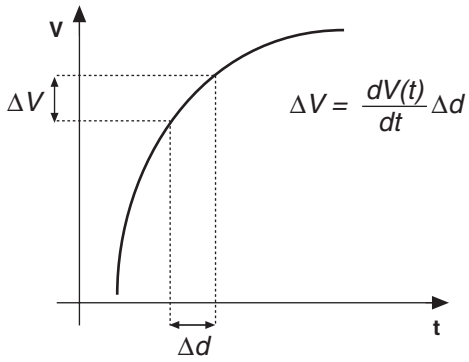

Figure 8: Jitter effects

The majority of currently proposed implementation techniques build on task scheduling schemes that are based on priorities with exclusion and precedence constraints only. Such implementations leave the extent of jitter undefined. This situation is aggravated in systems that rely on the use of hardware interrupts to capture sporadic events. The consequence is that control engineers tend to push the sampling rates in their requirements, so they can use filters to eliminate jitter effects from a signal. Unfortunately, additional filtering introduces longer time delays into the control loop, and, if a system contains many such signals that are actually oversampled, the total work load rises unnecessarily. This in turn may induce additional scheduling problems that will cause even more jitter. Even if the variance in input sampling rate is small, jitter can be present on the output signals, which is caused by delays in processing, or because signals are buffered before they are written to their respective devices. Usually, this is not a problem, since actuators and their connected machine parts can tolerate some jitter without adverse effects. Furthermore, jitter may also be introduced when deadlines must be sacrificed for some other more important issue. For example, in a radar system it is crucial to avoid data loss even at the expense of missed deadlines. By using buffers between the radar hardware interface and the subsequent processes the latter can be preempted, so no data is lost. Instead, the arrival time of processed data will gain some jitter. A system with multiple dedicated processors is very effective in reducing this kind of jitter.

Claim 2: Sampling intervals for periodic closed-loop control tasks must match the ones postulated by the problem analysis based on control theory. Input jitter for these signals must be kept minimal, and its amount must be controllable and guaranteed.

Soft Real-Time. In contrast to hard real-time requirements, where it must be ensured that all deadlines are met, there are some choices in soft real-time.

One possibility is to handle these requirements no different from those for hard real-time, a frequent option when the resources present in a system surpass the summed up requirements. If this is not the case, processing time that is available to those parts of a system with soft real-time requirements must be partitioned according to functional properties of these tasks. Different policies may be appropriate: possible failures to meet a deadline can be taken into account by stopping a task and handing the processor over to another one. Or, for example, in a video conferencing system, the last frame is kept when the data for the next frame has not arrived by the expected deadline. An interesting possibility in soft real-time are algorithms that are inherently soft, i.e., algorithms that lead to increasingly better results over time. Examples for this kind of approach can be found in vision decision systems. There, computation continues, with possibly different algorithms, until the deadline arrives. This approach requires at least one (the simplest) algorithm to have a hard deadline. It must be noted that picking a strategy that is adapted to a specific problem can be more challenging than determining which parts of a system must be designed to meet which deadlines.

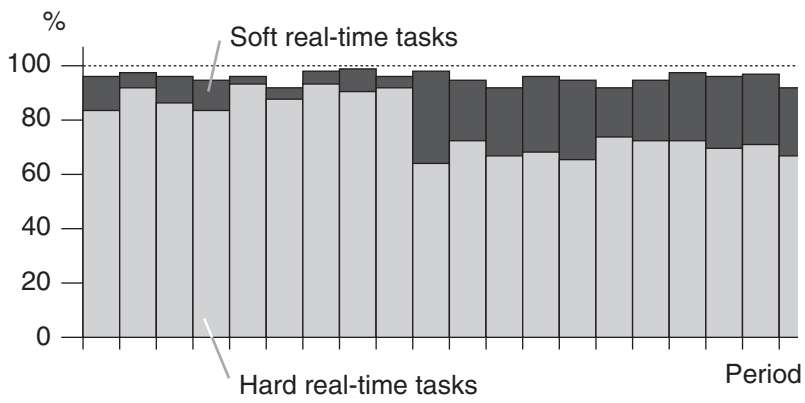

Figure 9: Variations in processor resource allocation

Although embedded systems with only soft real-time requirements do exist, in the case of hard real-time requirements this is hardly ever true for entire systems. There, hard and soft real-time requirements appear side by side and facilitate optimal resource utilization even when the processor loads generated by hard real-time tasks vary greatly. Thus, when hard real-time demands decrease, resulting from a change in system behavior such as shown in Fig. 9, the freed capacity will simply be allotted to pending soft real-time tasks. This, however, is only possible when the tasks with soft real-time requirements are fully preemptable, i.e., it must be possible to interrupt them at any time without

adverse effects on the schedules of hard real-time tasks and the availability of resources needed by them.

Claim 3: Tasks with soft real-time requirements must be preemptable, and their use of global resources must be limited to issuing requests through buffering mechanisms.

2.1.3 Accuracy Requirements

Input and output values in embedded systems are characterized by having a fixed range, which is determined by the physical process and the technology used in specific sensors and actuators. The values within this range are expressed with a finite resolution, determined by their binary representation and by the mechanism employed when converting from analog to digital or vice versa. One stage in the chain of involved elements decides on a signal's resolution. Very often (as in the example in Fig. 10) the severest restriction is brought about by the accuracy of the sensor itself, which is easily surpassed by the accuracy of an integer representation within the control system. From the point of view of resource adequacy the optimum would be to choose matching representations on all stages involved. This rules out floating point in many cases, although a direct conversion to this format may be preferable in systems where a floating-point unit is present and used to perform calculations.

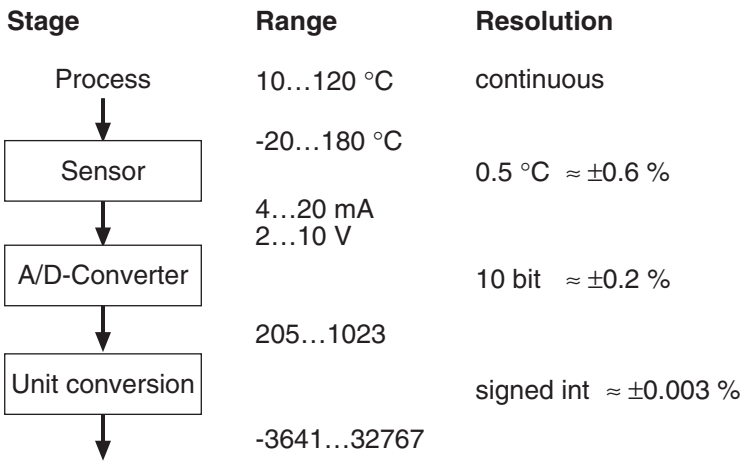

Figure 10: Input signal range and resolution

Absolute accuracy requirements are often rather lax for various reasons. Value inaccuracy frequently is not brought up by the ways data is handled in digital systems, but it is inherent in the physical processes themselves. The presence of ripples and other disturbances in a tank, for example, makes it pointless to measure its filling level with meticulous accuracy. In another context, measured

values may have chiefly relative significance, particularly the ones used in control loops. The reason is that a controlled process can only be stabilized with appropriate measurement repeatability, whereas the actual accuracy of these measurements is of secondary importance.

Claim 4: The accuracy of signals and their representation must be adapted to the chain of elements from a process' physical properties to the data formats required by the control system.

2.1.4 Implementation Requirements

Implementation requirements depend on the characteristics of the hardware environment, as well as on the paradigm that forms the basis of a specific implementation, which as described is either event-triggered or time-triggered. Both strategies require a resource management that guarantees the availability of the resources needed by a task, prior to the execution of that task. In both approaches the resource needs and availability assessment must be carried out while taking into account the timing requirements of the applications. Because in event-triggered systems the resource needs and their availability may vary at run time, resource management becomes dynamic and hence much more difficult. On top of that, timeliness guarantees may be violated in case the assumed peak load is exceeded during run time. Time-triggered systems benefit from a timeline divided into regular intervals, which allows for an off-line computation of resource needs. These needs are based on estimation, code analysis, or execution time measurement prior to deployment. Furthermore, time-triggered systems do not face the excessive number of possible behaviors, which in event-triggered variants must be carefully analyzed in order to establish their predictability [Jos96]. In comparison, event-triggered solutions are more flexible and less prone to waste resources.

While the choice of a paradigm for the implementation is often tightly linked to the paradigm chosen for the functionality, it is important to note that this is not necessarily so (as described in 2.1.1). On the contrary, the right choices in these two spheres may be exactly opposed, and it is not a problem to incorporate functionality based on an event-triggered model in a time-triggered implementation. These implementations are easier to master and the code's compliance with timing requirements on specific targets can be checked before installation.

Implementation efficiency is another important factor in embedded systems. The necessity of efficient coding to make optimal use of processor resources and to find one's way in tight memory occupation is one reason why program-

ming in assembly language is still widely used. This is especially so in areas where small systems and large numbers of installations are envisaged and the goal is to make do with minimum hardware.

On the other hand, the enormous performance increase of today's processors in the field of complex applications has resulted in the use of elaborate real-time operating systems. Their enticing ability to deal with a great number of tasks and to manage resources dynamically has not failed to show its influence on contemporary systems. The requirement to provide binding information on the system response and its ways of interaction has given way to techniques that can provide estimates at best. This is quite astonishing in light of the involved risks to humans and material values [Lio96, Risks].

Claim 5: The resource requirements and the timing behavior of the final implementation must be open to exhaustive static analysis. System performance should match the requirements, hence permit optimum resource usage.

2.1.5 Communication Requirements

On a higher level of abstraction the term communication is extremely comprehensive and includes a plethora of ways of how different entities can interact. A communications infrastructure must be capable of supporting both signalling for transactional interchanges and data propagation. Basically, all these methods are either synchronous or asynchronous. In many approaches the implementation of these interactions is abstracted away and it is assumed that some kind of agent between the various nodes in a distributed system will provide the required mechanisms (for example CORBA, which does not require the definition of task locations until run time [OMG95]). This view neglects that timing constraints may be associated with communication, with varying requirements depending on where specific connections within a layered system are situated. At each level the performance must fit the requirements of the communicating elements, which are typically more stringent the lower the level is.

On the level of implementations, the most simple case of communication is through global data, where a flag signals the receiving task that data is available. It is up to the receiver to check for any data that it may receive and to reset the flag after reading the data. Synchronous behavior of different tasks is accomplished easily, but this scheme obviously only works for communication that does not leave the common memory boundary of both tasks involved. If several nodes with their own processors and particular memory are involved, i.e.,

if basically asynchronous entities need to communicate, a communication network with appropriate protocols must be used. Networks range from wide area multimedia services to local instrumentation networks. They are related in that they demand a predictable quality of service, but differ considerably in the nature of their requirements. In a non-interactive television broadcast, for example, a high bandwidth and bounds on jitter are postulated, whereas delay is not critical. In playback applications, these criteria are the quality of service parameters that need to be controlled. If the bandwidth is not sufficient, packets are dropped as soon as the queuing capacity is exhausted. Also, the available bandwidth must exceed what is needed by the average transmission rate, so bursts can be cleared quickly. This differs from an instrumentation network that periodically transmits time-critical sensor data, where bandwidth may be marginal, but upper bounds on delay are essential.

Communication mechanisms that fulfill different needs can be classified as belonging to a generic type of communication network, two of which must provide guaranteed temporal performance (Fig. 11). At the lowest level, the *instrumentation network* connects a real-time system with its sensors and actuators. The requirements regarding latency and jitter are high, but because its bulk load consists of mostly small data packages, which are transmitted periodically, efficient and economical solutions are possible. The *real-time network* is the means of communication of individual nodes in a distributed real-time system. Here, deterministic behavior is most important, whereas demands on latency and jitter can be traded for reliability and fault tolerance. Operational robustness requires that correctable transient errors must not impair service predictability. Throughput should increase monotonically with the load and thrashing must be avoided. Finally, the *backbone* network links a real-time system with other systems [ARS91]. No hard real-time demands are imposed on these connections, and they are therefore not considered any further.

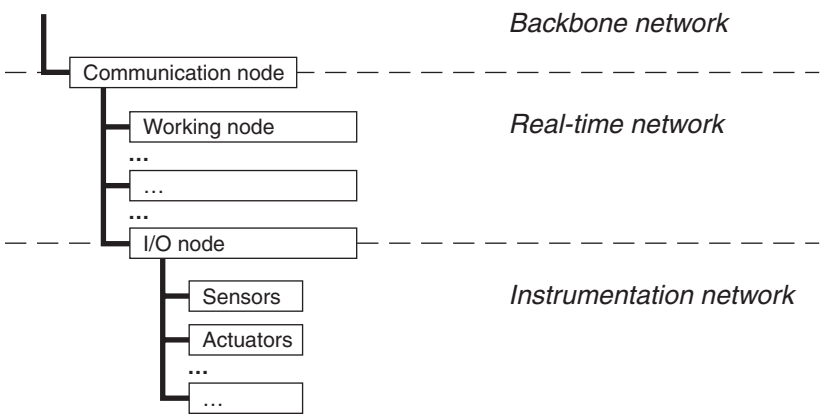

Figure 11: Embedded system communication architecture

Another classification is concerned with traffic volume and message sizes in distributed embedded systems. Some applications, for example in vision and radar systems, rely on high-volume data distribution. These messages must be dealt with alongside the more common communication traffic consisting of short, periodic messages. Networks for different applications therefore must vary significantly in their distinctive features. To choose the simplest network technology fulfilling the requirements is worthwhile not only because of cost considerations, but also because a complex protocol constrains design and development flexibility. Using available resources efficiently becomes difficult when an application must incorporate communication black boxes with a collection of services not needed for that particular installation.

The interplay of partners communicating in a distributed system may be governed by higher level protocols, but it is the media access protocol that determines a network's performance: efficiency, determinism, robustness, and flexibility. *Efficiency* denotes the relation of transmitted message content to raw network bandwidth, which is lowered by additional information in each message, and by the arbitration mechanism used to gain access to the network. Thus, for some protocols, the efficiency may vary greatly with network load, which in turn affects worst case behavior. *Deterministic* latency, or bounded and predictable message delay, can be achieved only under light load conditions with certain protocols, whereas it is inherent in others. Operational *robustness* depends on a protocol's ability to quickly detect and recover from transient errors. Furthermore, if a serious error occurs, all participating nodes must be notified with a low latency. Finally, *flexible* protocols tolerate network changes without reconfiguration when nodes are added and deleted.

Claim 6: A real-time network must be predictable and behave deterministically. Although message size and other operational properties must meet the needs of the application at hand, network performance should roughly correspond with the required minimum.

2.1.6 Reliability Requirements

Reliability requirements also embrace safety, maintainability and availability. Reliability denotes the probability that a system functions correctly according to specifications over a certain period. Safety refers to reliability with regard to critical failures that may have a harmful impact on lives and material. Usually cost or statutory regulations or both are brought in as a justification for the additional expenditures in developing safety-critical systems. Even in systems with a high production volume that seem mostly driven by cost, reliability

requirements may be high, such as in automotive applications that must operate reliably over a typical lifetime of some 3000 driving hours. Furthermore, in the long term, it may be expected that generally the tolerance for faulty programs will decline in the same manner as the tolerance for collapsing bridges had declined. Therefore, the expectations of system reliability will eventually slop over to noncritical applications, which means that the quality status of all systems must rise and that the development methods must concord with these expectations. Currently there remains a wide gap between 'normal' systems and safety-critical systems, because reliability concerns cannot be integrated well into the development process. Therefore, this additional burden is left behind, unless enforced by necessity or by regulations.

There are two aspects to reliability that tend to be confused, especially in software engineering. One denotes the probability of faults caused by physical breakdown of a device or its constituents, an area of research and practice with proven methods used in assessing the reliability of all kinds of technical systems [Bir97]. Thereby a system's failure rate is calculated on the basis of standard failure rate models of its components. In a reasonably well constructed system a single fault will not induce an unsafe condition, and the system may even be expected to recover from transient faults. These, of course, may occur any time in both hardware and software. It remains another question if software should be designed to handle all these cases explicitly, or if it may rely on exception handling built for example into a language or operating system.

The other aspect is concerned with faults that are actually introduced into technical systems through inadequacies in the process of its design and implementation. If engineers in one of the established disciplines are confronted with the latter, in all likelihood the incident concerns liabilities and guarantees, and it will attract lawyers. Unfortunately, systems that contain software are guaranteed to contain inadequacies, even errors, in their design. Therefore, in safety critical applications measures must be taken against both of these problems, and in practice faults of both kinds very often get treated the same. Various models of fault tolerant system architectures have been proposed and put to use, including redundant systems featuring cold standbys, warm standbys, replication with self-checking, replication with voting, and various levels of duplication [Sto96]. On the level of programs, a software architecture and language subset is selected that adds to the complexity of the problem at hand as little as possible. Likewise, systems can in a way be made insensitive to external events by choosing a time triggered approach. Time offsets are used to control precedence relations and to control jitter and drift, which also presents virtually the only way to achieve predictable behavior in distributed systems.

Some of these techniques, including the use of redundant hardware, are expensive, and they augment the complexity of a system. While cost considerations limit their application to the area of safety-critical systems, the same reliability requirement would call for simple solutions, which are easier to oversee, implement and maintain. Those measures necessary to compensate for the additional complexity augment costs even further. The cost-neutral measures, on the other hand, could be put to use profitably in *all* embedded systems.

Claim 7: Reliable solutions are lean solutions.

2.2 Development

Embedded systems differ from conventional data processing installations in many ways. A major difference lies in their heterogeneous architecture, consisting of microcontrollers, dedicated signal processors, memories, and custom-built analog and digital components. As described in 2.1, they are characterized by reactive behavior, real-time requirements, fault tolerance and predictability. The complex relationship between the involved hardware and software components, as well as the relationship of these components with a physical environment make development of embedded systems difficult. The admonition to 'never trust an actuator' leads to defensive strategies, which further complicate development work.

Additional problems arise from inevitable conflicts among constraints and optimizing goals, and all this is made worse by the development process itself. It is characterized by a collection of methods that very often are only badly integrated into the development flow. The limitations and restrictions inherent in these methods and the tools they are associated with impair a developer's freedom of movement even further.

Cross-development, where the development platform differs from the intended target, presents advantages with regard to available tools and network connections with other team members and data bases. These advantages are outweighed by difficulties encountered when attempting actual implementations. Because analytic methods – if considered at all – cannot master the complexity of real-world systems, this step is taken with incomplete and unproven solutions. To this are added the problems of debugging fast responsive and distributed systems in operation. Therefore, the dominant problem in the development process is the wide gap that remains between the test phase in a simulation and the actual realization on the final target. This gap has been apparent to many practitioners and is also slowly getting attention from academic researchers, as illustrated in Fig. 12 (adapted from [ABB93]).

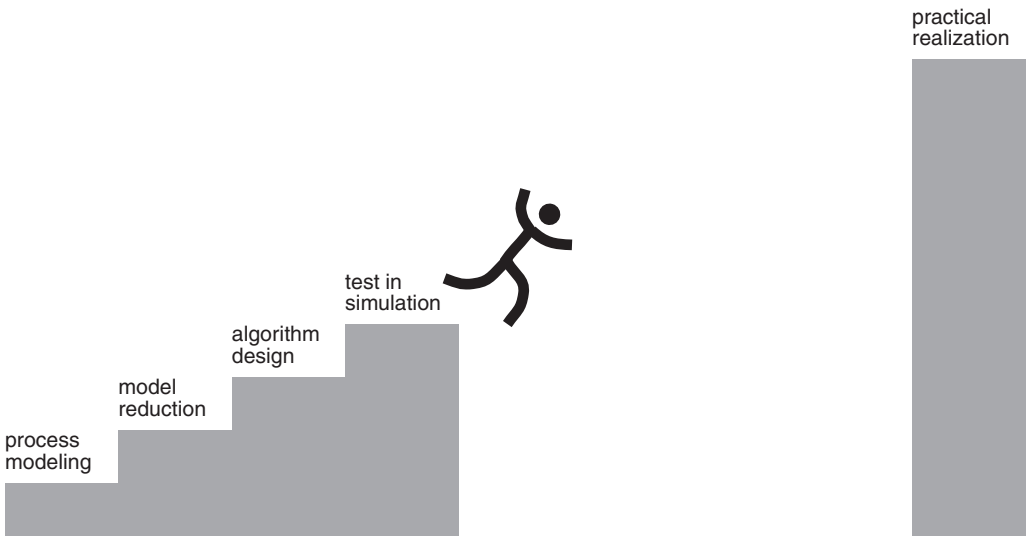

Figure 12: Gap in the development process

The gap is narrowed with the application of design principles and methods that are aimed at precisely defined areas of the entire problem. It is further narrowed when these areas overlap only minimally, but it will not disappear as long as the integration of different methods remains inadequate. Subsequently, various methods and techniques used in the development of embedded systems are briefly presented. In the same manner as has been attempted for the requirements (see section 2.1), the central points of these methods are retained.

2.2.1 Building Principles

Two building principles are central to most established engineering disciplines. These principles – the pairs decomposition ↔ composition, or top-down ↔ bottom-up – are associated with specific phases in a development process and complement each other ideally. Consider the building of a house, for example. It starts out with a planning phase, where an abstract view of the involved elements and their interplay is sought. The final product of this top-down activity are blueprints that will be used in the later construction of the building. The subsequent construction is *based* on the blueprints, but is attempted with completely different methods. It can be seen as the bottom-up composition of smaller constituents in such a way that elements with the properties and the comportment as defined in the plans are obtained, a proceeding that stands out against pure top-down approaches relying on step-wise refinement. These exist only in software engineering, since in the example of building a house they would lead to the reinvention of windows, doors, doorknobs, hinges etc. for every new house being built. Architecture and design are top-down activities in both cases, whereas the construction of the required elements is more favored

by compositional, i.e., bottom-up approaches. One reason for this is that assumptions about the top level of unknown systems should not be made at an early design stage. Also, components obtained by decomposing entire systems are not generic, which makes them hard to reuse.

It must also be noted that building principles are predetermined by the chosen development method. Developers may be forced into adopting one or the other principle exclusively, or – even worse – they may be faced with an inscrutable conglomerate of both methods. This is the case with object-oriented methods, where inheritance can be used not only to express specialization, but also generalization. Therefore locality is lost, and the use of multiple inheritance especially leads to web-like dependencies.

Essence 1: *Top-down decomposition and bottom-up composition complement each other. Both these principles are used to advantage in the respective application domains of architecture and construction.*

2.2.2 Architecture

Even if the development approaches presented in section 1.1.2 lack an overall view, there is of course something to both of them. In fact, if an attempt of tackling the problem as a whole is to be successful, both have to be considered carefully. Therefore, in a well-balanced approach these extreme views ought to be reconciled. In such an attempt, the starting point remains unaltered: loosely coupled spheres must be identified and isolated, so they can be dealt with separately throughout most of the development process. It is important to note that postulating the separation at the outset of the endeavour is not sufficient, nor is the proposal of methods with which to attack the then separated problems on their own. The key issue is to go beyond the diverse spheres and the particular methods used within a sphere, and *enforce* this isolation using a description of the components and their interrelation on an even higher level. In this binding architectural view the points of contact between components are based on replicas of the associated elements that are present on both sides of the dividing line. Explicitly specified connectors between such components further assist in the decomposition of entire systems [Sha96]. The component architecture offers insights into a system's structure on a high level and at the same time ensures compatibility of the low-level interfaces that emerge when code is generated. It is the *pivot* of constructive development effort and the basis of project consistency; therefore it must be defined as early as possible. From then on, if changes are kept well under control, mutual influence of spheres that are otherwise but weakly coupled is effectively limited.

This notion of architecture, which primarily deals with a system's operational structure, is complemented by another view, commonly also termed 'architecture'. This view is the implementation architecture that describes the hardware topology the system will be deployed on. Such a topology consists of a set of resource items including processors, busses, sensors and actuators. Components of the abstract operational architecture must eventually be mapped onto one distinct implementation architecture. Different approaches to architectures treat these two aspects with varying emphasis, some of them reasoning about the cooperation of components [AlG97], others focusing on the implementation [GSH97, Kop97]. If architectural concepts are solely based on the level of implementation, they cannot be used to contemplate component interaction; on the other hand, if they neglect implementation issues in favor of abstract interaction issues, they will not be of much use in practical development work [Mik99].

It is general industrial practice to choose the hardware environment at the outset of a project, with the disadvantage that it will be evaluated only late in the development process, namely when the entire code is ready for installation. It is much more desirable to deal with an abstract operational architecture during development and to postpone mapping this architecture onto real hardware as long as possible. Such a generic component architecture does not only permit an optimum adaptation of hardware resources to the actual needs, it will also tolerate changes in the implementation architecture due to unavailability of key hardware items, or due to a mid-life update of control systems.

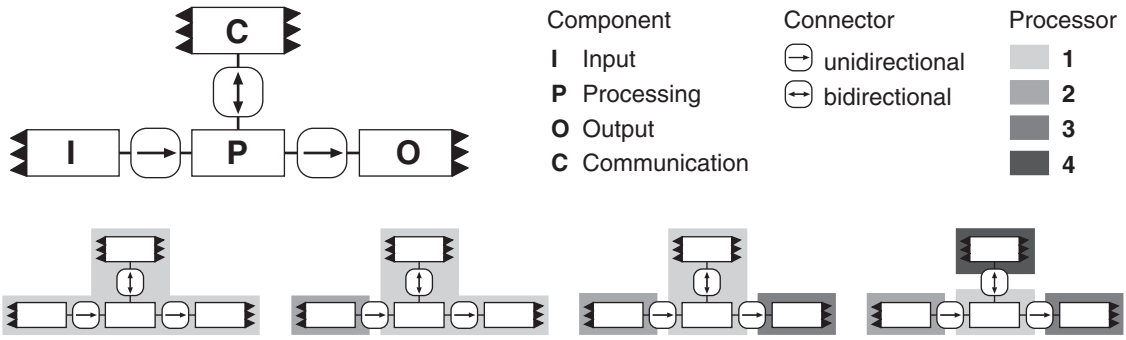

Figure 13: Component architecture mapped onto different implementation architectures

A sample of these different views is given in Fig. 13. The operational architecture of this system consists in a decomposition into four components (top left in Fig. 13). These components are mapped onto different implementation architectures with one to four processors. The example illustrates that system decomposition is only possible within the constraints of a given component architecture, i.e., mapping onto more than four nodes would require a redesign

of the component architecture. With regard to later enhancements it is therefore important to provide enough detail in the abstract component architecture already at the outset of a project.

Aside from the benefits gained regarding the separation of concerns for the actual development work, this approach permits to isolate the two only faintly related kinds of architecture: the operational structure and the implementation structure. The first view is concerned with functional entities as suggested by elements in the environment, the latter refers to a hardware topology, which can be viewed as an emerging property of the control system.

Essence 2: *The two facets of architecture (operational and implementation) must be complemented by an integrative component architecture that isolates operational from hardware issues.*

2.2.3 Language Issues

Program correctness is not a language issue. System architecture has a far larger impact than the choice of programming language, because the product life of many embedded systems is considerably longer than the period for which a particular programming language is fashionable. Moreover, there are no languages, with the few exceptions that were designed specifically for real-time purposes, which can be used without reservation in embedded systems, including even the most modern candidates (as is illustrated for example by the proposed limitations for Ada [StB94], C++ [ECS99], Oberon [Sch97], and Java [RTJ99]). Aside from those languages for which *all* currently available compilers and libraries lack conformance with the standard, such as C++ [Pla99], all this makes much of the debate about 'which programming language' fairly irrelevant. In addition, code quality still depends to a great extent on the ability and willingness of its programmers, and it is disputable if code from one language is any more reliable than that from another (see [SpG84] for an example of reliable assembly programs). As regards programming paradigms, functional and object-oriented variants both seem possible choices. Whereas code module sizes as produced by today's compilers have become almost identical, differences remain in the obtainable run-time performance. In the case of C and C++, `#define` macros are replaced by in-line functions, which introduce basic blocks and therefore prevent compiler optimizations. Since C++ performs poorly as a module-based language, all parts of a problem solution must be made objects, even those that could better be described functionally. Debugging of object-oriented programs is complicated, because symbol names are not confined to within a specific scope. Therefore, setting breakpoints and finding

global variables becomes difficult without good naming standards. Finally, details left unspecified by the language designers, together with other details that are too hard to implement, make C++ a non-portable language.

When code is generated instead of written by hand, the abstraction level of this code is a matter of choice. Much has been said in the favor of well-constructed and strongly typed languages that appear and vanish in various disguises. Generated code in one of these languages has the advantage of still being comprehensible, which may be an issue when source-level debugging is attempted (it also offers the questionable opportunity of manual modifications). The other extreme would be to make code generators produce object code directly. The first choice seems appropriate, but since many of the required assertions provided by compilers for real high level languages can be performed by the code generator, strong typing, the absence of pointers, checking of array bounds and many more are not compulsory for the generated language itself. On the other hand, the use of assembly language or even object code in code generation would incur too many portability limitations. The solution is to choose a level where both the issues of portability and 'low-levelness' are best taken care of. Currently this is the C language, which has remained the preferred language for writing drivers and many other low-level mechanism even with the advent of its object-oriented descendant. It has the required flexibility, since it counts among the low-level languages in the views of its critics [Joy96] and even of its originators [KeR88], and yet can provide some relief to programmers through abstraction when utilized cautiously. To this end, many programming teams and institutions restrict the use of certain constructs and prescribe ways of how to use others [MIS98]. While guidelines of how to use or avoid individual constructs of a language do seem inadequate as a basis for entire projects, they are appropriate if only a small part of a project's code must be written by hand. This is especially true if these parts are either at the heart of the system to provide kernel functionality to the generated code (in which case the code is in care of only a few specialist programmers), or if drivers for peripherals must be integrated (which are small self-contained components that can be analyzed and tested before incorporating them into a project).

It is a common misconception that the limitations imposed by a high-level specification or modeling language result in reduced functionality. Although it is true that the expressiveness of a general purpose language such as C is not available, these environments offer a collection of features adapted to the specific problem domain. The advantage is, that the unnecessary and confusing choice among different ways of formulating an intention is replaced by one single construct. The same goals can be achieved, with the additional benefit of relieving the developers of the plethora of constructs they must otherwise be

familiar with. When a programming language is used in the creation of something new, this knowledge is neither necessary nor present to achieve a certain goal, but it *is* required to understand what others have done. Any opportunity that leads to simpler and hence understandable solutions should be seized.

Essence 3: *Since code was invented to be understood by machines, it should be generated, not written. If it has to be written manually, only restricted language subsets must be used.*

2.2.4 Tools

Up to now, tools and tool environments in software development have never been able to satisfy the users' needs completely. This is particularly so in the realm of real-time systems, where it becomes clear that particular engineering tools support only a fraction of what its development comprises. For example, a specific tool is required to find an optimal controller structure and its parameters, another one helps describe the system's event-based reactive behavior, still another assists in the design of the input and output subsystem, which interfaces with the system's sensors and actuators. Additional tools may not even contribute directly to the later solution, nevertheless they are indispensable for tasks such as version control and project management. The examples illustrate that the whole domain can only be addressed from within a heterogeneous environment that incorporates many different tools, each of which can only be used for a part of the overall system development cycle.

Unfortunately, tools often lack a clear separation of concerns, reflecting a property of the methods they support. The two prevalent approaches are directly based on respective programming paradigms, being either functional or object-oriented. The former invariably leads to system structures that are implementation-oriented even on a seemingly high level of abstraction, whereas the latter fail to provide a complete specification of global behavior [Ham97]. It centers around classes, which is essentially an architectural view, but at the same time attempts to describe behavior with state-transition diagrams. This overlapping of problem concerns violates the envisaged orthogonality; it leads to systems that are hard to understand and even harder to change.

On the other hand, tool users in such a setup face integration problems. Large projects may involve many different teams, and not all of them will use or be able to use the same tools, and furthermore, it may be necessary to incorporate legacy models in still different formats. Even if it is acceptable to keep these models and their documentation well separated from each other, the different models will need to be united in any simulation or target implementation.

The separate tools involved in a development need to exchange data. Habitually, the potential pairs of tools exchanging data are closely related to the underlying life cycle model. Linear models, such as the waterfall model, assume that each phase produces output, which becomes the input to the next phase. If different phases roughly correspond to different tools, batch-only, unidirectional data exchange is required. More recent models (e.g., the spiral model [Boe88]) assume that similar activities must be performed repeatedly during the life cycle, sometimes using the same, sometimes other tools. The intermingling of development models with tool domains leads to overlapping capabilities of tools and the data they work on, mainly brought about by mistaking simplification for abstraction. Data exchange cannot work under the assumption that a specific tool exports all its data, which then is read by another tool. Team projects impose further requirements onto data exchange; there, it must be avoided that inconsistent changes are made by different members of the development team, and that individuals or groups work on outdated information.

Therefore, the models that tools are based on must be orthogonal (in the sense of a clear separation of concerns), independent from the development process, and integrated into an information database, where the relevant project information is readily available to all tools.

Essence 4: Tools must be integrated into a coherent development setup with well separated domains. Consistent projects are based on clearly defined and automated dependencies.

2.2.5 Modes of Operation

Other complications in the development are based on the necessity that parts of the evolving application are to run in different environments, either because of the requirements that the final installations must meet, or only for trial and testing purposes while work is in progress (Fig. 14). Individual implementations of such different modes of operation share a common functionality, hence only their connection must be adapted to specific hardware environments. This is true of all comparable development setups, but is only rarely exploited. Because implementations are often needed in different versions and simultaneously in parallel, it is easy to get lost in the jungle of versions and variations, even if at first glance they do not seem to differ much.

This is one reason for developments to evolve into completely separate lines due to initial, subtle differences. Aside from making it more probable to loose track of what is going on, these redundancies also cause undesired additional maintenance costs.

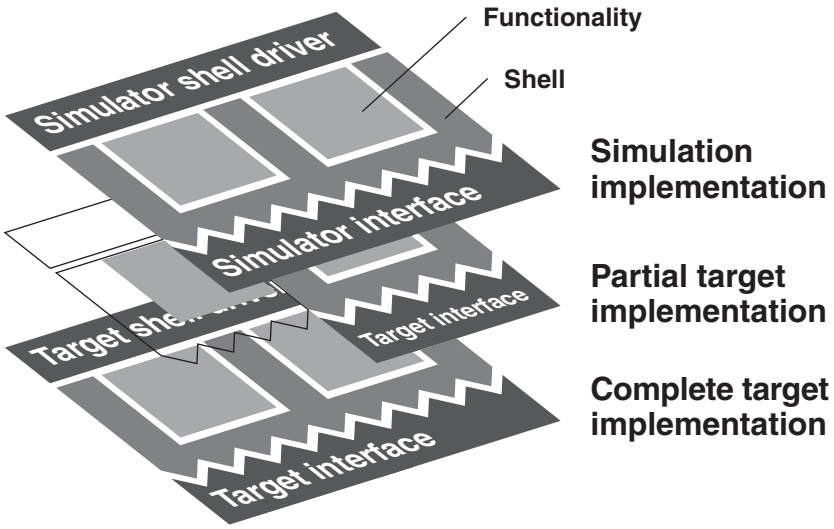

Figure 14: Modes of operation

From another point of view, modes are designed to support different run-time configurations of the whole system at different times. Examples of this can be found in flight control, where ground handling, take-off, cruise, and landing represent the different modes, as well as in communication systems that in one mode of operation service a large number of connections exclusively, or dedicate a part of their resources to administrative tasks under low load conditions in a second mode. This way of flexibly managing system behavior usually requires the explicit definition of operating modes in a dedicated language. These languages have no connection with the lower application domain, therefore an additional and difficult step is required to identify dependencies and analyze consistent configurations off-line [FeL98]. Another approach (see chapter 7), which avoids inconsistencies, extracts the various configurations from a system's functional model directly.

Essence 5: Implementations for different modes of operation that are necessary during development must be deployable from within a coherent system representation.

2.2.6 Simulation

The advantages of working with software prototypes are widely acknowledged in the development of non real-time computer systems. Their use can even be extended to those real-time systems with somewhat elastic timing requirements that permit temporary weakening during a simulation. However, if stringent timing requirements are dictated by the controlled processes in the environment, only complete, reliable code can be allowed to run on the implementa-

tion target. Therefore, simulation with the control system code acting upon a virtual machine with the same reactive behavior as the real one is of overall importance. This step offers a number of benefits in development, it

- provides the necessary insights into complex process behavior,
- eliminates the risks and possibly high cost of just running code on the actual target and observing the consequences, and permits the evaluation of various control strategies in a safe environment, prior to implementation,
- helps increase productivity, because work on the control system is allowed to progress independently from the state of the envisaged environment, which may itself still be under development,
- shortens the initial operation phase when switching from simulation to the final target and its environment, and thus helps to keep projects on schedule in their sensitive last development phase.

Simulation is a simplification of the real world, and is thus inherently an approximation, arrived at with techniques involving discrete events and system dynamics. Confidence in a model depends on the model's foundation, and also if the model captures those aspects of reality that will be decisive in the interaction with the control system. This can be assured with model verification and validation, a step that at the same time permits to test the (informal) requirements specification itself.

Essence 6: Simulation implementations are mandatory in the development of complex systems with rigorous timing requirements, which are dictated by processes in the environment.

2.2.7 Performance Verification

As in the case of functional correctness, ways are needed to predict an embedded system's performance with regard to timing requirements. Whatever hardware environment may be chosen, there are basically but two ways of ensuring the required performance when implementing embedded systems, testing and analysis. In spite of all criticism regarding efficiency and completeness, system tests and simulations still are the most widespread methods of ascertaining performance. Since it is impossible to trace and measure all paths through a complex program, worst case execution paths are sought by executing test cases. The creation of suitable test cases to be used in time measurements is a vast problem field all by itself. Recent attempts to find the longest execution times even make use of genetic algorithms [JoW98]. When a system built along these

lines is finally installed on a target, developers ensure that it is lightly loaded, and they hope that performance will be similar to when it was tested.

The other approach is based on analysis, where the behavior of a system and its environment is examined comprehensively prior to deployment to show that it will work under the expected load conditions. In modeling environments that support notions of time, this analysis is accomplished on a high level of abstraction, albeit using assumptions regarding the behavior of the processes in the environment. So, the validity of assertions gained in this way depends on the quality of the assumptions used for the analysis, a fact that had to be learned the hard way in a number of important projects (as for example the Mars Pathfinder rover [JoR97]).

A combination of the two approaches would therefore be preferable. The performance must be verified with actual execution time *measurements* of the final code. These measurements are not based on test cases, but are made according to the known worst case execution paths as defined in a functional model. This, however, is only possible when a static and deterministic solution can be given. Such a solution also permits optimum processor utilization, albeit it is paid for with a loss in flexibility.

Essence 7: *When a consistently high processor utilization is mandatory, system performance can only be asserted by examining the final code directly on the target, presupposing static and deterministic solutions.*

2.3 Deployment Techniques

The final product of development is an operational system that behaves according to the requests and constraints set forth in the requirements specification. Customarily deployment, the last step in a development, is most expensive in time and resources, because it concerns two parts: hardware and software. Development methods as presented in section 2.2 strive to narrow the remaining gap between preceding development steps and the final move onto a target. The gap from Fig. 12 can be narrowed using these techniques, but the height of the last step remains. However, it can be reduced considerably with a fitting choice of deployment techniques, some of which are presented below.

2.3.1 Deployment

Most processors can only execute one task at the time. Additionally, this task can be interrupted by a hardware mechanism. A simple enhancement to

accommodate more than one task uses an interrupt procedure to switch from one task to another, saving and restoring respective control blocks. The majority of small embedded systems currently in operation makes use of this efficient task switching model. Larger applications may need more sophisticated solutions that include message passing, means for task synchronization and schedulers that are capable of dealing with different task priorities. These services are accomplished through an entire software layer introduced between the hardware and the application. This software layer is expensive, both in terms of money (many commercial kernels are subject to royalties), processing power (the overhead introduced may be substantial) and memory (unneeded components may be dragged along if a kernel's structure is not modular).

Whatever design and implementation techniques involved, the ubiquitous input-processing-output cycle is what finally runs in embedded applications. More refined ways of structuring complex systems have led to a collection of singular tasks that replace monolithic programs. Modularity at the task level should permit system compositions involving new parts as well as legacy code. The drawbacks of these techniques in hard real-time systems are countered with more efficient hardware in the case of speed considerations, and testing in the case of unpredictability concerns. If components that were split up in the design phase are recombined into a single task for the implementation, benefits for processor utilization and response time predictability can be achieved (Fig. 15). Then, the number and types of tasks do not depend on a system's specific functionality, but represent the more generic structure common to all embedded systems.

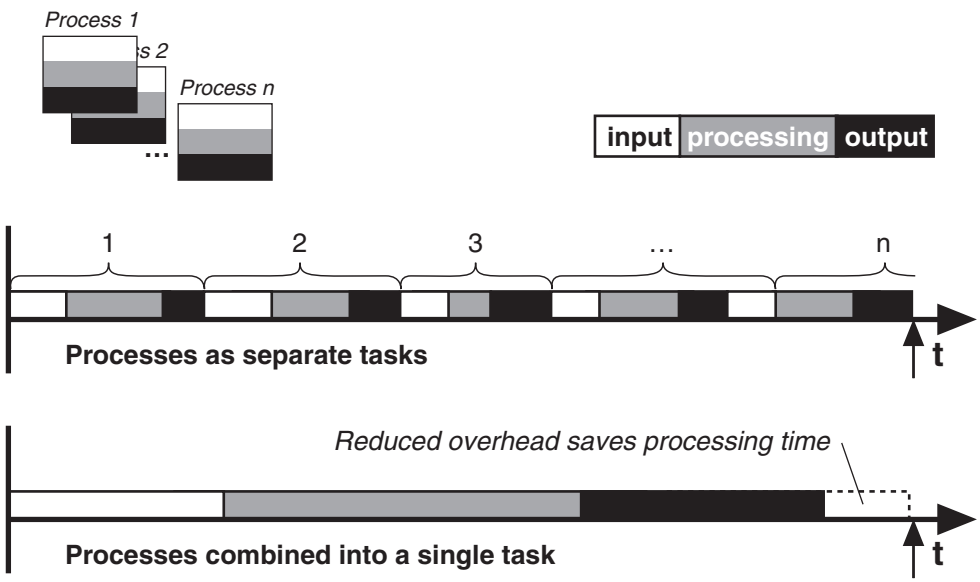

Figure 15: Implementation variants

2.3.2 Code Generation

As described in section 2.2, the most unpleasant problems appear after work on the implementation has begun, when it may be discovered that a design is either wrong, impractical or that alterations to the original intentions need to be made because of implementation constraints. If changes are made at this stage, it is very difficult to keep the design and the implementation synchronized. An obvious solution is the automatic compilation of a higher level specification into optimized code, thus uniting specification and implementation from different levels of abstraction. Code generators also do away with another development problem, namely that in cases where system behavior deviates from what was intended, the reason could be conceptual or a coding error. Code generators confine the suspects to areas that define the desired behavior.

If generated code is to replace manually written implementations altogether, the generators have to meet a number of difficult demands: small code size, fast execution, adaptation to vastly different hardware and operating environments, to just name a few. Adjustments to specific environments should be done automatically from the same specification and tests created in the design phase must remain valid also during the implementation phase.

Generators share many advantages with compilers, which are among the most reliable programs in use today. The reasons are obvious. A compiler implements a language that is generally based on a formal semantics, its structure is highly developed as a result of the history of compilers, and errors become apparent and are repaired quickly because of its widespread use in different environments. The same is true for code generators, as they are built generically according to the formal specification of a behavioral model and used in many different applications.

Another major improvement of generated over manually written code lies in the possibility to structure the code according to guidelines that are much stricter than the ones postulated by language subsets, such as presented in section 2.2.3. Thus, implementations can be obtained that are free of problematic constructs, such as recursion, loops, or dynamic memory management. While it is easier for code generators to adhere to guidelines than it is for programmers anyway, additional generated files can further assist the setting up and analysis of entire implementations before their installation. Examples are executives that help to check and calibrate hardware interfaces to the environment, or alternate representations of code flow that are needed for performance analysis. These are especially useful, since they replace the tedious search for worst case execution paths in arbitrary code with a limited number of such paths derived from a higher level model.

2.3.3 Resource Management

Embedded systems are real-time systems in so far as they can allocate existent resources to fulfill their various assignments. As long as the objectives are met, technically it is of no importance how well this allocation performs, since it is hard to calculate and measure anyway. Between the extremes of static (off-line) and dynamic solutions there exist a variety of different approaches. Success or failure also depend on a number of things, such as the relative size of the chores in contrast with the hardware capacities, the properties of the singular parts in a system, the experience of the implementors, and not least – luck. Managing resources in a real-time system is expensive, and the costs do not decrease equally fast as the hardware costs do. This explains current trends to omit optimal but expensive resource management and replace it with cheaper additional hardware, at least for products that are not destined for a mass market.

Application programs for embedded systems are often partitioned into tasks representing units of operation. If these tasks are to run on a single processing entity, the processor resources must be divided among the candidates, an activity generally termed scheduling[1]. The simple dispatchers of the past have become more complicated and often are part of sophisticated real-time operating systems. Because these usually make no assumptions about the tasks they are dealing with, schedulers are increasingly powerful and generally achieve more than what is required in particular cases. While the benefit is enhanced flexibility that may include dynamic reconfiguration of entire task assignments, there is also a price to pay in the form of increased overhead, both in terms of memory and processing time (and possible royalties per installation!). Therefore, the choice of a suitable scheduling strategy depends on a number of factors, including:

- the number of processors in a system (and if they are the same or different types),
- possible priorities of some task over others,
- the chosen task synchronization methods,
- implementation specific properties that may permit or exclude the application of specific techniques (e.g., if tasks may be preempted or not).

Scheduling Principles. Static solutions have the advantage of deterministic behavior, and the relatively simple dispatcher requires only a fraction of the entire processing power for administration at run time. A system remains stable

1. In running systems the actual resource allocation according to a schedule is performed by a dispatcher, but in the following both activities are referred to as 'scheduling'.

even in the presence of failures in the environment, namely when a primary event signaling an alarm is followed by an avalanche of correlated events. The static nature of this solution guarantees that the maximum number of external events that are dealt with is bounded. For the benefit of system stability, violations of the timing requirements of some of these events must be accepted, still a preferable situation over a system that stalls completely. A more important limitation is the concept's relative inflexibility, which precludes alterations and maintenance of systems while they are up and running.

Dynamic methods postpone the explicit decision about the order of running tasks until run time, permitting dynamic adaptations to changing behavior in the environment. It is also possible to replace parts or even add new parts in running systems. This approach leads to systems with non-deterministic behavior, i.e., only (sometimes prolonged) operation will show, if the behavior of the entire system conforms to the specifications. Another complication results from sharing resources among tasks with different priorities. If no further precautions are taken, priority inversion can occasionally change priorities to the opposite. The (with regard to static dispatching) more expensive task management, together with additional measures, such as priority inheritance, are the reasons why dynamic methods need a substantial share of processing power. With non-deterministic schedules, overload protection is difficult and adds additional overhead. For all these reasons smooth operation is only possible when the behavior of processes in the environment the was assumed during development remains valid throughout a system's lifetime.

If a system accommodates sporadic tasks, the minimum time between the occurrence of two events of the same kind must be known in advance, or else it will not be possible to predict this system's performance under heavy load. This is a property of the environment, however, which may or may not hold in all circumstances at run time. Therefore, the input system must detect violations of this condition and protect the rest of the system from possible overloads and ensuing violations of other timing constraints [Kop97].

Even if the list of scheduling algorithms is long, basically they are all variants of one of three basic solutions:

- dynamic deadline scheduling,
- rate monotonic scheduling,
- round robin scheduling.

The first two methods are based on dynamic task management, which is necessarily also preemptive. Tasks in such a system must therefore tolerate interrup-

tions by other tasks or the operating system. Round robin scheduling, on the other hand, is non-preemptive if all tasks are allowed to run to completion.

Scheduling problems have been treated extensively in the literature; for some time, scheduling even appeared to be the central question of embedded systems. Because the solutions emerged isolated from other issues, the resulting theories make no assumptions (aside from precedence and exclusion relations, and deadlines) about their objects, the tasks in a system and their comportment. If such assumptions can be made, because the comportment of tasks fits certain guidelines, the problem is defused and simpler solutions become feasible.

Dynamic Deadline Scheduling. Scheduling algorithms deal with priorities, whereas the real objective is meeting deadlines. Requirements are stated in the form of deadlines for specific tasks. Because there is no easy way of implementing a deadline driven scheduling system directly, all commercially available real-time operating systems work with task priorities. This is a radically different concept, and so the mapping of deadlines to priorities in a complex system remains a chore that requires expertise and care. Besides, because changes in the task allocation may disrupt a hitherto working solution, the mapping must be revised whenever the allocation changes.

Rate Monotonic Scheduling. This method achieves optimum dynamic scheduling of tasks with fixed priorities. The algorithm uses a distinct priority level for each sampling frequency present in the system, specifically: the shorter the task period, the higher its priority. All processes in the system are periodic, have deadlines at the end of their periods, and are totally independent of one another. This optimum fixed priority policy produces a feasible schedule of a set of tasks, if any other algorithm can do so [LiL73]. If sporadic tasks are present, a system may result where some sporadic tasks do not make their deadlines (assuming that periodic tasks have priority over sporadic tasks). The question of which sporadic task will miss the deadline is not answered by priorities alone. And, as with all dynamic methods, there remains the insecurity of the real environment deviating from the assumptions made for the analysis. Processor utilization is impaired as well, because some 30% slack must be allowed for uncertainties, even if the assumptions hold.

Round Robin Scheduling. This method is tailored to systems with tasks all running at the same priority. This either implies that the sequence in which tasks are dispatched is non-critical, that it remains stable (as for example in the typical input-processing-output loop of embedded systems), or that possible variations are statically defined. Except in its time-slicing variant, which is not suited for embedded systems, the method is not preemptive; tasks run to com-

pletion. Because the schedule is determined statically before deployment, the required run-time support is minimal and easily implementable. A major benefit is the ability to analyze the final implementation before its installation on the target.

Systems that contain tasks with different priorities cannot be realized using this method. However, the simple and efficient implementation makes it the ideal candidate for smaller systems. In such a solution, priorities on the task level are eliminated entirely and the respective mechanisms transferred to the functional level of individual tasks. This way all timing requirements can still be fulfilled and system performance is assured by simply verifying that the available processing time is sufficient to execute all tasks in sequence.

2.3.4 Communication

Instrumentation networks are built using one of a number of field bus standards with envisaged throughput rates of some Mbit/s (for example CAN, Profibus, Interbus-S), or with proprietary buses that achieve considerably higher rates (such as 120 Mbit/s on FAST-bus [Bac99]). These networks replace the previously present wiring harness that connected input and output nodes with the respective sensors and actuators. Instead of a star topology, there now is a ring or a bus. Physically, the many individual wires may be eliminated from an installation, but the ordering of the signals as present in the software remains unchanged. Because this intricate point-to-point relationship of sensors and actuators with the reactive structure of processing nodes is known a priori and does not change easily, the amount of traffic on the network can be bounded using implicit flow control strategies.

The interplay of partners communicating in a distributed system may be governed by higher level protocols, but it is the media access protocol that determines a network's performance: efficiency, determinism, robustness, and flexibility. The *efficiency* of a protocol denotes the relation of transmitted message content to raw network bandwidth. Overhead is caused by additional information added to each message, and from the arbitration mechanism used to gain access to the network. Thus, for some protocols, the efficiency may vary greatly with network load, which in turn affects worst case behavior. *Deterministic latency,* or bounded and predictable message delay, can be achieved only under light load conditions with certain protocols, whereas it is inherent in others. Operational *robustness* depends on a protocol's ability to quickly detect and recover from errors, while a *flexible* protocol tolerates network changes without reconfiguration when nodes are added and deleted.

Embedded real-time networks require high efficiency, deterministic latency with known worst case response time, operational robustness, configuration flexibility, and low cost per node. These properties can be enhanced with priorities either applied to each node or based on the message types. Message priorities are favored for transactions that are sporadic and time-critical at the same time, such as exception handling.

All communication standards are based on the layering principle. Hardware and software components are layered to form a protocol stack. Equivalent layers exist on each device and the two application programs that want to communicate do so via their local stack. A stack contains a physical layer that defines data rates, the encoding of individual bits, and the way that bits are combined into frames; a data link layer that provides addressing, error detection, and retransmission capabilities to ensure the reliable delivery of all packets; and a link management layer that enables multiple simultaneous conversations between the two connected systems. Some selection criteria are listed below (from [UpK94]):

- polling, TDMA, and connection-based protocols are simple, but may not provide sufficient flexibility for advanced systems,
- token-based protocols are predictable, but can have high overhead and require complex software to maintain robustness; if priorities are included, certain guarantees for worst case access can be given,
- binary count-down protocols rely heavily on the bit dominance characteristics of the physical medium,
- separate critical traffic from normal traffic as much as possible,
- CSMA/CD is not suited for hard real-time systems with heavy traffic,
- if the possibility of network failure has to be accounted for, redundant data paths and end-to-end protocols that expect and tolerate channel failures must be provided,
- jitter and bandwidth need to be controlled to support quality of service for playback applications; enough bandwidth must be available for the average transmit rate plus a little extra so bursts can be cleared quickly.

3 Problem Decomposition

Essentially, the chores of embedded control systems are simple. Complexity comes into play only with the development process, when a basic activity as shown in Fig. 16 is combined with other such activities, when higher levels of abstraction are needed to deal with behavioral issues, and when actual implementations must be attempted. Of course, the reason for combining several activities with an analogous structure, and thus the main reason for complexity in embedded systems is that the environment itself is complex. Likewise, understanding this complexity without the help of abstract representations would be impossible. But, and above all, the complexity in a system is greatly accentuated by mingling development issues that should remain separate.

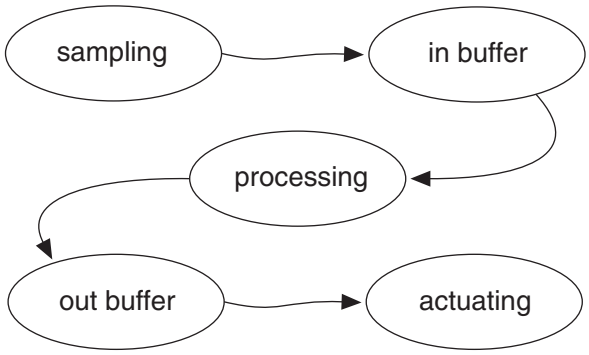

Figure 16: Basic activities in control applications

Just like the sequence of basic activities from Fig. 16, all the other issues that must be dealt with in a development are simple in their own right, such as the problems of architecture, data input and output, communication, functional behavior, implementation, and development phases. Because it is the combination of all these different problems and their solutions, while keeping up the claims made in chapter 2 that causes additional complexity, a bird's-eye view on development is needed to clarify dependencies. Three practically orthogonal

aspects of decomposition can be discerned. Ideally, variations along one of the dimensions leave the two others untouched (Fig. 17):

- *Architectures:* the operational, implementation and component architectures that are needed to associate different views of a system during development. Their coherence is accomplished through part-of relations.
- *Concerns:* the separation of a system's functionality from the connection to its environment, which is achieved through the definition of rigidly isolated problem domains.
- *Abstraction:* the representation of functional models and connective specifications, as well as the source and object code representations. Coherence of representations on different levels is achieved through generators that link the more abstract specification layers with the implementation.

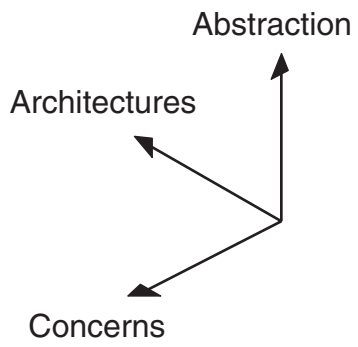

Figure 17: Aspects of decomposition

3.1 Architectures

Architecture sums up all the different structural considerations. One such view is imposed by the disposition of elements or element groups in the environment, termed the *operational architecture,* typically a 'boxes-with-lines' description. This is the view everyone adopts in their reasoning about system behavior – everyone but the control system engineers. For them, even when they use seemingly abstract approaches, the *implementation architecture* lies at the heart of every system. Unfortunately, any development based on a specific and fixed implementation architecture leads to rigid concepts that are hard to adapt to changing environments. Other methods, which endeavor to arrive at an implementation by refining an abstract top layer, are affected by this conception in a different way. Because of the direct association of operational elements with those in an implementation, changes on the top layer influence the implementation of otherwise independent elements, possibly disrupting a previously working solution.

The operational view is a premise from outside the field of control engineering. Its use as a basis for one notion of architecture is not only motivated by the need for a 'common language' among all participating parties, but also by the frequently required independent development of some of the entities in the environment. In these cases the separation, which also affects the control system, will be made according to the division in the operational view.

Implementation architecture is the hardware topology a system is finally deployed on. Ideally, it will be selected according to the emerging needs towards the end of a development process, but in reality the hardware is fixed just as often from the outset of a project, as a result of customer requirements or company policy. Nevertheless, hardware independence does not only permit to postpone the actual selection of number and types of hardware items, it would also facilitate later migration to different setups.

Therefore, these basically unrelated views, dictated by circumstances outside of the software development process must be complemented by a mediating third description. This *component architecture* is the *pivot* that incorporates aspects of both views and provides the much needed independence in development.

3.2 Separation of Concerns

The urge to start coding as soon as possible is deeply rooted in all breeds of software development. Quite in contrast to established engineering disciplines (which deserve the designation) a solution fragment can serve as a viable product, i.e., a program made to run with little expenditure. Another discord is the lack of established design methods in software engineering, which permit abstract concepts to depart easily from what is practicable or implementable. In a field where the correspondence of concepts with their realization is insufficient, the distinct phases of planning and implementation become intertwined, also because often developers need to experiment with their envisaged solutions. All this has shifted the focus in software development entirely away from the problems themselves and onto the methods used for solving them. The result is that the initial problem is nowhere formulated explicitly. This approach may be sufficient if the problems are well known and innovation lies solely in devising new solutions to old problems, although this does hardly apply to software engineering with its vast and growing areas of application.

This attitude is not confined to 'hacking', it has also crept into many methods explicitly used for problem analysis (such as object-oriented methods, where objects that were defined after those in the environment at the outset gradually

change to become parts of the solution). The magnetic attraction of solution-orientation can only be avoided with a more general view that is basically concerned with the world and not with programming. The problem that must be solved is located in the world, not in the software. Software (together with the computing hardware) is the solution. The phenomena of the world cannot be described with a programming language, which comes into its own only after the controllable phenomena have emerged from the problem description. *Problem frames* as proposed by Jackson [Jac99a] provide such a view.

A simple control problem reduced to its principle parts is shown in Fig. 18. It consists of the machine to be built *(Control Machine),* the part of the world that interacts with the machine *(Controlled Domain),* and the desired properties, i.e., properties that the domain does not possess intrinsically, but that must be brought about by the machine *(Required Behavior).* In the figure, the solid line represents an interface of shared controllable phenomena; input phenomena *CD:I* are controlled by the controlled domain, output phenomena *CM:O* by the control machine. The dotted ellipse and arrow express the optative requirements in terms of controllable phenomena *C.*

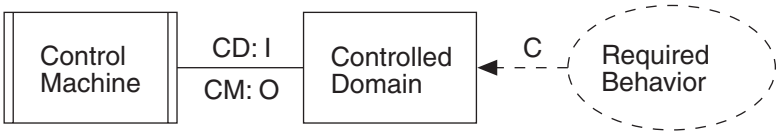

Figure 18: Simple control problem

Problem frames do not just aid in understanding the complex relationship of a system's principle parts. The decomposition of the world into distinct problem frames corresponds to isolating specific aspects of the entire task of building a control machine. This isolation permits the separate and partial development within a limited and clearly defined scope. Also, the expressiveness of methods associated with tightly constrained problem frames can be deliberately limited to the minimum needed to attain a solution. If solutions to a partial view are specified on a higher level of abstraction instead of being programmed directly, a limited scope provides further advantages. The abstract model that is created must match the problem that a specific frame is concerned with. Because the purpose of a model defines the abstraction rules, narrowing the scope results in simpler and more manageable abstractions.

A general adaptation to control applications leads to greater differentiation of both the controlled domain and its control machine [Fie00b]. Sensors and actuators *SA* are objects in the real world, albeit with a special role regarding a control problem (Fig. 19). Specifically, it depends on the availability of a cer-

tain sensor/actuator sharing phenomena with the controlled process *CP* through physical interaction *H,* if the required behavior is attainable at all. On the other hand, it is also mandatory that the reactive behavior of the embedded machine *EM* can be devised according to the requirements *C* without considering the practical realization of connecting with the controlled processes. After all, it is no accident that in all sample problems this connection is replaced by data vectors that mimic real input and output.

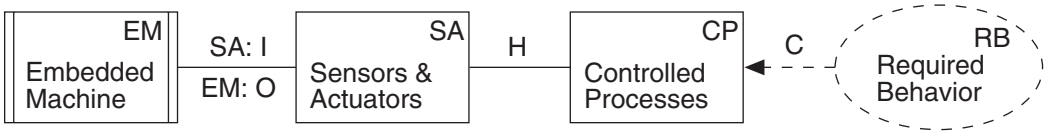

Figure 19: Embedded system problem frame

The general embedded system problem frame is still a composite frame. It would be hard, if not infeasible, to capture all its properties in a uniform description. Accordingly, the composite frame is expanded into a collection of subproblems, each characterized by an elementary frame with its own appropriate description (Fig. 20). Aside from the two basic frames representing the functional and the connection problems, others are introduced merely as mediators between subproblems (italics in Fig. 20).

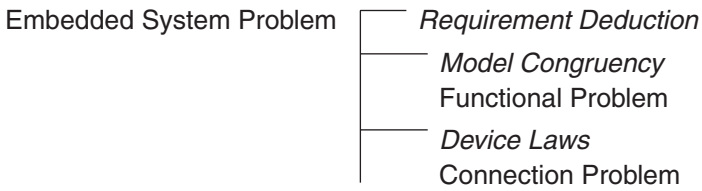

Figure 20: Problem frames used in the decomposition

The original requirement description *RB* is stated in terms of phenomena that can be observed in the controlled processes *CP,* consisting of the desired sequences of process states and events. At least partly, these may be stipulated in terms of phenomena that the embedded machine *EM* cannot control directly, and sometimes not even measure through the sensors provided. Therefore, the requirement description must be refined, preferably in terms of phenomena that *can* be shared with the embedded machine (Fig. 21). The original requirement *RB* appears as an indicative description (solid ellipse), expressed in terms of phenomena *C* that may or may not be shared with the embedded machine. It is replaced by functional requirements *FR* in terms of events and actions, so, that *RB(C) = FR(EA).* In a development, this frame serves the purpose of identifying the necessary set of events and actions that can then be used in achieving the objective, i.e., the required behavior *RB.*

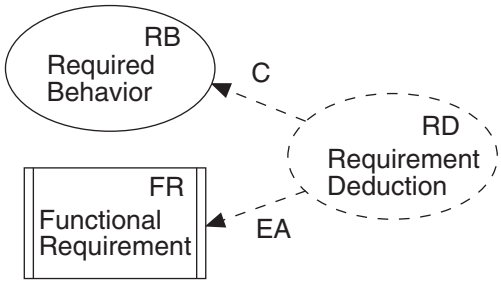

Figure 21: Requirement deduction problem frame

On this foundation, the three basic activities input, processing and output can be further refined, which leads to two composite problem frames, each with an additional auxiliary frame. This separation highlights the objective of the possible dedicated models and tools for each of the two basic problem concerns:

- *Functional problem frame:* correct reaction to events in the environment.
- *Connection problem frame:* information transport between this environment and the control system.

3.2.1 Functional Problem Frame

The functional problem frame deals exclusively with the functionality of the control system (Fig. 22). The machine to be built is the functional machine *FM,* whose purpose is to interact with the controlled processes *CP* in such a way as to fulfill the required behavior *RB* in its guise *FR.* Since *FM* does not share any phenomena directly with the controlled processes *CP,* the model *CPM* of the controlled processes is introduced. This model substitutes the direct connection by a notification scheme, whereby the occurrence of events *E* within *CP* is signalled to *FM* as messages *E'* (likewise *A'* and *A* for actions in the other direction).

The solution for this problem frame is attempted by defining a behavioral relationship of events *E* and actions *A* for the functional machine *FM,* according to the required functional behavior *FR* (as deduced from *RB* and expressed in terms of events and actions). The appropriate abstraction for this frame is to leave out any details not pertaining to the relation between the course of events in the environment and the inner states of a model.

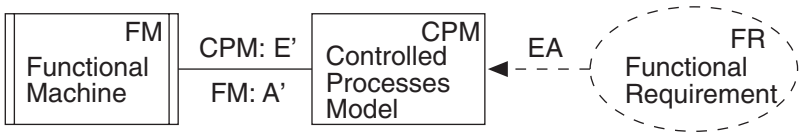

Figure 22: Functional problem frame

An auxiliary frame is needed to establish the congruity between the controlled processes *CP* and the model *CPM* (Fig. 23). Essential phenomena events *E* and actions *A,* as called for by *FR* in the requirement deduction frame, are mapped onto respective messages *E'* and *A'.* By clearly stating the difference between the real events *E,* occurring in the domain of the controlled processes, and their notification *E'* to the functional machine, problems caused by an inevitably asynchronous connection are made explicit. Obviously, these must be dealt with by the designers of the functional machine.

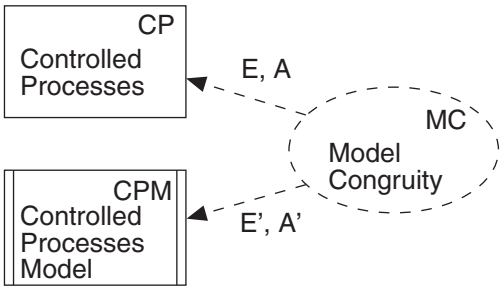

Figure 23: Model congruity problem frame

The functional problem frame is usually the most complex and at the same time the one where alterations are to be expected until late in a development. It is vital that models within this frame remain unaltered during structural adaptation, such as operation mode transitions (see section 2.2.5).

3.2.2 Connection Problem Frame

In this problem frame (Fig. 24), the machine to be built is the connection machine *CM,* operating on domains *FM* and *SA.* Just as is the case for the functional problem frame, the connection requirement *CR* defines the abstract relationship of domains in terms that in reality are not directly shared by adjacent domains. As in the functional problem frame, events *E* and actions *A* are postulated by the requirement. Events *E* (caused by the dynamics of *CP)* must be related to *E'* (the corresponding message that makes this event known to *FM),* and action messages *A'* (caused by *FM)* relate with *A* (that influence *CP).* The relation is again based on the model congruity frame from Fig. 23.

The requirement *CR* with regard to the interface to sensors and actuators *SA* is expressed in terms of inputs *I* and outputs *O.* These are indeed phenomena the connection machine *CM* shares with the sensors and actuators. But, inputs and outputs neither appear in the original description of the required behavior *RB,* nor in the deduced functional requirements *FR.* This relation is established in the auxiliary instrumentation problem frame (Fig. 25).

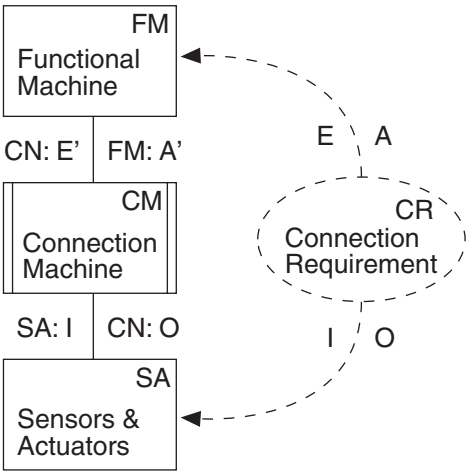

Figure 24: Connection problem frame

The relation is determined by physical interaction *H* and gives rise to device laws *DL.* Since the device laws depend on unchangeable properties of the controlled domain (consisting of sub-domains *CP* and *SA),* they are given, which makes *DL* an indicative description. It specifies how events *E* and actions *A* relate to inputs *I* and outputs *O,* respectively.

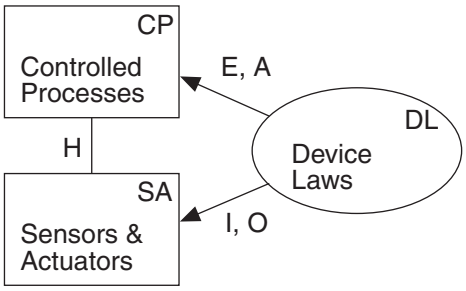

Figure 25: Instrumentation problem frame

Assuming a 1:1 relation of events *E* and actions *A* with their respective messages *E'* and *A'* (established in the model congruity frame), the connection problem achieves just this transformation. Although the transformation is realized through inputs and outputs, this fact remains hidden from the functional machine, which is the main purpose of the explicitly proclaimed connection (Fig. 26). Therefore, the connection requirement *CR* is directly based on the device laws, in that $CR = DL^{-1}$.

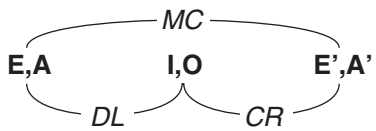

Figure 26: Connection transformations

The connection problem frame's functionality is basically implementation-oriented with a regular structure, so that a useful abstraction can be established on the grounds of information transport in a generalized form. The connection problem frame unites input, the extraction of events contained in the new data, and the output.

3.2.3 Classification

The interfaces of domains are classified as being active, inert reactive, or both. Inert reactive domains initiate no events, but merely react to events initiated by a connected domain. The reaction affects some other shared phenomenon, but afterwards the domain returns to an inert state until a fresh shared event occurs. For the simple control problem from Fig. 18 the classification is congenital: the control machine reacts to events emanating from the controlled domain. As long as the controlled domain is inactive, no reaction, and hence no processing in the control machine, is needed.

This view is preserved in the functional machine, even including its solution, where it can be implemented as a reactive block. The same is true for the connection machine, as long as it is concerned with transforming action messages originating in the functional machine to corresponding actuator outputs. They can be considered as the continuation of previous functional processing, i.e., in this respect the connection machine is also inert reactive. However, there is a discord with regard to events and event messages. While the conception of events (via inputs through sensors) from the controlled domain driving the connection is convincing in the problem frame, realizations along these lines would be difficult. They are possible, if the controlled domain and the sensors share *all* events. In this case, the hardware interface device could drive the connection actively (for example by raising an interrupt). Nonetheless, if continuous input values contain 'events', these events remain hidden from sensors and input devices. Accordingly, interrupts cannot be generated at the interface and the events must be obtained later by comparison against some threshold.

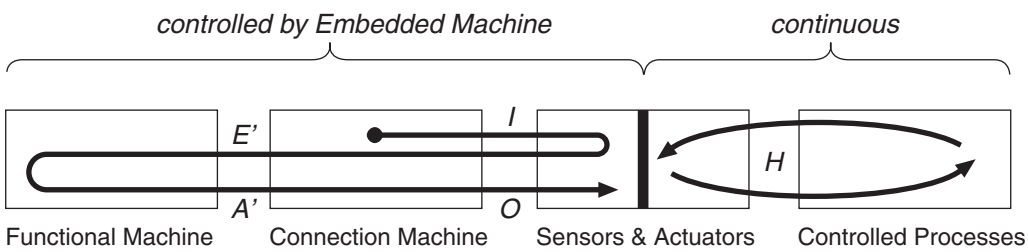

Figure 27: Domain classification

Therefore, as far as the realization is concerned, the connection machine is declared the (only) active part in the system (Fig. 27). It induces inputs, detects the occurrence of events, and calls the reactive functional machine with the appropriate event message. It then relays action messages, the functional machine's response, as output to the actuators.

Based on these precepts problem-oriented development methods with separate solutions for the functionality and the connections to the physical processes can be built. Because the interfaces between the problem frames are explicitly modeled and less prone to frequent changes than the frames themselves, development may progress independently. The separation remains intact even after system deployment, a considerable benefit for later maintenance.

3.3 Abstraction

'Abstraction facilitates the understanding of complex realities, because it emphasizes an outline and omits details that are not relevant on a certain level. An abstraction of a set of objects is a description that applies equally well to any of them. Each object is an instance of the abstraction, and there exists a one to many relationship between the abstraction and the objects it describes' (D. L. Parnas in [HPP81]).

Abstractions are *models* of reality. Useful models are accurate, simpler than reality, and understandable. To arrive at these goals, the purpose of the model must be well defined, and assumptions made during modeling clearly stated. The validity of conclusions derived from a model is limited by the assumption coverage [Kop97]. Three spheres of activity can be discerned that correspond to levels of abstraction (Fig. 28). Scopes for architectural and constructive models were given in the previous sections, whereas the final program code (an abstraction from native machine instructions) is based on the definition of programming languages and the ways they are implemented on specific processors.

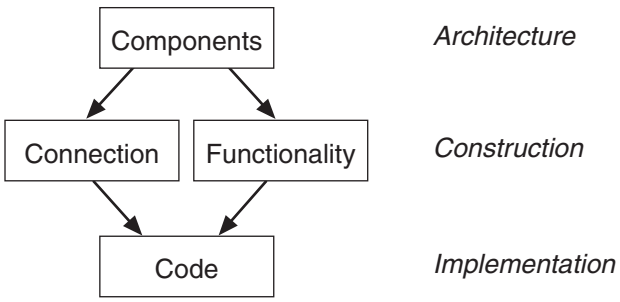

Figure 28: Spheres of activity

More subtle categories of abstraction are required within the spheres of activity themselves, where developers work on many different levels of abstraction. Each level has its own set of description methods and tools, and it is very challenging to maintain coherence throughout an entire project, practically still accomplished with just decisiveness and discipline. Often these problems are aggravated by wanting concepts that mistake omitting information for abstraction (resulting in oversimplification). Valid abstract concepts according to the introductory definition produce a number of views that engage one another in a formal way. If these relations are supported by some automatic mechanism, preserving project consistency is greatly facilitated. Dependencies are made obvious, and the number of documents that needs maintenance is also reduced, since some of them can be produced automatically from others. Even when this is not possible, the consistency of descriptions that depend on some higher level of abstraction can be checked automatically.

The component architecture must contain all the information of both the operational and implementation architectures. Their representation is informal, or only partly formal, in contrast to the formal description of the component architecture. Therefore, conformance of these descriptions is subject to manual examination. In the case of the operational architecture this is not a problem, considering the description's high level of abstraction and its orientation towards elements in the environment. In the case of the implementation, the most detailed version of the possibly many different anticipated implementation architectures must be covered.

The component architecture serves as a foundation in the elaboration of both the functional model and the connection specification. System construction means putting elements defined in the component architecture in concrete terms. The solutions must conform to the interface requirements postulated in the architectural description. Not only is it possible to automatically check this conformance, some element properties can also be directly imported. A system's functional model is divided into clusters, then built using the given component arrangement, event and action messages, and possibly data and data types. In the connection specification the same definitions are used.

Specifications and models are associated with the implementation through code generators. Even if the development methods and teams are unrelated, the methods' common foundation in the component architecture results in integral implementations with the required functional and performance properties. The last step, from source to object code, is of course automated as well. It is conventionally performed by appropriate compilers and linkers.

3.4 Practical Development Suites

The three orthogonal aspects presented in this chapter can serve as a foundation of comprehensive tool suites. It is not sufficient to attempt development within the scopes laid out by problem frames, because they are not independent after all; the braces of an architectural description are needed to relate the different concerns during development. It is not sufficient to describe a system's architecture using components with arbitrary characteristics; components must have specific implementation properties to get a perfect whole when the various pieces are combined (Fig. 28). It is not sufficient to relate behavioral descriptions with *any* code that implements this behavior; the code must do so in a well defined way. To this are added the claims made in chapter 3, which all together might compromise the usefulness of such a tool suite for any practical application.

The tool suite introduced in the following chapters lives up to the numerous confinements. An unsuspecting observer might shrink from a development process crammed with even more obstacles than there are to begin with, whereas in fact the gross limitations of the design space turn out to be beneficial in finding lean and adapted problem solutions.

4 Architecture

Complex systems are built as a collection of a number of different more or less independent blocks. It is still common practice to define their interrelation only implicitly in the interfaces of the active components. As a result, the system structure is hidden, and the generic character of components is compromised by embedded references to other components. A clearer picture of system architecture as a framework for satisfying requirements requires concepts that treat not only *components,* but also the *connectors*[1] between these components as distinct entities [Sha96]. Components are used to organize the computation and data into parts with a semantics and well-defined behaviors. The interaction among the components is represented by connectors. They mediate the communication and coordination activities among components. The description of software architecture with appropriate languages and tools has begun to emerge as a discipline apart from algorithms, data structures and other implementation details. Because its focus is on the configuration of components and connectors, it represents a top-down view of a system. It replaces the now common informal 'boxes-with-lines' description, but also satisfies developers who need a detailed connection and interoperation model of the system that they have to build. Architectural views also provide the basis for a system description in terms of the system's constraints, which most often derive from the requirements. Numerous architecture description languages have been developed that not only cover syntactic aspects of components and connectors, but also permit to perform dependency and consistency analysis by incorporating behavior as formal annotations [AlG97, GaS94, Luc96]. For analysis, formal descriptions of the components and connectors are used to reason about system behavior. In the construction of a working system, functional models (the foundation of any implementation) are associated with the com-

1. cf. architectural element *connector* vs. *connection problem.*

ponents of the architecture. Systems developed in this way are based on an explicit structural skeleton, which helps to maintain consistency among the various elaborated models. Finally, generic and well-defined architectural components are also a prerequisite for their reuse.

Architectural descriptions along these lines are commonly seen as a representation of system resources and connections on a high level of abstraction, explicitly in support of software development. Other areas of engineering use the term architecture in a much broader sense, namely as the representation of knowledge shared between a diverse team of experts from different fields. Adopting this view, an architectural description must embrace all system characteristics needed to make it the primary representation not only as a technical basis for software engineers, but also as a communication vehicle for project managers, hardware engineers and domain experts. New points of view are thus introduced that must be matched in the description by adequate representations of different aspects. The semantics of these representations permits an association of architectural elements with specific strands in the ensuing development process. But, considering the variety of its users, cognitive qualities are also important, a claim that leads to representations closely matching meaningful properties of the application domain. The degree of complexity of a representation depends on the complexity of the controlled processes in the environment, but is made worse by variations introduced when the same particulars are described differently by different people. This type of complexity is effectively controlled by limiting the expressiveness of the description language to what is needed [AnG99].

Of the three architectural descriptions introduced in section 3.1, two correspond to views from different fields, which are then integrated into a third view. Information from both the *operational architecture,* reflecting system structure as seen by domain experts, and the *implementation architecture,* which embraces all conceivable hardware topologies, blends into the *component architecture.* The foundation of the component architecture forms the *pivot* for all subsequent constructive development work.

4.1 Operational Architecture

The 'architectural' views of domain (i.e., mechanical, electrical, chemical etc.) engineers are typically informal 'boxes-with-lines' descriptions. In spite of this, and because domain engineers think in terms of the application as a whole, their point of view is on a high level of abstraction. For a number of reasons, their way of partitioning a system must be replicated in the software structure:

- it is the basis for cost considerations and process management,
- the cooperation across different professional fields is facilitated with a notation that is generally understood and accepted,
- strategic decisions in a development are made in terms of components as identified by domain experts.

Examples are the integration of legacy components, preferably including existing control elements, or the need for independent development by different parties. Having anticipated the separations along precisely defined interfaces does not only save time and expenses, it also helps preserve the integrity of entire concepts.

The possible variety of 'boxes-with-lines' descriptions as an abstraction of components and their interconnection may seem immense (not to mention the semantics inferred by the various designers). However, the object is abstracting from real machines and not from software and its mapping onto a hardware topology. Thus, the generic techniques that are applied in building machinery help confine the variety of architectural structures. For example, components in a drive train are arranged according to their physical connection, which is linear in general; components of a production line are related in harmony with the flow of workpieces, etc. Both these examples suggest linear topologies also for their control systems, which may not be the preferred solution from the point of view of an installation's control system. With additional components (which do not have a counterpart in the environment), hierarchy can be introduced into the structure, and the interfaces made simpler by lowering the number of interaction points. In the exemplary operational architecture for a small robot (Fig. 29), the single additional component *Head* permits the realization of a star topology. Because the nature of interactions between the basic components *Motion, Handling, Navigation* and *Perception* are not well defined at this stage, the otherwise necessary mesh interconnection is replaced by a single connection to each basic component. The advantages are that the interfaces remain stable even if more components are added, and that the details of interaction among components can be resolved later, when *Head* is refined.

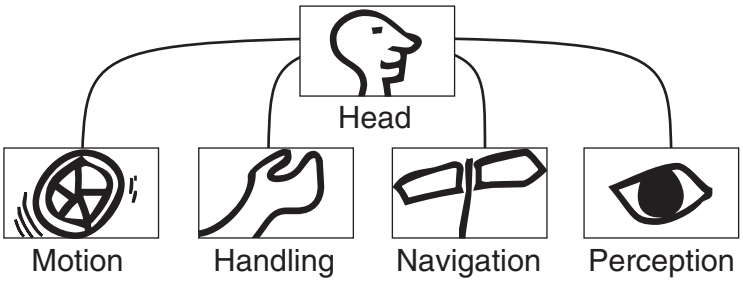

Figure 29: Robot: operational architecture

The usefulness of obviously informal descriptions for the actual realization is not readily perceived. Its main purpose in the development process is that of a mediator between domain experts and control system engineers. Advantages for control system development result from using diagrams that designers are going to draw anyway and link them to a formal specification. From there, the interdependence of the various representations in a system can be managed by tools. This is a prerequisite for tool-assisted configuration management, which can then replace otherwise laborious manual procedures.

4.2 Implementation Architecture

Implementation architectures describe the configuration of a system's hardware resources and their interconnections. Customarily, this is the architectural description 'per se' in control system development.

Domain experts very often neglect this view and, on the other hand, hardware engineers are usually not known for reducing the burdens of software developers; rather, they are driven by technical constraints and aspects of resource usage. The complexity of implementations would, however, be reduced, if operational and implementation architectures were brought into line. Mutual dependencies make this a difficult undertaking, which usually is only possible if resource requirements of the various operational elements are known. Because these are an emerging property of the development process itself, it is wise to consider a *collection* of hardware topologies instead of sticking to the single one held appropriate at the outset of system construction.

In the case of the example of Fig. 29, the collection might include – aside from the trivial case with only one processing entity – a topology where peripheral access and actual processing are each assigned to separate processors (Fig. 30), or a truly distributed implementation with a separate processor for every major element of the operational architecture, plus one for the hardware interfaces (Fig. 31). Processing nodes consist of a host computer and an arbitrary number of interfaces for communication as well as input and output.

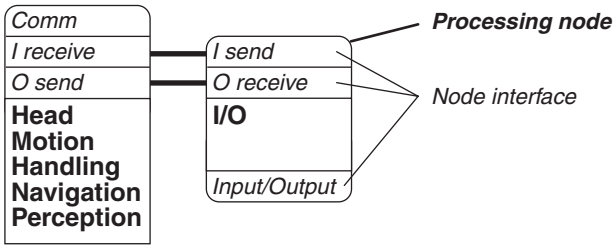

Figure 30: Robot: two-processor implementation

In the topology of Fig. 31, some of the elements have their own peripheral interfaces, additionally to the services provided by the node *I/O*. It is envisioned that *Perception* may need high input bandwidth exclusively (for bulk vision data), and *Motion* and *Handling* might require their own direct outputs. The granularity of a decomposition is up to the developers and their expectations, but, since these are tentative architectures, overdoing it merely helps to keep up future options. Thus, in the example, further refinement could lead to the decomposition of single elements into clusters of processing nodes.

The degree of distribution in actual implementations can be less than proposed in any of the envisaged implementation architectures. In fact, most often this will be the case, as the implementation architecture merely sets *limits* to the degree of distribution. This information is carried over to the component architecture (see section 4.3), which defines deployable entities, and whose granularity is based on the most detailed hardware topology provided.

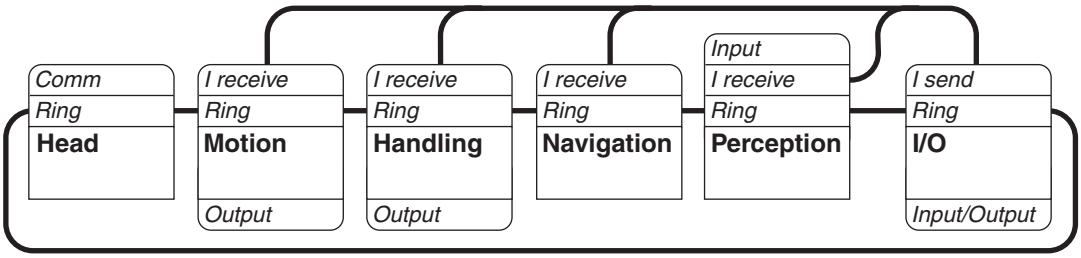

Figure 31: Robot: fully distributed implementation

In the implementation architecture, resources take the form of generic units that can be employed in a topology. Generic hardware units are processor types, which may access peripheral devices and communicate with other nodes via bus and network types. The quality of a proposed topology with respect to the problem solution depends on attributes that can be measured quantitatively. Resource adequacy is one such attribute, indicating that ideally, the processing demands of a component are just satisfied. Resource adequacy restrains both cost and system complexity, but it can only be asserted when the processing demands are exactly known, i.e., after concluding the constructive work on a component. This is also the main reason for postulating a separate architectural description (the component architecture) that is independent of any implementation architecture.

Another quality attribute is system reliability, which for systems with many parts results from a combination of the mean time to failure (MTTF) of its components. The use of redundant elements to achieve reliability *and* resource adequacy is a difficult task in itself. For a systematic evaluation of these properties tools exist that make use of genetic algorithms [NiB97].

4.3 Component Architecture

One goal in problem-oriented development of embedded systems is utmost abstraction from any envisaged implementation[1]. Furthermore, the actual construction of a system's components is undertaken within a limited and precisely defined scope, i.e., the global view is also abstracted away. This is the main intention behind a formal component architecture that unites related components into groups, and defines the necessary interconnections between these groups. This top-down view of the system to be built is the *pivot,* which forms the basis for the (compositional, i.e., bottom-up) construction work. The construction involves two separate problems according to the decomposition presented in section 3.2. Because the component architecture is less prone to frequent changes than the problem areas themselves, development may progress independently on the grounds of the previously defined architectural elements.

The emphasis is on the *kind* of elements used in the architectural description. Instead of introducing ever new kinds of elements in the description of bigger and more complex architectures, scalability is achieved by replicating generic design elements with specific behavior and interface properties. One characteristic of this approach is the limited number of precisely defined element types (as advocated by [GaS94], where properties and relationships of existing architectures are exploited), another consists in the separation of setting up the architectural description from actually allocating components onto processing elements. These two traits are in striking contrast with the notion of 'architecture' in the understanding of object-oriented methods, where the implementation inherits the structure of what starts out as an architectural design (see for example [LaH94]).

Architecture descriptions come in many guises. While they differ in their details, there are some recurring elements, namely components, connectors, systems, properties, and styles [GKP00]. *Components* represent computational elements and data stores of a system. The points of interaction between a component and its environment are called *ports.* Interactions among components are represented by *connectors,* which mediate communication and coordinate activities among components. Their interfaces are defined by *roles* played by the participants in the interaction represented by the connector. *Systems* are configurations of components and connectors. The configurations assign component ports to specific connector roles, thus restricting the component in the

1. Note that this is not contradicting section 4.2, which advocates a joint optimization of operational and implementation architectures! Rather, the proposal to use a *set* of feasible hardware topologies at the beginning of a design process is in support of the abstraction called for here.

interaction represented by the connector. In hierarchical systems, components and connectors may represent subsystems with internal architectures, a hierarchy that is sometimes implicit. For example, a functional component's internal structure may be arbitrary, as long as it is within the constraints of the component's ports and does not appear on the level of system architecture (see section 4.3.2). Another example are connectors, which may expand to several parts with their own active behavior (in case they link one processing entity to another), or else disappear completely from an implementation (with a single processor). Characteristics that cannot be expressed with structure alone are incorporated as *properties* of the elements of an architectural description, such as the interaction protocol for an interface. *Styles* can be used to constrain a general purpose architecture description language by defining a specific vocabulary and ways of use for this vocabulary.

The constraints of the problem separation set guidelines for defining generic components and component properties, including their roles in a later implementation (which are derived from a component's domain classification; see section 3.2.3). Considering the declared separation of functional from connective aspects, the component architecture specification does not need the expressiveness of 'universal' architectural modeling languages. This separation also permits to hide the internal refinement (the local architecture, as it were) of components on the architectural level, and it emphasizes the difference between strong binding (within single components) and weak coupling (among components). Therefore, only a minimum of different component types as well as ways of component interaction, hence connector types, is needed (Fig. 32). The component architecture is a purposeful description that builds on an implicit semantics of its elements[1]. It is based on a system-wide set of data, function, and interface abstractions provided by a restricted set of generic architectural elements.

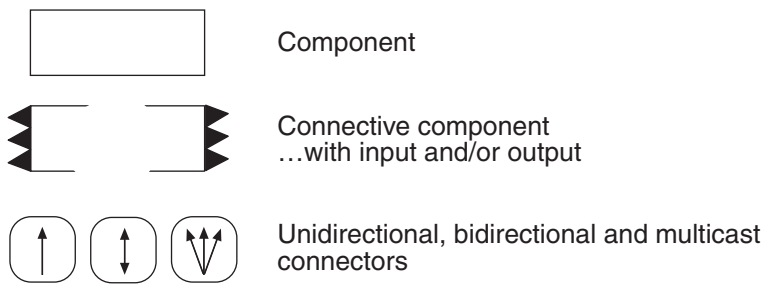

Figure 32: Component architecture notation

1. i.e., semantic properties of the generated implementations, brought about by the code generators.

While verification and property assertions are not directly attempted, relevant properties of the employed component and connector types can be expressed in general purpose architecture description languages with a sounder formal basis (for example as a style in Wright [AlG97]) to make use of the elaborate analysis tools available in these environments.

4.3.1 Specification

The emphasis in the presented approach is on its applicability to the development of real-world systems. For this reason, implementation details, such as constant and data type definitions, have been introduced into the notation. Starting from the component architecture, construction of components is undertaken in different environments, according to their problem frame membership. Since these environments produce their own code, type congruency is important to achieve syntactic agreement. On the other hand, there is the advantage of generated implementations, consisting of modules with strictly controlled execution properties. Thus, each kind of component has inherent semantic properties that can be relied on in the architecture specification. The initialization of components (and connectors, in case they are at all present in an implementation) can be included in the specification, whereas connection closure and halting of component computation need not be considered in non-terminating systems with a static architecture.

While the component architecture is explained with diagrams, actual architecture specifications are given in textual form. The specification language itself is described in more detail in appendix A.

4.3.2 Components

The general notion of a component is that of an abstraction representing a locus of computation and state in a system. Components are elements that will be refined as parts of either the functional or connection problems (see chapters 5 and 6, respectively). The interfaces of components are called ports. They define syntactic properties, namely the messages that are transmitted through a port and their types, valid types being all standard C types plus the previously defined custom types. Ports are qualified as being either sources or sinks.

Functional Component. Functional components can interact symmetrically via asynchronous connectors, and they can be attached symmetrically or asymmetrically to connective components (Fig. 33). Their constructive refinement is a CIP implementation unit (see section 5.4), which will result in a functional

reactive block whose interface consists of a number of functions associated with messages. A message is sent to this reactive block by calling the appropriate function (with possible data parameters); as a reaction the functional block may emit messages by calling one or several other functions itself. Port definitions of functional components include, in addition to messages, declarations of these functions and their parameter types.

The semantic properties of functional components are derived from the behavior of their internal refinement. It depends, for example, on the transition structure of the process connected to a port, if this port can operate according to a specific communication protocol. A conceivable addition to the CIP generators would be to extract traces from a completed functional model that could then be used for consistency analysis on the architectural level.

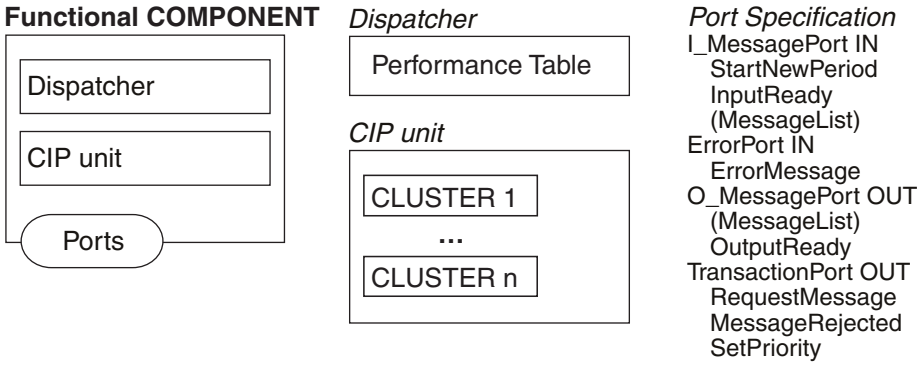

Figure 33: Functional component

Connective Component. Connective components accomplish the chores of communication message handling, data input, event extraction, and data output. The event extractor component (Fig. 34) set aside, there are but three different kinds of basic connective components (Fig. 35).

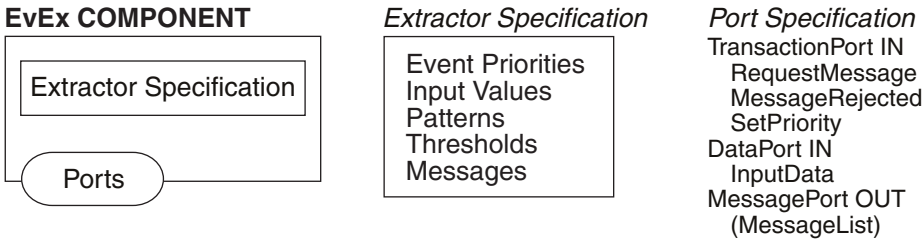

Figure 34: Event extractor component

As the name implies, event extractor components search the newly acquired input data for changes indicating events in the environment. Their specification associates binary patterns of discrete signals, or thresholds of quasi-continuous signals, with event messages. One of several lists of event priorities is

enabled, corresponding with the currently active mode of operation. Event extractor components interact with input connective components (as a sink for input data) and with functional components (as a message source).

Event extractor components are a special case among connective components, in that their specified functionality is tightly linked to the models of functional components. Not only their port message lists must coincide, but the relation of sensor phenomena with messages that is brought about must match the intentions the functional component is built on. For groups of event messages that are extracted from quasi-continuous input signals, traces are predetermined through the continuity of these signals and the inherent property of the extractor algorithm to preserve this ordering. However, nothing can be said about discrete input signals nor about correlations among different signals, since these depend (via input component) directly on the external processes.

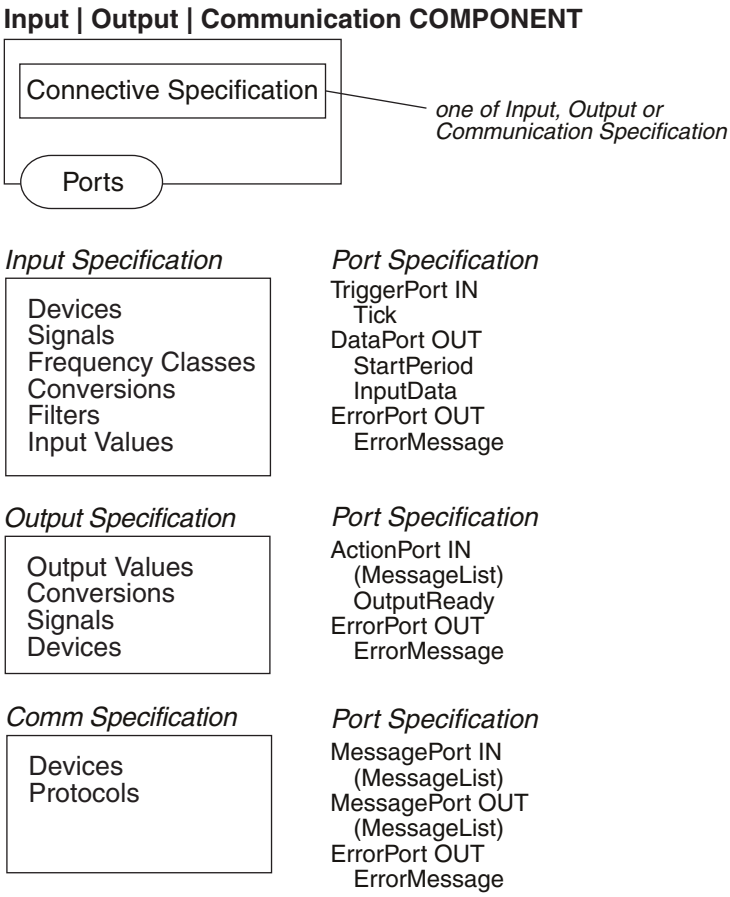

Figure 35: Connective components

With the exception of the properties of communication components, which depend on the chosen communication paradigm, the port properties of the remaining connective components (Fig. 35) remain structurally stable, only their types and sets of messages vary.

The functionality of connective components is limited to data conversion specified in terms of peripheral devices and their data formats, signal conversion and verification methods, and abstract messages. There are no message traces that could be extracted before system deployment, because these traces depend on the behavior of the external processes.

4.3.3 Connectors

Connectors mediate the interaction among components, i.e., they restrict the behavior of the attached components with protocols, sets of messages or permissible traces. Rather than expressing these characteristics explicitly, the component architecture relies on implicit properties of the connectors used. Connectors guarantee syntactic and semantic compatibility between the components they relate, and need only be present as active elements in a final implementation if the components they connect are situated on different nodes in a multiprocessor installation.

There are two basic kinds of connectors, one (low-level) is associated with connective components and their interface to other connective components, the second (high-level) with functional components (Fig. 36). The first kind is typically associated with bulk data transmission in a general form, requiring adapted low-level implementations tailored to the selected implementation variant. The second kind provides asynchronous links across the unit boundaries of functional components for potentially distributed implementations.

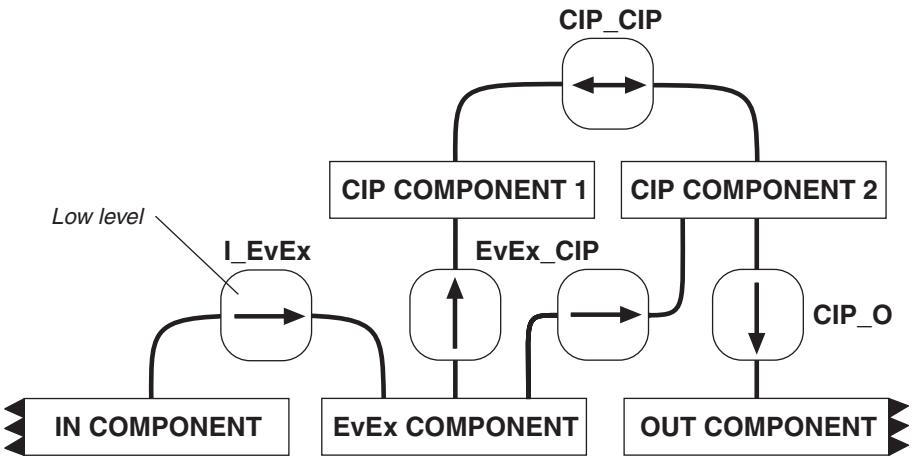

Figure 36: Connectors

Low-Level Connector. Some low-level messages are signals, while others contain data. Data transport can be simple or complex, depending on the different ways of implementation. As an example, consider a signal from a data input

component received by an event extractor component (connector *I_EvEx* in Fig. 36). If both components are located on the same processing entity and share memory, the receiver performs data reads directly from the shared memory, whereas a distributed implementation contains a mechanism for copying the data before the signal message is delivered. In any case, with single writers, contention is not a problem.

High-Level Connector. There are two kinds of high-level connectors to consider. The first kind is a true high-level connector, in that it is applicable to communication messages, which are sent from one functional component to the other (connector *CIP_CIP* in Fig. 36). These messages are queued and disposed of on processing. Queuing is required to maintain the time-ordering of protocols and processing sequences between communication partners. The queues are situated in the connector itself, permitting a choice of location for the queue if sender and receiver are on different processing entities in a distributed implementation.

The second kind is used to link connective components with functional components and vice versa. Event messages to a functional component are not queued, instead they are pulled actively from their source, the event extractor *(EvEx_CIP* in Fig. 36). This evades problems associated with phase sensitive real-time images that may become temporally inaccurate when functional processing cannot catch up with the queue contents. Like communication messages, action messages that are sent by a functional component need to be queued as well, to make up for transmission delays or asynchronous peripheral accesses *(CIP_O).* In some situations, the occasional loss of a message may be permissible, excluding messages that are discrete by nature (e.g., the setting of a switch). An example where losses can be allowed are periodic output values originating from a control loop. Should the connector queue run out of space, as a last resort these messages can be weeded out.

To maintain the option of flexible component assignment to different processing entities, connectors are asynchronous by definition, even if a specific assignment leads to a synchronous implementation. The limitation to components with a run-to-completion semantics and a sole communication mechanism via asynchronous messages effectively limits the number of connector types. Connector implementations can rely on the execution properties of the generated component implementations, i.e., a number of fixed implementations are available for each connector. To realize a connection in the resulting software system, appropriate implementation mechanisms are chosen according to the physical location of the communicating partners.

4.3.4 The Pivot

The component architecture combines the operational and implementation views of architecture into a pivotal description that serves as a basis for constructive development work. The pivot is strictly a basis only, in that it remains untouched by subsequent constructive work. Instead of refining components in place and within the context of an architectural description (as for example in object-oriented methods [SGW94], or DisCo [KaM99]), actual system construction, up to generating complete implementations, takes place elsewhere. Given the paradigms of specification and modeling tools, and the properties of associated code generators used for solving the functional and connection problems, chances are that the emerging implementation parts will fit and result in a working solution…

Another look at the robot example reveals that its component architecture (Fig. 37) takes into account the operational architecture from Fig. 29 on p. 63, in that it provides individual event extractor components for each subsystem (thus permitting separate development of these subsystems). It also takes into account the most detailed implementation architecture of the ones provided (in this case the one from Fig. 31).

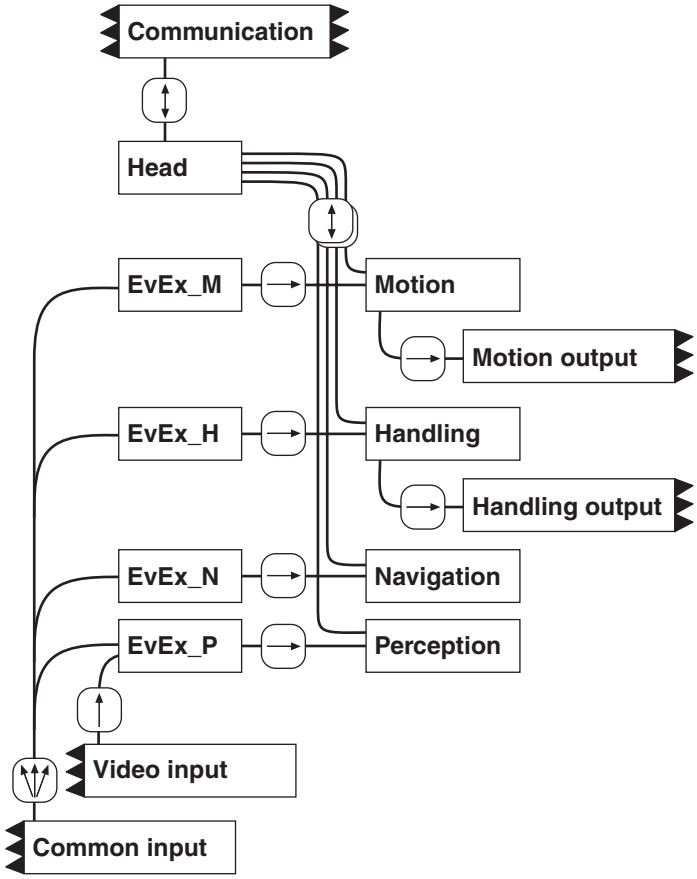

Figure 37: Robot: component architecture

The selection of a specific implementation architecture can thus be postponed to when the processing requirements are more exactly known, i.e., towards the end of the constructive phase. Only then can the allocation of components onto processing elements be evaluated, according to processing and timing requirements.

Dependencies in the development process itself are straightforward, conceding relative freedom to developers working in separate problem areas, which are well isolated by the pivotal component architecture ('local change' in Fig. 38). Alterations on the level of architecture or yet changes in system requirements are reflected in the component architecture and necessitate adaptations in all problem areas.

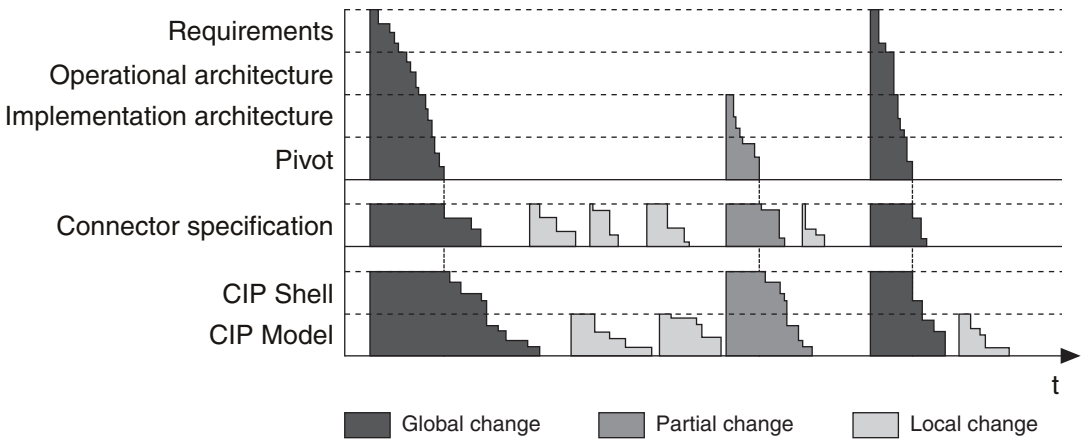

Figure 38: Development ripples

5 Functionality

Mastering a specification language is not easy. At first sight, the translation of functional requirements written in natural language into a formal notation is more difficult than attempting implementation directly. This fact presents a substantial negative incentive to the adoption of formal notations, although it forces developers to do without the advantages of abstraction on a high level, without the benefits of restrictions of a general purpose language to the subset needed in the solution of a specific problem, without code generation, and without the possibility of applying verification techniques.

Design and coding languages are rich expressive formalisms, permitting a variety of solutions to given problems, although the full range of solutions is usually much wider than is necessary or useful. Practitioners want to solve naturally occurring domain problems, for which they do not need the full expressiveness of the languages they use. Guidance in the solution of commonly occurring problems and restrictions in the use of language features are helpful [MIS98], but even more advantageous are methods that provide a means of formulating solutions in terms of the given problem.

To solve the functional problem means refining the relevant functional components of the component architecture (see section 4.3.2). Within the constraints of the component's port properties, the refinement can be structured in a problem-oriented way. An appropriate method is the construction of a formal reactive behavioral model, often called an essential model [WaM85], because it is independent of technical interface concerns. Its purpose is to specify the functional system behavior in subject-matter terms, based on the time-ordering of events. This is in accordance with chapter 3, where the connection of an implemented essential model to the real environment was established as a problem in its own right.

5.1 State Machine Models

Following the proposition made in section 2.1.1, the behavioral description is attempted with a notation based on finite state machines. Pure finite state machines, introduced below, cannot directly be used for two reasons:

- systems described as one coherent model are too complex (because of the 'state space explosion' that afflicts any non-trivial models),
- the limited expressiveness of pure finite state machines rules out algorithmic descriptions (which are indispensable in practical applications).

While more elaborate modeling frameworks eliminate these shortcomings, these must be carefully scrutinized for their abilities to fit the component architecture and to generate code that is efficient and executes deterministically.

5.1.1 Finite State Machines

State machines unite combinational circuits with storage elements. The contents of all storage elements make up the *state* a machine is in. The observable behavior at its output depends on the input sequence and the machine's internal structure. A finite state machine is defined over sets of inputs, states, outputs, and their interrelations. It can be modeled using a synchronous 7-tuple Mealy machine $(X, S, \Gamma, f, s_0, Y, g)$, with

X:	an input alphabet,
S:	a set of states,
$\Gamma(s)$:	the set of permissible events while in state s,
f:	a transition function, $f: S \times X \rightarrow S$, only defined for $x \in \Gamma(s)$,
s_0:	the initial state, $s_0 \in S$,
Y:	a set of outputs,
g:	an output function, $g: S \times X \rightarrow Y$, only defined for $x \in \Gamma(s)$.

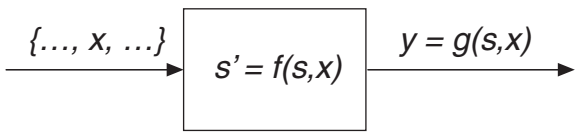

Figure 39: State model of finite state machine

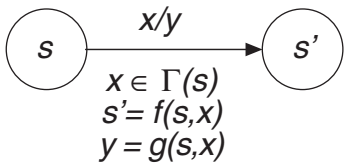

Figure 40: State diagram of finite state machine

Consider the simple example of a communication protocol with the following specification for the sender side:

- messages can be processed if they arrive while the sender is idle, otherwise they are lost,
- during processing, a copy of the message is stored, transmitted on request and a timer set,
- if an acknowledge is received, the stored message is deleted,
- when the timer expires, the message may be retransmitted on request.

This behavior is modeled by the example machine in Fig. 41, which exhibits the properties $X = \{a, t, x, r\}$, $S = \{i, m, p\}$, $Y = \{0, 1\}$ and

a:	a message arrives,
t:	the message has been transmitted,
x:	the timer has expired,
r:	acknowledge received,
i:	sender is idle,
m:	message present,
p:	message is being transmitted,
1:	output, if the message was successfully transmitted.

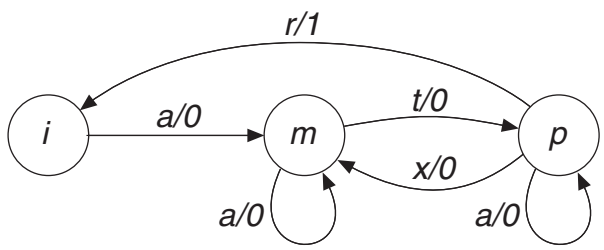

Figure 41: Example state diagram: communication protocol

Practical applications of state machines face two major problems. One problem is their limited expressiveness, generally mastered with extensions that permit additional functionality, the other regards the unwieldy size and blown-up state space encountered in models of any non-trivial system. The size of state machines grows exponentially with the number of states, a fact that precludes their straightforward use in real systems. To overcome this impediment, practicable methods view systems as a collection of smaller state machines that bring about the desired behavior of the system as a whole by mutual repercussion. They attempt to reconcile two contradictory claims: maintain an ongoing relationship with the behavior of the external processes, and offer the benefits of a well-structured solution at the same time.

5.1.2 Hierarchy

A collection of state machines can be organized in various ways. Although many reactive systems have a hierarchical structure describing the composition of system parts, hierarchy is difficult to apply. The reason is that low-level interactions often have an essential influence on the high-level behavior of a reactive system. Composition hierarchies restrict interaction to local composition levels, but these are likely to clash with the actual problem structure. Another kind of composition hierarchy denotes nested state regions of an underlying unfolded state machine (like in Statecharts [Har87]). The purpose of this hierarchical structure is clustering and refinement of behavioral descriptions, whereas interaction dependencies are not restricted by the state hierarchy. The CIP method [FMN93], on the other hand, is a compositional approach, which clearly distinguishes reactive behavior (of a process) from higher-order interaction (behavioral changes in subordinate processes). The master-slave hierarchy relation of the CIP method is a set of associated state machines, whereas hierarchical state machines in Statecharts represent compositions based on nested state sets.

5.1.3 Cooperation

In models that build on a collection of parts to describe complex behavior, cooperation of system parts within the same model is either synchronous or asynchronous. Synchronous cooperation, known from description techniques like Statecharts and ESTEREL [BeG92], is needed to model synchronous propagation of internal interactions. Asynchronous cooperation, on the other hand, which is supported by parallel modeling languages like SDL [Fae93], JSD [Cam89] or ROOM [SGW94], is necessary to express concurrency. The two ways of cooperating also support the notions of loose coupling (asynchronous) and close binding (synchronous). Therefore, the CIP method makes available both synchronous and asynchronous methods of cooperation.

5.1.4 State Machine Extension

Data processing problem solutions are aptly described as transformational systems, whereas solutions in process control show the characteristics of reactive systems. For each, there exist matching and preferred descriptions, e.g., algorithms for transformational systems, and state machines for reactive systems. The descriptions are equivalent, but it is unwieldy to realize, say, a counter with a state machine (each value taken on by the counter must be represented by its own state, resulting in blown up and unusable models). On the other hand,

algorithmic descriptions of state-based behavior are hard to understand, because these descriptions are actually programs, and as such located on a lower level of abstraction.

Inevitably, both these kinds of representation are needed to express the functionality of embedded systems; state-based models describe the functional behavior, whereas algorithmic descriptions are required for formulating computation, such as quasi-continuous control problems. Integrating algorithmic descriptions into a process model is achieved more easily and preserves the high level of abstraction of this model, making it superior to the inverse. To this end, some means of incorporating algorithmic descriptions in a state-based solution must be provided. These *extensions* of the formal model allow the addition of algorithmic elements (i.e., program code) either to states (still a Mealy machine, with code executed upon entering the state via an arbitrary transition), or to transitions (executed when the specific transition is triggered).

The equivalence of transformational with reactive descriptions is appropriately illustrated by the most simple of state machines with only one state and one transition. The entire problem solution is contained in the single transition as a code extension, which executes whenever the transition is triggered. This setup even has its uses when legacy code must be integrated into a project.

5.1.5 Semantics

The most widely used state-machine representation of behavior for reactive systems is the Statecharts formalism and graphic notation [Har87, HaG96]. While it presents intuitive ways to express hierarchy and concurrency, serious deficiencies concerning its syntax and semantics have led to the development of many variants. The variants basically fill semantic gaps with assumptions in their implementations. Not surprisingly, merging these attempts into a coherent whole has proved impossible. Some of the more severe problems (described more fully in [vdB94], which also compares 20 Statecharts variants) include:

- the synchrony hypothesis, which postulates immediate reactions to changes of state in the environment (contradicts all practicable implementations),
- the notion of instantaneous states (that are entered and exited simultaneously, i.e., in no time, according to the synchrony hypothesis), combined with events that are not consumed, but persist until the system has settled (may cause self-triggering and violates causality),
- no distinction between synchronous and asynchronous interaction, nor between external and internal events (global events and a global name-space impede the composition of separate models),

- non-deterministic process activations and cyclic transmission paths allow infinite transition sequences (introducing non-deterministic behavior).

The CIP method, on the other hand, is based on a formal semantics [Fie00a].

5.2 Communicating Interacting Processes CIP

Using the CIP method [FMN93, Fie94, Fie99], the functional behavior of a system is specified by an operational model of cooperating extended state machines. 'Operational' means that the model is formally executable [Zav84]. The CIP meta-model is based on a compositional mathematical formalism developed for this purpose [Fie00a]. Mathematically, a system is defined as the product of all state machines comprised, corresponding to a single state machine in a multi-dimensional state space.

CIP Tool[1] [CIP00] is a framework of graphic and text editors supporting full coherence among various architectural and behavioral views. The tool is a component framework based on an object model implementing the CIP meta-model. Specifying models by composing and associating elements in a graphic editor ideally fits the problem-oriented construction process of the method, providing more flexibility and intuition than language-based specification techniques. The tool generates code and exports model properties in various formats, suggesting verification and model checking by means of other tools.

Models are built with a notation of synchronous processes enclosed within asynchronous clusters. Clusters are the coarse-grained elements combined in the implementation unit of a functional CIP component as present in the component architecture. For a comprehensive view of the system to be built, all the units defined in the component architecture can be dealt with simultaneously, and their interrelations defined on a functional level of abstraction. This is especially important for implementations that may be placed into a multiprocessor environment. The method permits to deal with both event-based and hybrid systems with integrated quasi-continuous processes. Event and communication messages from the environment cause a transition in the receiving process, which activates other processes for his part. Transitions may generate pulses to other processes, as well as action and communication messages for the environment, and they can also contain additional code for data manipulation and control functions. Hierarchical system decomposition is further supported with a powerful master-slave concept.

1. CIP Tool®, partly developed at the Computer Engineering and Network Laboratory TIK of ETH, is now a registered trademark.

Chain reactions initiated by a message cannot be interrupted; their execution depends on the message itself and the internal states of all activated processes. A local architectural description (interaction net) together with explicitly specified execution cascades set upper bounds to these chain reactions and prevents the specification of cycles. Code generation for complete models is straightforward and produces efficient and deterministic implementations.

A system is built with only a few essential elements (processes and channels embodied within clusters), which are put together with appropriate glue parts (interaction net and pertaining pulse translation, message translation, inspection and master-slave assignments). The basic elements are self-contained and keep their interface properties even when detached from the surroundings.

In accordance with section 3.2.1, the construction of a CIP model is based on a virtual connection to the external processes. The virtual interface of the external processes consists of collections of event and action messages designating instantaneous subject-matter phenomena occurring in the environment. Events, often called discrete events, are caused by the autonomous dynamics of external processes. Continuous behavior of external processes is captured by comparing quasi-continuous input signals to defined thresholds, or by periodic temporal events with associated state values. Actions are environment phenomena caused by reactions of the embedded system.

Even if the task of associating environment phenomena with messages is not in the scope of functional models (instead, it is resolved in the connection, see chapter 6), functional models are constructed with the concept of *real* events and actions. It is safe to do so on the grounds of the problem frame in Fig. 23, p. 55, which establishes congruity of real phenomena with their respective messages.

In the final installation, the functional solution is contained within a reactive block (Fig. 42). The virtual connection becomes real through the implementation of the connection solution, which transmits corresponding messages whenever an event has occurred or an action must be produced.

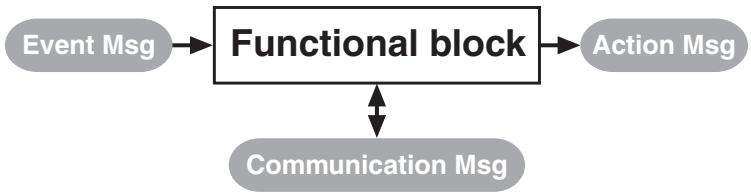

Figure 42: Functional reactive block

5.3 CIP Constructive Elements

A CIP model is composed of a set of state machines termed processes, groups of which are held together by clusters. Process cooperation is asynchronous across cluster (and system) boundaries, synchronous or asynchronous within individual clusters. Formally, a cluster represents a state machine with a multi-dimensional state space, with the state of a cluster defined by the tuple of its process states. Although clusters as well as processes represent parallel behavioral entities, their composition semantics is essentially different; clusters model concurrent functional blocks of a system, while processes represent orthogonal and synchronously interacting components within a cluster. Hierarchical composition structures based on a simple 'part of' hierarchy relation can be introduced at both levels.

5.3.1 Process

Processes are modeled as extended finite state machines (Fig. 43). Process states are represented by circles, transitions by labelled transition boxes. State transitions are activated by either an asynchronous message or a synchronous pulse. An input message, for which the current state has no outgoing transition causes a context error. Pulses, for which there is no transition, on the other hand, are ignored. In a transition, one pulse and a message to each outport can be emitted.

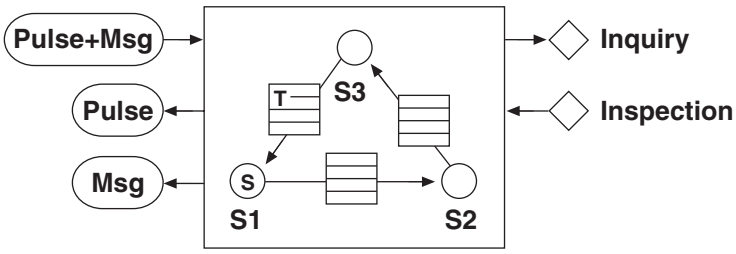

Figure 43: Process structure

Data processing and algorithmic concerns are supported through state machine extensions, which consist of static process variables, data types for messages and pulses, operations and conditions. By means of state transition structures and operations executed within transitions, functionality can be specified on two different levels of abstraction. Variables, data types, operations and conditions are formulated directly in C, the programming language of the generated code, and incorporated in-line. From the high level modeling point of view these constructs represent primitives, which add computational power to the pure state machine models. This pragmatic approach based on the implementation

language permits the use of functions, data types and other elements from existing sources and libraries. It also presents a weak point, since C constructs may inadvertently compromise the implementation properties of the functional model. Therefore, only a limited subset of the language is permissible, excluding for example all non-deterministic constructs, such as loops and memory management functions.

The interface of a process consists of three parts (see also section 5.3.2):

- the communication interface is defined by one or more inports and outports. A port is specified by the set of messages to be received or sent. Each inport and outport is connected in the communication net to an incoming or outgoing channel, respectively,
- the interaction interface is defined by distinct sets of inpulses and outpulses,
- inquiries defined for a process permit the unobtrusive inspection of its inner states by other processes in the same cluster.

Process Arrays. Replicated processes are modeled as multidimensional process arrays. The multiplicities of the singular array dimensions are defined by abstract index types. Using common index types for different process arrays allows modeling of finite relations among process arrays, usually expressed by means of entity-relationship diagrams.

5.3.2 Synchronous and Asynchronous Cooperation

Communication. Processes of different clusters communicate *asynchronously* with each other and with the environment by means of messages, sent and received via channels. Channels model an active communication medium, which retains the sequential order of transmitted messages (Fig. 44 shows the communication channels and messages of a simple motor controller[1]).

```
CHANNELS
   Commands MESSAGES enable, disable
   SwitchEvents MESSAGES closeSwitch, openSwitch
   ControllerEvents MESSAGES Standstill, Sample

   ControllerActions MESSAGES MotOn, MotOff, MotValue
   IndicatorActions MESSAGES GreenOn, YellowOn, RedOn, LightOff
```

Figure 44: Motor controller: channels and messages

Asynchronous communication implies that the write and the read action of a message transmission takes place in different cluster transitions. Processes rep-

1. The motor controller system is further explained in subsequent examples.

resent receptive behavioral entities, which must accept messages delivered to them at any time. The properties of those channels that are seen at the system interface must match the interface definition at the component architecture level. The transition structure of a process hooked up to a channel defines the traces that this channel can accept or produce. This information can also be used for consistency checks.

The communication topology is specified by a graphic net model, in which channels are attached to process ports (Fig. 45). Source and sink channels model the virtual connection to the environment, while internal channels remain part of the model and do not show up at the interface.

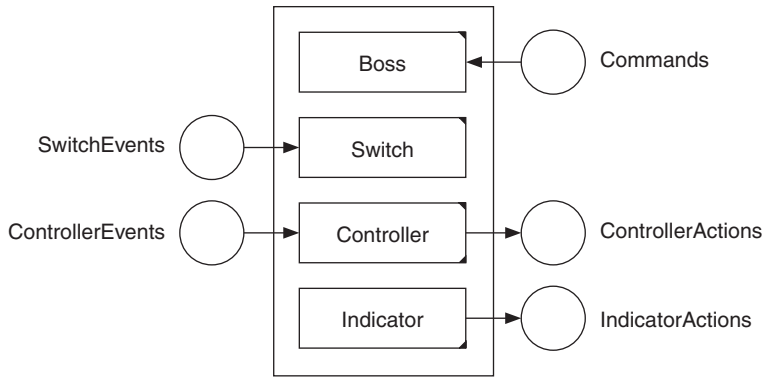

Figure 45: Motor controller: communication net

Interaction. The processes of a cluster interact *synchronously* by means of multicast pulses. Pulses represent internally transmitted events. The straight directed connectors of the interaction net (Fig. 46) define the pulse flow structure of a cluster. Every connector has an associated partial function termed pulse translation, which relates outpulses of the sender to inpulses of the receiver process (Fig. 47). Rhombic connectors declare state inspection (see below).

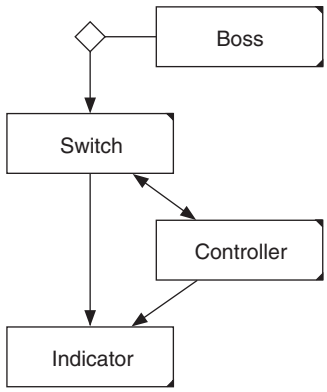

Figure 46: Motor controller: interaction net

A cluster is *always* activated by a channel message, which leads to a state transition of the receiving process. By emitting a pulse, the receiving processes can activate further processes of the cluster, which can in turn activate other processes by more pulses. This chain reaction is not interruptible and defines a single state transition of the entire cluster. Aside from generating pulses, activated processes can also write messages to their output channels.

```
PULSE TRANSLATION
   Switch start          → Controller switchOn
   Switch stop           → Controller switchOff
   Switch clearRequest → Indicator requestClear
   Switch setRequest     → Indicator requestSet
   Controller spinup     → Indicator running
   Controller spindown   → Indicator stopping
   Controller standstill → Indicator stopped, Switch stopped
```

Figure 47: Motor controller: pulse translation

For models with multicast pulse transmissions, the structure of the interaction net does not sufficiently restrict the potential pulse transmission chains, as non-deterministic process activations and cyclic transmission paths are possible in general. To ensure deterministic pulse propagation, interaction is specified as sequential multicast. The outgoing interaction connections of each process are therefore defined as a totally ordered set. If a process emits a pulse, the receivers determined by the partial pulse translation function are triggered sequentially in the specified cast order to cause subsequent interaction chains. Thus, cluster transitions represent well defined sequences of process activations. To ensure bounded response times of system reactions, cyclic interaction paths must be excluded, a model property that is enforced by the tool.

The specification of process reactions and pulse interaction by means of transition relations and pulse translation functions formally defines a deterministic cluster behavioral model, from which all potential interaction sequences can be deduced. All potential interaction sequences can be shown graphically by the tool, either in the form of cascades (with processes as interacting elements), or as interaction trees, which also provide the details of pulse translation.

State Inspection. Conditions of a process' state transition structure are allowed to depend on the states and variables of other processes of the same cluster. Read access to the data of a process is called state inspection; like in object oriented models it is granted via access functions termed inquiries. Unlike in pulse propagation, which involves both the sender and receiver processes, an inspected process remains passive. Static context dependencies between processes by means of state inspection are declared as rhombic connectors in the interaction net (Fig. 46), arrows denoting the direction of data flow.

5.3.3 Behavioral Structuring

The state transition structure of a process specifies how the process must react to inputs by state changes and generated outputs. Often, such a behavior becomes quite complex due to inputs that must influence the full future behavior of the process. An alarm message, for instance, must lead to behavior different from the normal case, until the alarm is reset. Dealing with these behavioral variants in the same transition structure results in a superposition of the normal and the alarm case.

Modes. Since different ways of behavior that are conceptually distinct occur frequently in practice, the full behavior of a process can be modeled by a number of alternative behaviors, termed modes (Fig. 48). Each mode is specified by its own state transition diagram, based on the process' static set of states, ports and pulses. A mode change does not affect the current state, which can only change when a transition in the active mode is triggered by an input. This rule reflects the fact that higher-order interactions do not alter the history of basic reactions. Behavior change without context change is typical of reactive systems. Examples are the initialization, working, and shut down behavior of device controllers, the modes of operation in classic control systems, or the various error modes of protocol drivers in layered communication systems.

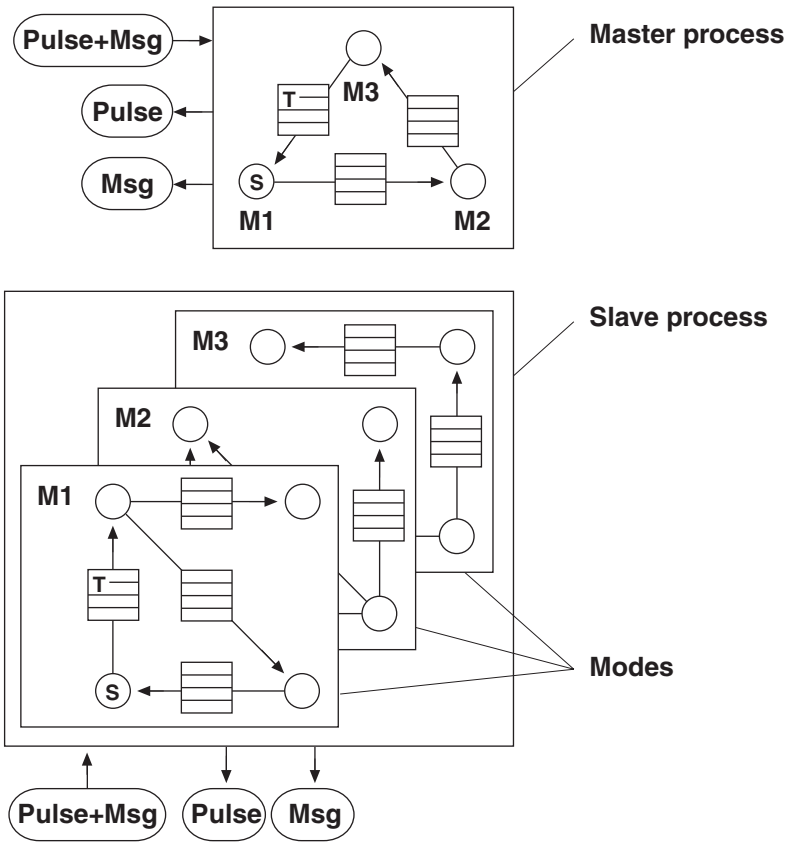

Figure 48: Master-slave hierarchy

Master-Slave Relationship. Modes introduce higher-order interaction in the form of a master-slave relationship, in addition to and aside from the basic interaction between processes and the environment or other processes. Higher-order interactions change the influence of basic interactions, while basic interactions cause state changes in the involved components. The generic master-slave pattern consists of a master, which can induce behavioral changes in a slave. "One man's state is another man's process" genuinely expresses the master-slave relationship.

Mode changes of a process can be induced by one or more processes designated as master. This relation within a cluster is specified by a master-slave graph (Fig. 49). A triangle connects one, two or three masters to a slave. The graph must be acyclic, a restriction complying with both the problem domain and the quest for comprehensible system descriptions. When a slave is governed by more than one master, a potential mode control conflict is resolved in a master agreement associated with the master-slave connector. A master agreement, essentially a multi-dimensional table, defines how the mode of a slave is determined in relation to the combined master states.

MODE SETTING Controller

Boss STATE	off	on
Controller MODE	disabled	enabled

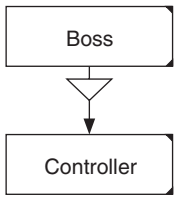

Figure 49: Motor controller: master-slave hierarchy

The active mode of a slave is determined by the current states of its masters. The association of master states and slave modes is specified by a corresponding mode setting table, which defines a total function from the Cartesian product of master states to the modes of the slave (the table in Fig. 49 relates master states and slave modes from Figs. 50 and 51, respectively).

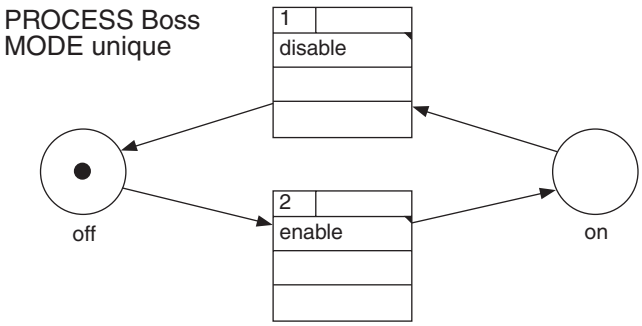

Figure 50: Motor controller: Boss (as master)

Thus, a mode change of a slave can occur whenever one of its masters changes its state. Even when the *Controller* process of Fig. 51 is in the *off* state in the *enabled* mode, its master can induce a switch to the *disabled* mode. The effect is simply that the motor will stay *off* when a *switchOn* pulse is later sent to the process. If, for functional reasons, a mode change should not occur in certain states, it must be prevented by means of an explicitly modeled interaction between slave and master.

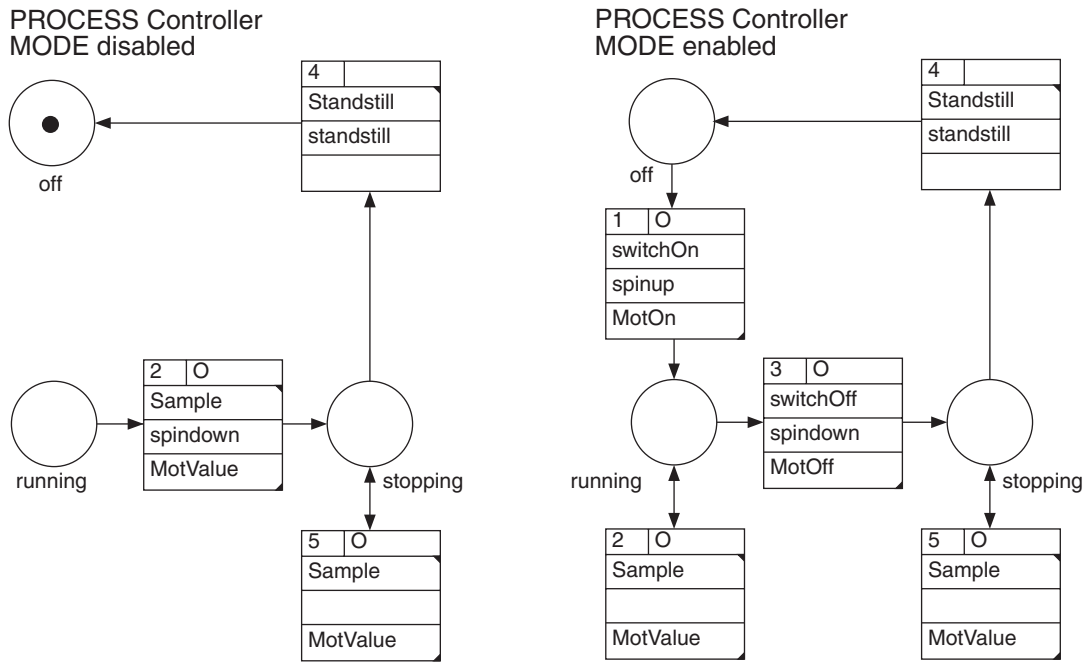

Figure 51: Motor controller: Controller slave modes[1]

A slave can itself initiate a mode change by sending a pulse to one of its masters. This scheme is used typically when an error is recognized by a slave and its master must then be triggered to induce a change to an error mode in other slaves as well.

5.3.4 Extensions

To support data processing and algorithmic concerns, the pure state machine model is extended. Possible extensions are static process variables, data types for messages and pulses, operations and conditions. Variables, data types, operations and conditions are formulated in C, the programming language of the generated code. From the high level modeling point of view, these constructs

1. The initial state of a process is marked by a token. High and low corner marks in a transition box indicate that the associated symbol denotes an input or an output message, respectively.

represent primitives, which add computational power to the pure models. The specified code constructs are included in-line in the generated code, a pragmatic way to easily incorporate functions and data types defined elsewhere.

Conditions. The same behavior as with a master-slave relationship can be achieved in the motor controller example with a completely different technique, albeit at the cost of departing from the pure state machine model. The two modes of the *Controller* process can be combined into a single mode, which brings about non-deterministic behavior. When the process is in the *off* or *running* states (gray in Fig. 52), there are two potential transitions for the input messages *switchOn* and *Sample,* respectively. Associated conditions render the process behavior deterministic. Such conditions can depend on the input data and the values of the local process variables, but also, by state inspection, on the states and variables of other processes of the same cluster.

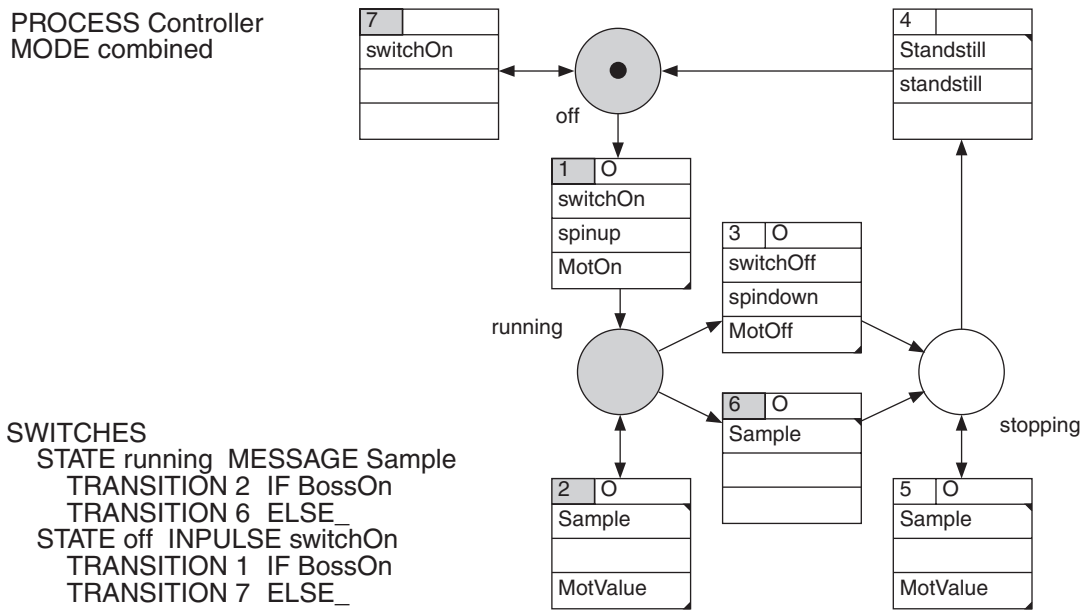

Figure 52: Motor controller: Controller combined mode

Since the example attempts to replace the master-slave relation introduced in Fig. 49, the switches of the motor controller depend solely on the *Boss* process. This process makes information about its state available through an inquiry (Fig. 53), added to the model of Fig. 50.

INQUIRY
```
int BossOn (void)
{
  return (STATUS.STATE == on);
}
```

Figure 53: Motor controller: inquiry

Operation. Another extension to the pure state machine model consists in adding code snippets to transitions (which are then marked with an O). An example are algorithms for continuous control, developed separately and used as complete building blocks. The motor controller contains two extensions of this type, associated with transitions *running:Sample:running,* and *stopping:Sample:stopping,* with the controller algorithms for running and stopping the motor, respectively. In the transitions triggered by *switchOn* and *switchOff* these algorithms are initialized (the description applies to both Figs. 51 and 52).

5.4 CIP Implementation Units

Implementations of a CIP model are defined by partitioning the set of clusters into implementation units. For each functional component of the component architecture at most one unit is permissible, although several components can be combined into a single unit. A unit is transformed into a reactive machine with interface properties conforming to those defined in the component architecture. The code of an implementation unit is made up of two parts, the CIP shell and the CIP machine, which are produced in two individual generation steps.

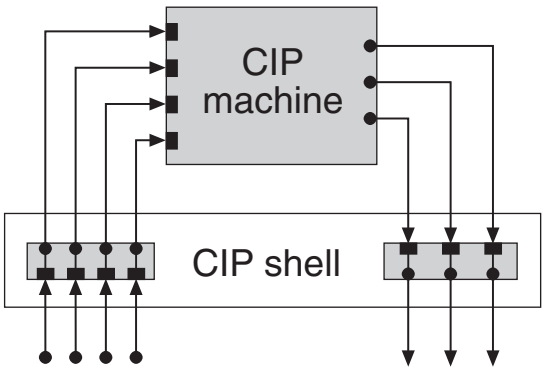

Figure 54: Implementation of a CIP unit

CIP Shell. The CIP shell represents the interface of the implementation unit: it consists of two linear structures of function pointers, one for the incoming and one for the outgoing channels. The CIP shell code is generated from the channel specifications only, and so is independent of the modeled clusters.

CIP Machine. The CIP machine is a passive object implementing the reactive behavior of the CIP unit; it is activated by channel function calls through the input shell. Every call triggers a cluster transition from which channel functions may be called through the output shell.

Once triggered by a channel message, the reactive machine runs to completion, i.e., it activates all processes involved in a cascade along the interaction tree, which is well-defined for each input message (see section 5.5.3). Performance information about the worst-case execution path of each interaction tree, including transition operations, is obtained from an abstract model representation that is generated by the tool along with the implementation code. The possible interaction paths of a model (the number of which is small compared to models without interaction restrictions) are used for setting up an exhaustive series of measurement runs on the target itself. In the final implementation, performance data thus obtained are brought in for use by a simple run-time dispatcher (see section 7.2).

Furthermore, because of the cooperation of parallel entities defined within CIP models, there is no need to implement conceptual parallelism by means of multi-tasking. As a rule, only one CIP unit is present on a single processor. The overhead associated with task scheduling is thus reduced to the sequential dispatching of just a few tasks, basically the ones already identified in Fig. 16, p. 49: input, processing, and output. The details of integrating functional reactive blocks into complete implementations are treated more deeply in chapter 7.

The environment-oriented development process of the CIP method allows the various CIP shells to be modeled at an early stage. The tool supports locking of these interface specifications for extended periods. In accordance with the constraints set forth in the component architecture, the shells serve as semi-rigid joints among the functional solution and its connection to the environment. Functional model development thus takes place in its own undisturbed scope, and it can proceed concurrently with ongoing activities in other domains.

Partitioning. System partitioning depends on the component architecture, which postulates separate deployment of its components onto their own processing entities. Clusters, representing very weakly coupled functional blocks, are the smallest entities assignable to an implementation unit. Therefore, a functional element of the component architecture must be represented by at least one distinctive cluster. For specific implementations, however, all clusters present in a system may be assigned to a single unit. In that case, communication channels, i.e., the ones providing cluster interconnections, disappear from the unit's interface, if desired. Their implementation, including buffers, becomes part of the generated functional code.

As an example, the internal communication channels of the robot controller (according to its component architecture in Fig. 37, p. 73) give rise to a communication net as shown in Fig. 55. Channel interface functions for the vari-

ous connections of cluster *Head* with the working clusters and vice versa are present in the fully distributed implementation, whereas in a one-unit implementation the entire network would disappear from the interface.

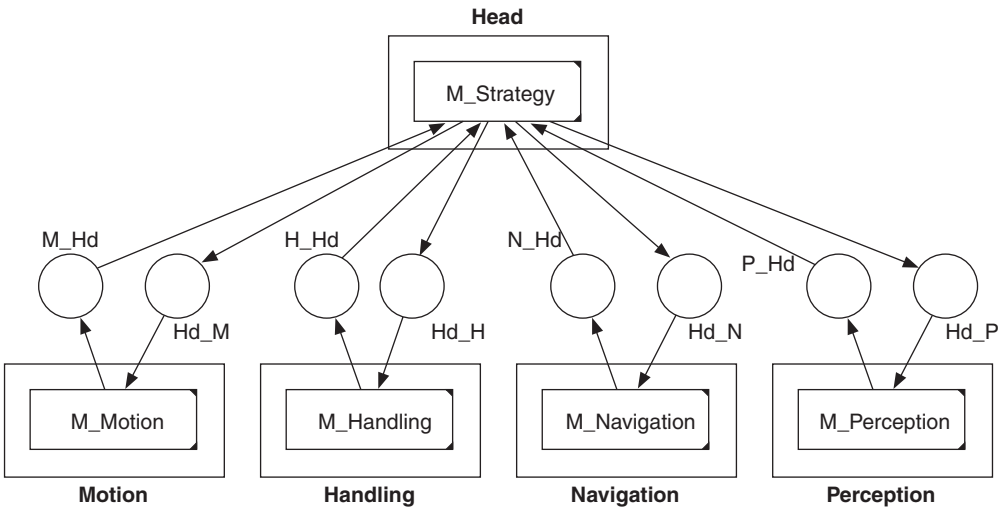

Figure 55: Robot: internal communication

The definition of several implementations containing dissimilar units and cluster partitioning is advantageous even in those cases where only a distributed target implementation is envisaged, as single-unit implementations can be easily integrated into a simulation environment (see section 7.2.3).

5.5 Working with CIP Models

Working with behavioral models instead of programming requires a different attitude towards the development process itself. Not unlike the plane taken to the streets by a driver who mistakes it for a car, the new wings of a powerful modeling environment can make 'programmers' hit the walls. Some of the concepts that favor take-off are introduced below.

5.5.1 Functional Models within a Context

An operational behavioral model provides a well defined level of abstraction. In addition, the CIP method adopts the concept of environment-oriented behavioral modeling from the JSD method [Cam89]. This concept bases the development of control systems on a realized model of the environment inside the system to capture the essential behavior of the controlled processes. This leads

to the construction of CIP models consisting of three parts, elaborated in sequence (Fig. 56):

1. the virtual real world interface (events and actions),
2. the behavioral context model,
3. the functional model.

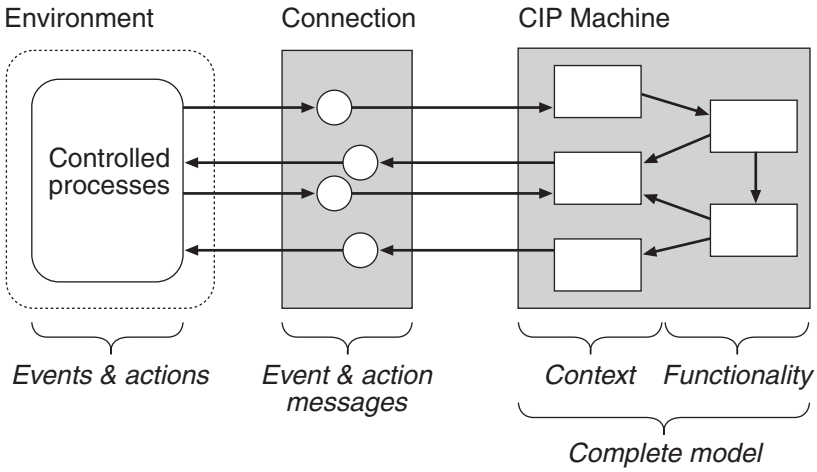

Figure 56: CIP high-level process interface

Events and Actions. The context of any functional model is not only determined by phenomena of the environment, but also by the amount and quality of the information that can be made available about this environment in the form of events and actions. Elaborating event and action collections represents a crucial development step, because it determines the level of abstraction used to solve the functional problem. On the one hand, the collected events and actions must suffice to bring about the required behavior of the environment. On the other hand, the feasibility of the connection must be ensured by verifying that all events can be recognized, and all actions can be produced by means of the available interface devices.

Events and actions can have attributes, which are transmitted as data with the corresponding messages. An event attribute describes a circumstance of an occurring event, such as the bar code read by a scanner. An action attribute describes a circumstance to be brought about when the action is performed, like the position of a valve.

Events are classified into process events and temporal events. Process events are caused by the autonomous dynamics of external processes, while temporal events occur at prescribed points in time. In general, a process event is related to a discrete change in the external process. Discrete states in a model often denote a whole range of external process states, thus representing abstractions

essential for the required functional behavior. Examples are the level ranges of a liquid in a vessel, or the set of ready states of a complex device. Continuous states are complemented by matching event extractor specifications that capture the behavior of continuous processes (see section 6.4). Sampling events (such as the one represented by the *Sample* message in the motor controller example) are periodically occurring temporal events whose attributes are the sampled process states. Similarly, the embedded system influences continuous processes by repeated production of attributed actions.

While events and actions that are required and available are obtained by looking at the controlled processes, the process interface of functional architectural components is determined by the sets of related event and action messages.

Context Model. With the virtual real word interface given by the collections of events and actions, the first part of the model to be constructed is the context model (Fig. 56). The purpose of the context model is to establish interface processes for the model and to connect them virtually to the environment. The interface processes are deduced from descriptions and process models of the environment. They receive event messages and produce action messages through appropriately specified source and sink channels. The state transition structures of the interface processes describe the valid sequences of received event and produced action messages, in other words, they represent protocols of the virtual communication with the environment. Thus, the context model formally describes the behavior of the individual external processes, as seen from the CIP model.

Functional Model. In the third step, the CIP model is completed by creating function processes, which interact and communicate with the processes of the established context model. Modeling interface processes means understanding the behavior of the external processes; but it also means anticipating the way they will be controlled when the system is completed. It is by means of the function processes that the required behavior of the controlled processes in the environment is finally brought about. In a development, it is commonplace to tackle the primary functionality of a system first, based on the normal behavior defined by the context model processes.

Inevitable expansions must later be added to take care of behavioral variants, such as error modes, unexpected events, etc. Usually, the bulk of a completed solution can be attributed to such expansions, which can only be added at the cost of breaking up the neat initial structure. With the CIP method, this structure is preserved, because extensions are added to processes as what they genuinely are, namely additional modes.

5.5.2 Initialization

A system must be initialized in two phases, consisting of the initialization of the control system itself (low-level activities, such as bus and address modifier configurations, the initialization of peripheral devices, memory distribution, setting up the task structure; higher level activities such as initializing the inner states of models and control algorithms), followed by an explicitly specified 'learning' mode of operation. This mode with limited functionality endeavors to copy the state of the external controlled processes to the inner state of a model by observing the undisturbed or only gently influenced behavior of the external processes.

5.5.3 An Eye on Interaction

Interaction is a key issue in any system comprised of different parts, whether they are functions, classes, or model components on a higher level of abstraction. It is difficult to keep track of interactions and interaction participants in any sizeable project, and the more recent programming paradigms have not made it easier. In a modeling environment that does not restrict component interaction (like Statecharts, with its global event broadcast), the task is even more laborious. CIP Tool assists the developer not only by obliging her to explicitly define interaction paths, but also through different graphic overviews derived from the model itself.

Cascade. Cascades show sequences of potential process activation. This overview abstracts from messages and pulses, but provides insights into a cluster's structure by displaying the longest possible activation paths, with individual processes of the interaction net as a root. As an example, the cascade of process *Controller* in Fig. 57 arouses suspicion, as *Controller* and *Indicator* appear twice.

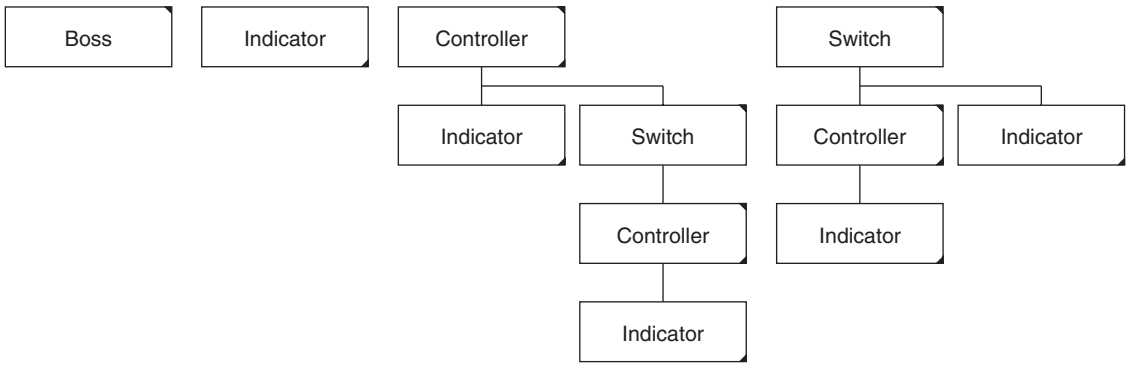

Figure 57: Motor controller: cascades

Interaction Tree. The more detailed interaction trees (Fig. 58, only shown for process *Controller)* reveal that this is not a design flaw, but indeed a possible sequence. The sequence occurs, when the motor is reaching its stopped state, and the start button is being pressed at the same time.

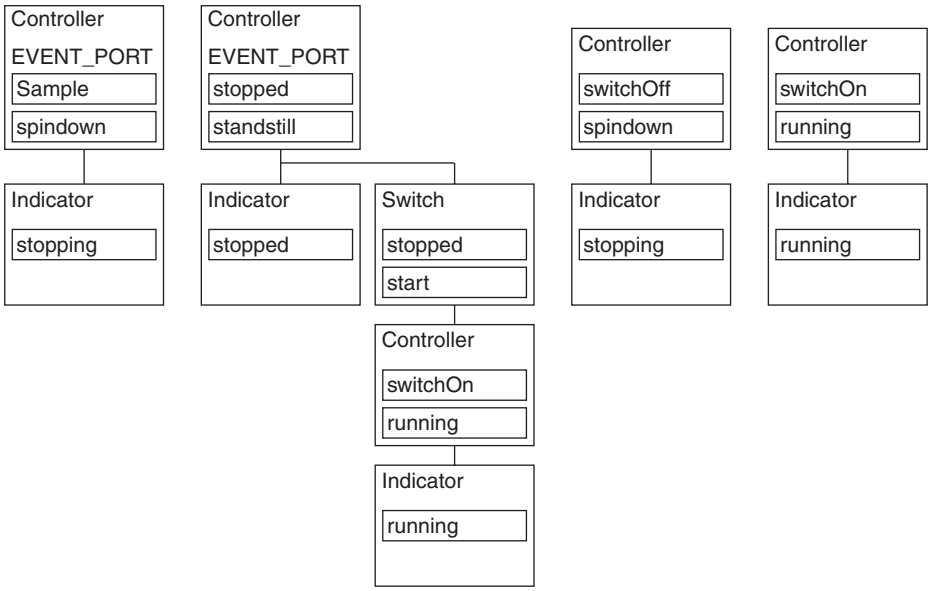

Figure 58: Motor controller: interaction trees

Reporter functions export entire CIP models in selectable detail for documentation purposes. Other non-formal descriptions (such as UML-diagrams) may also be generated. They can be useful in highlighting certain aspects of a solution, but present a one way path.

5.5.4 Behavioral Patterns

Many processes that are used to solve practical problems show the same recurring behavior, a simple example being that of a process with on-off behavior that can be used to model all kinds of switches and other discrete sensors. For many of these common ways of behavior, abstract patterns emerge, similar to those proposed in object-oriented design [GHJ95]. However, the two kinds of patterns are not on the same level of abstraction; while design patterns are abstractions of key aspects of program structures, CIP patterns present abstractions of behavior.

6 Connection

One characteristic of embedded real-time systems is their tight relationship with peripheral devices. The interfaces of these devices come in all flavors and may lead to a plethora of different value encodings and timing properties in a single system. Aside from complicating the initial development, the necessary changes in systems that are inadequately structured can be severe if one of these complex and restrictive external interfaces is replaced.

This problem was recognized early on and has led to the concept of 'abstract interfaces', an attempt to isolate a system's functional code from all interface device details [HPP81]. An abstract interface may consist of virtual devices with device-like capabilities that are partly implemented in software. Virtual devices simplify the rest of the software, liberating it from interface particulars and enforcing a disciplined and efficient use of resources. However, the idea of replacing real devices with virtual ones has its own problems. A virtual device that is based on a deficient abstraction may need to be changed at its interface when the device is replaced (if the abstraction did not cover all possible device variants). On the other hand, internal changes may become necessary even when the device remains the same (if device-independent functionality that was misplaced into the virtual device needs to be adapted).

Another problem is the implementation of virtual devices. Device access requirements frequently postulate their own protocols, which must be dealt with locally and lead to independent entities with their own behavior and state. Consequently, very often virtual devices are implemented as tasks and integrated in a system's scheduling scheme. Thus, a complex operating system is called for by the virtual devices even in those systems where the functional code alone could be implemented with a simple dispatcher [LaH94]. In these cases, the device interface tasks themselves introduce jitter and may undermine the required sampling properties of individual signals (see section 2.1.2).

How can formal methods be applied in the connection domain? Complete formal descriptions would not only describe software, but involve the device hardware as well. This is infeasible for reasons of complexity, even if device internals are known (usually they are not). Accordingly, developers depend on hardware and software elements that have proved to be useful and reliable in other installations [Web97]. If only parts of the connection are considered, however, formal descriptions *are* useful. Hardware items are then viewed as black boxes, and their low-level access routines left in the domain of programming, although the ways how these routines are combined are described formally.

The solution of the connection problem (see section 3.2.2) attempts to interface the functional code to a specific environment (Fig. 59). Because it must do so within the constraints of the component architecture (see section 4.3.3), updating input values or the opposite, setting new output values, are straightforward duties for which a suitable abstraction can be found. The absence of structural difficulties encourages simple, sequential solutions for the three basic tasks:

- data input, conversion and verification, with subsequent mapping to functional event messages,
- data output of converted and buffered functional action messages,
- communication with other nodes or across the system boundaries, received and sent via buffers as communication messages.

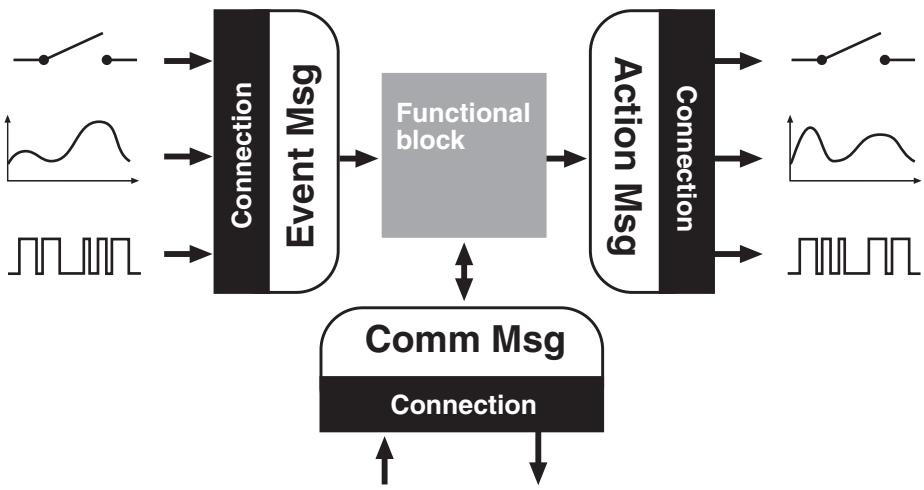

Figure 59: Interfacing the functional block

The chores of the connection fall into two categories: one is concerned with low-level device access, data conversion and data transport (in distributed implementations). The second category deals with event extraction. The required code and data structures are generated from abstract specifications

that are given in terms of hardware devices, signals, and signal properties in the case of input and output, and in terms of signals and event and action messages in the case of event extraction (see appendix A). Code generators can exploit the constraints of the component architecture persistently to achieve simple sequential solutions for each of the architectural components.

Aside from freeing developers of programming troubles, the main objectives of this approach are a uniform distribution of processor loads, code with properties that fits the overall implementation scheme (see chapter 7), and the possibility to create implementation variants from the same specification. Simpler variants, such as the special case of simulation (where values are merely copied from and to data vectors), are based on a subset of the specifications.

6.1 Device Access

As noted above, working out the details of interfacing to the existent heterogeneous peripheral hardware is a difficult task in itself. Input and output operations on some devices are fairly complicated and must be made with multiple accesses, possibly also requiring delay periods in between (Fig. 60). In addition, variations with regard to data width, channel number, multiplexer setup and possibly filter and amplifier settings must be accounted for. In present-day systems it is often not possible to exploit all the features of a hardware device, because it would be too tedious to program and impossible to master all potential side-effects. Consider a 16-channel A/D-converter with the possibility to set parameters together with the number of the channel one wishes to convert. Filter and amplifier settings must be issued ahead of the conversion to allow for signal stabilization, i.e., the current setting is influenced by the previous one, introducing dependencies between channels not otherwise related. In practice, this situation is often avoided by using the same settings for all 16 channels and enabling the round-robin conversion option many devices offer; a waste of bandwidth for all but the signals with the highest requirements.

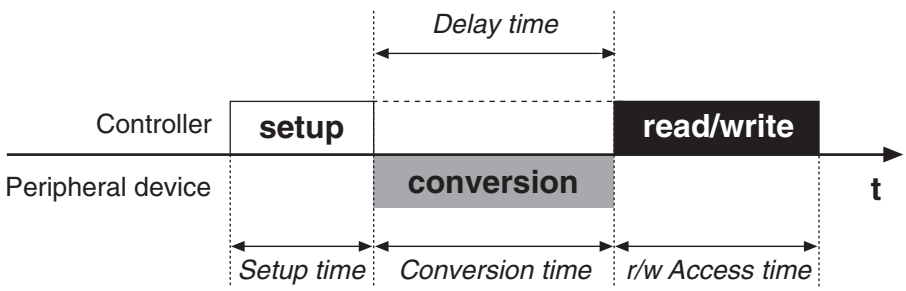

Figure 60: Device access sequence

Selective specific multi-stage accesses combined with delays as shown in Fig. 60 offer a potential for increasing performance, in that all the device accesses together present a collection of tasks with precedence and exclusion relations. A schedule for these tasks must satisfy the following requirements:

- a transaction consisting of the stages 'setup–wait–read/write' on an individual peripheral device must be completed before another transaction may be initiated with the same device,
- delay times must be matched or surpassed.

Accesses as Tasks. The obvious solution of taking 'task' literally and casting device accesses to individual tasks that are schedulable by a operating system is inefficient. The cause of this inefficiency is the conversion delay, because the required delay times usually are too short to justify rescheduling at the task level. These delays thus remain periods where the processor idles until the next read or write operation is allowed.

Accesses with Interrupts. Another solution uses the capability of many peripheral devices to notify the processor of the end of a conversion themselves through a hardware interrupt, freeing the processor in the meantime. Because the schedules of 'normal' processing are not independent of these interrupts (since scheduled tasks initiate device accesses in the first place), there is a tendency that interrupt service routines take their processing time from the same tasks on every occurrence, a fact that run-time schedulers cannot take into account. Furthermore, interrupts introduce non-deterministic execution sequences, augmenting complexity in systems with many such devices. Although the overhead penalty is smaller than if rescheduling were attempted at the task level, this approach leads to obscure run-time behavior.

Accesses in a Static Schedule. Fig. 61 illustrates the benefits of an alternative where the access routines of several devices are interleaved and the processor idle time minimized. The postulated delays are realized by inserting accesses to other devices, making use of the known device access times.

This technique is applicable to both input and output sequences, although in a different way. For reasons of efficiency, static solutions for the device access schedules are preferable. While these are also feasible in the case of input access, output schedules are not statically determined, because they depend on the behavior of the functional components. There, a simple 'dynamic' solution consists of two queues. One of these queues contains output accesses without timing constraints, which are used to realize the delays required for accesses in the other queue. As in the case of input accesses, device access time must be known in advance.

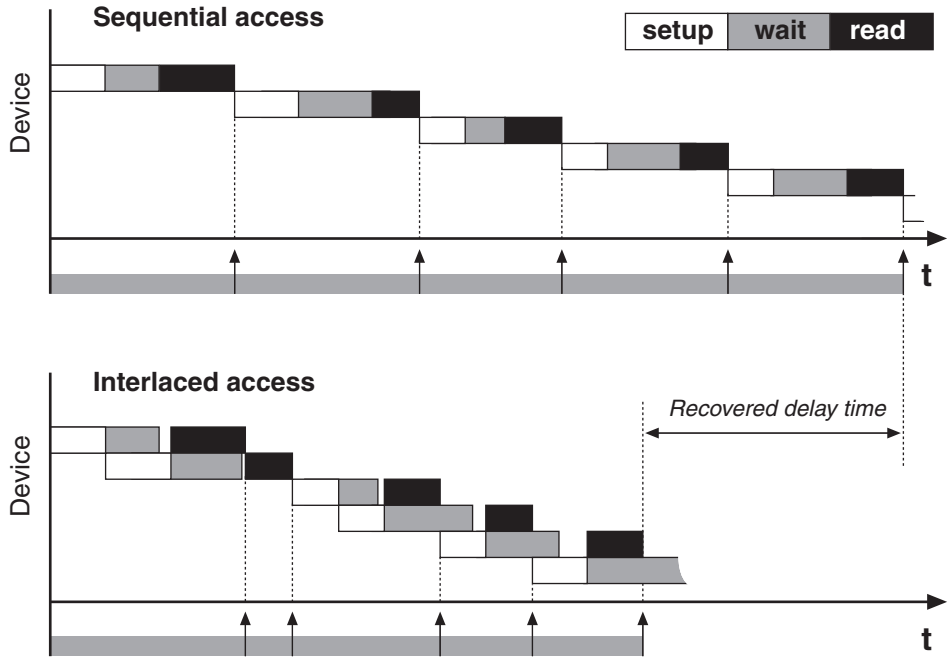

Figure 61: Optimizing device accesses

From a timing point of view, input and output are separate activities, but this separation does not always hold for the devices themselves. Aside from using some of the resources in common (processor, memory, data bus, etc.), certain peripheral devices can deal with both inputs and outputs (e.g., interface modules, motion counters/timers, etc.). If accesses are not serviced with atomic instructions on these devices, exclusion relations may exist between ongoing input sequences and outputs, and vice versa. Therefore, both input and output accesses must be defined in the same specification, which thus comprises *all* peripheral devices present in a system. The code generators are then able to add the necessary mechanisms to prevent unwanted mutual interference.

6.1.1 Representation of Continuous Signals

The internal representation of input (and output) values usually deviates from sensor and actuator data formats. The choice of internal representations is influenced by resource availability (processor performance, memory size, presence of a floating-point unit, etc.), as well as by the physical process and the technology used in specific sensors and actuators (see section 2.1.3). It is difficult to deal with measurement representations in different units (as exemplified by the many conversion-related incidents reported in [Risks]), so a common format is mandatory. Even if not required for accuracy reasons, the use of `float` (e.g., 32 bit) in SI units is advantageous, and is a natural choice in systems with a floating-point unit.

Discrete signals are aptly represented by a single bit, but since these are not available in C (at least not in a universally portable way), bits are treated as members of data Bytes or words. 'Conversion' in this case involves breaking up their original hardware-oriented group membership to create new associations based on functional groups of discrete signals, the masks defined for these groups, and their timing requirements.

6.1.2 Encoding of Discrete Values

Input values from discrete sensors have a binary representation using one bit in a data word, the most compact way of representing information. Still, there are two reasons for rearranging this information. The first is that the original representation is dictated by the interface devices, which unite a number of such signals (considering that these devices service 32 or more discrete signals at once). This results in a heterogeneous collection of signals where not all of its members share the same frequency class (see section 6.2).

The second reason is that it is unlikely that more than one bit has changed when a new set of values is read. Accordingly, there is potential for optimizing the transmission scheme used to distribute such values. The given device representation dictates device accesses, because even when only some of the signals contained in a device are needed in a given period, the device must be read at least with the frequency of its 'fastest' members. However, in a distributed system with common input it is unnecessary to pass on *all* the freshly acquired data to the working nodes.

Therefore, three variants of discrete value encoding and forwarding have been investigated:

1. individual signals are masked in the sender and groups of values are formed according to the frequency class of each signal; further compression is neither required nor possible, and the format is directly accessible in the receiver node (Fig. 62),
2. single data words containing discrete values are sent in compressed form; masking is performed upon expansion by the receiver,
3. entire data words containing redundant bits are sent uncompressed and are masked by the receiver.

Optimum use of the available channel capacity conjectures the re-encoding of discrete signals in groups, corresponding with their specific frequency classes. On the other hand, the necessary compression and decompression in the sender and receiver, respectively, can be detrimental with regard to timing aspects. Actually, a comparison of the three different variants reveals only

minor timing advantages for sending compressed data, but it is superior in another respect: if sampling requirements of signals not belonging to the highest frequency class must be maintained (e.g., if they were postulated to achieve low-pass filtering, which rules out oversampling), the timing advantage is frittered away by the then necessary masking. In addition, forwarding outdated information adds to the ugliness of the third solution.

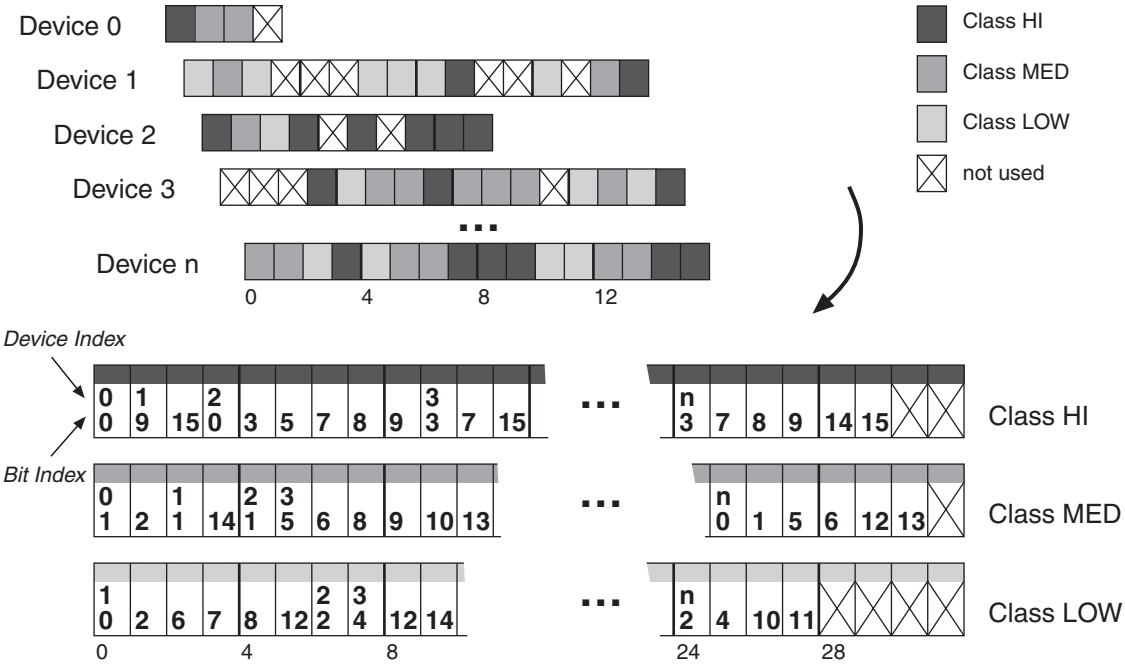

Figure 62: Encoding of discrete values

Value output to discrete actuators is not affected by these considerations. Action messages are translated into predefined bit patterns, which are masked onto a copy of the device output word. This copy is then written to the device.

6.2 Data Acquisition

Newly read raw input values must be converted to their internal representation, verified for plausibility (if possible) and filtered (Fig. 63, left). In a decentralized system, incremental updates of operative input data are broadcast periodically to all working nodes. Events pertaining to a specific node and its currently active event priority map are then extracted decentrally and used to trigger processing within the functional reactive block of this node.

- newly read values are available in their respective hardware formats (raw input data) and must be converted to the system's predetermined standard formats (e.g., physical units),

- the converted values are filtered and, if possible, pass a plausibility check. Data recognized as being faulty is marked invalid. In case a floating-point representation is chosen, invalidity is expressed by assigning the value `NaN` (Not a Number, IEEE standard indicating an invalid floating-point value),
- in distributed systems, the data must be forwarded to the working nodes for use by the event extractor components associated with functional units, and by functional units directly (Fig. 63, right).

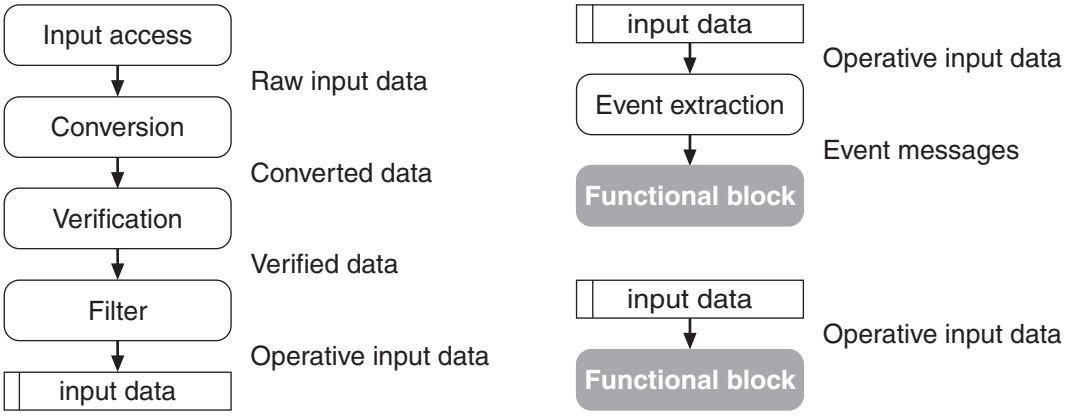

Figure 63: Input data flow

Sampling frequencies of individual signals are postulated depending on properties of the physical processes in the environment, as well as on the demands of the selected control algorithms (see section 2.1.2). A system's base sampling rate is dictated by the signals with the highest timing requirements. This group of signals leaves little margin for variations, and accordingly will simply be read with the base sampling frequency, whereas all the other signals need only be read occasionally. Occasionally means that the point in time when a signal is read can be chosen freely, as long as the signal's demands on sampling frequency and possibly jitter are met.

A seemingly simple solution would be to count intervals to define groups of signals that must be read in one sampling period (which is given by the signals with the highest timing requirements). A group would consist of those signals that must always be read, plus the ones whose counter just expires. Unfortunately, this approach leads to extremely unbalanced resource utilization (Fig. 64). Even worse, unequal processor loads from one sampling period to the other are not limited to data acquisition, instead they afflict all subsequent processing stages. Compared with the average utilization, hardware resources in such a system become disproportionate, if only for the capability to cope with the worst case (arrow in Fig. 64).

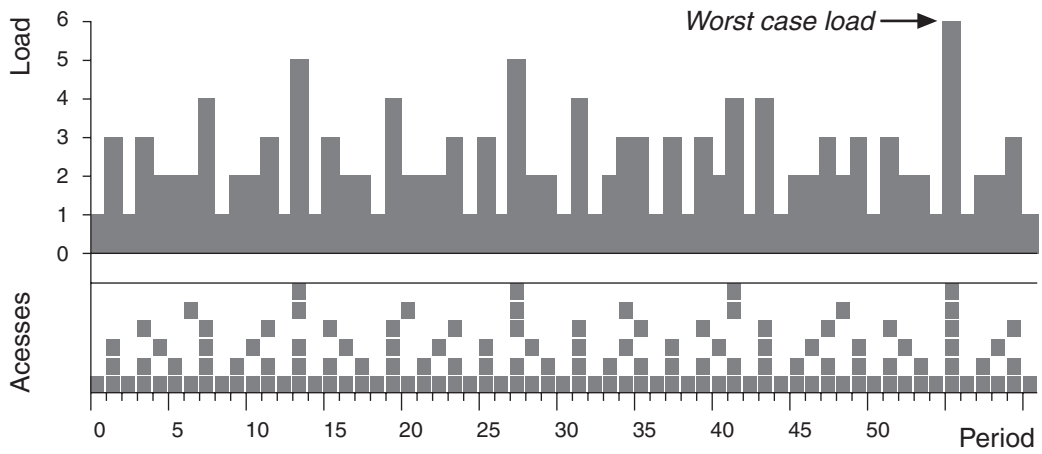

Figure 64: Unbalanced capacity utilization caused by signal periodicity

Therefore, aside from minimizing absolute processor loads, the objective of an input system must be to distribute work loads evenly among all periodic invocations, a demand impossible to fulfill manually. A feasible schedule for the input task as a collection of smaller tasks with precedence and exclusion relations is found statically and leads to a single sequential structure. The scheduler algorithm solves the aforementioned problems: it eliminates possible idle times and ensures practically equal processor loads for each invocation (Fig. 65).

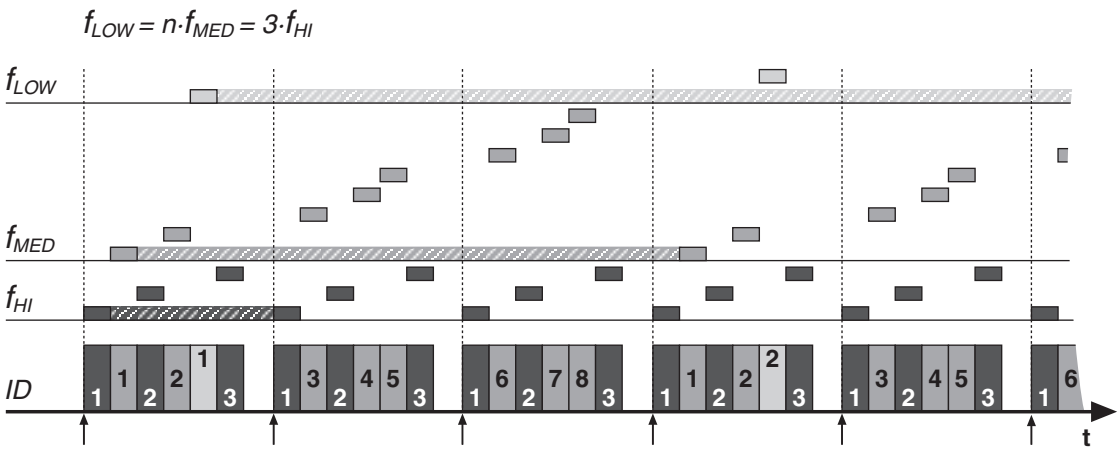

Figure 65: Input access schedule for three frequency classes

Because any changes in the combination of devices or access frequencies require recreating the schedule, this clearly is not a convenient method for manual implementations, although fortunately the affinity to general scheduling problems (as described on p. 100) make it suitable for code generation.

The specification for data acquisition is a high level description of signal properties and requirements (see section A.2). Since this specification deals with peripheral devices, a description in terms of devices, signals, and signal conver-

sions is appropriate. From this data, the generator creates an optimized and statically determined schedule, using a signal's timing requirements and the corresponding device's low-level functions for initialization, setup, access, conversion, verification and filtering. The execution and required delay times of the low-level access functions written in C must be profiled beforehand to enable the code generator's static scheduling. Usually, the time required for any read or write access depends on properties of the data bus, whereas delay time is determined by the devices themselves. In practice, this signifies that the results of just a few access measurements are applicable to a range of different devices, whereas delay times can be obtained from device data sheets. The input access schedule based on actual execution time measurements obtained on the final target itself finds its complement in the performance analysis of the functional implementation (see section 7.1.1).

The advantages of the scheme come at the price of some specification and implementation restrictions, which are in accordance with the chosen time-triggered implementation paradigm (see section 2.1.1):

- sampling jitter is kept out of input components only if there are no interrupts in the system (aside from a basic timer tick)[1],
- all required sampling rates are mapped onto frequencies that are dividers of the base sampling frequency. Further simplification is achieved by assigning all signals to one of several frequency classes.

Assuming three signal frequency classes (f_{HI}, f_{MED}, f_{LOW}), the generated input access schedule consists of a switch-statement with f_{HI}/f_{LOW} cases. The schedule is generated heuristically (see section A.4.5), using the access time of one signal to guarantee the necessary delay time for another signal, as in Fig. 61. If no more access routines are available in a slot, or where they cannot be used because of exclusion relations (when signals must be acquired by the same input device), as a last resort plain delays are inserted into the schedule.

Unsatisfactory schedules and wasted processor time reflect deficiencies at the hardware architecture level, in that too few device accesses are available with which to create mutual delays. For small systems with few input signals there may not be a satisfactory solution. In the example in Fig. 66 (with a frequency class relation $f_{HI} : f_{MED} : f_{LOW}$ of $1 : 10 : 10$), a small collection of input signals with unfavorable exclusion relations causes the accumulation of delay time. The plain delays result in a very low processor utilization of only 42%.

1. In implementations where interrupts must be used for dealing with certain devices (such as network and other communication interfaces), zero-jitter behavior of input components is still available if interrupts are disabled during data acquisition.

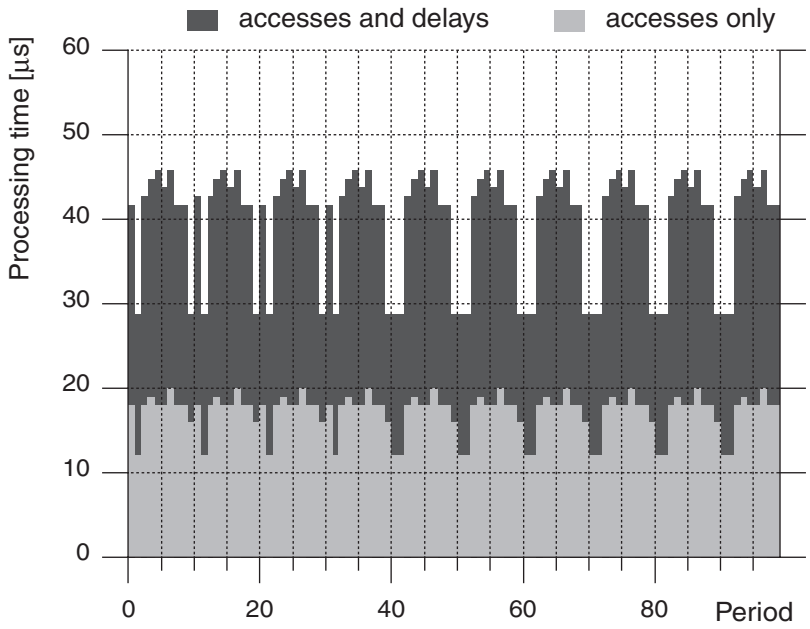

Figure 66: Processor utilization 42% (15 input signals, class HI: 2, MED: 9, LOW: 4)

A more advantageous setting with five times as many signals, including the original ones, makes the required processing time rise a mere 30%, practically eliminating the idle delay time (Fig. 67).

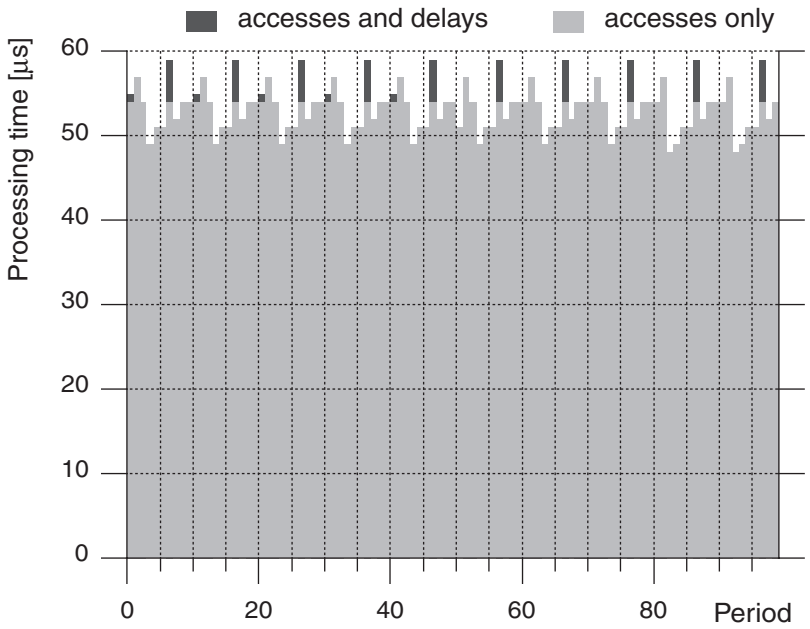

Figure 67: Processor utilization 99% (75 input signals, class HI: 8, MED: 39, LOW: 28)

The example also shows that the input task presents a very uniform processor load and, since the actual execution times were used in creating the schedule, processing time can be given in absolute numbers.

6.3 Input Data Distribution

The task of distributing input data is very simple in single-processor implementations, where it amounts to writing into global memory. In distributed systems, however, communication bandwidth among processing entities often is a costly resource. Because input data acquisition presents an important and recurring load of inter-processor connections, optimizing input data distribution is worthwhile.

On the input node (or, in highly distributed systems, on an additional node dedicated to data conversion and distribution), continuous input values are converted from a hardware format of, for example, 12 or 16 bit length to their internal representation, i.e., the format required by subsequent stages. If a floating-point representation is chosen, it cannot be optimized further by compression. On the other hand, channel bandwidth use in distributed implementations is influenced by a suitable coding of entire data blocks, including discrete values. The justification for choosing a specific method is based on the following definitions and benchmarks:

p: the inverse relation of a signal's sampling frequency to the highest sampling frequency present in the system (f_{HI}), corresponding to the delay in periods in between successive accesses as generated by the static scheduler,

m: the number of continuous signals that must be read in the various frequency classes,

m/p: number of continuous signals that must be transmitted in each period,

w: data width of the converted input values in Bytes, for example for values of type `float` (32 bit, $w = 4$).

Again assuming three signal frequency classes, plausible values (aside from p_{HI}, which is always 1) would then for example be $p_{MED} = 10$, $p_{LOW} = 100$. With a base sampling frequency $f_{HI} = 1000\ Hz$, the resulting lower frequencies thus amount to $f_{MED} = 100\ Hz$ and $f_{LOW} = 10\ Hz$. These values were used in the following comparison, where three different encoding variants are considered:

Variant 'vector'. Package protocol: `float, ..., float, float`

The number of sent `float` values equals the number of continuous signals in a frequency class. Each signal value is allocated space in a vector; this vector is sent in its entirety in each period.

Number of Bytes transmitted per period: $w \cdot m$

Variant 'indexed'.
Package protocol: `index, float, ..., index, float, stopbyte`

Each read value is sent individually. In order to identify it, the signal's input vector index is also transmitted. A stop-Byte marks the end of transmission. The average number of values read in each period is m/p, the index (1 Byte) being added to the actual signal value (w Bytes).

Number of Bytes transmitted per period: $(w + 1) \cdot \frac{m}{p} + 1$

Variant 'masked'. Package protocol: `mask, float, float, ..., float`

Mask size is m bits, rounded to the next Byte. Along with the newly read values a mask is transmitted, where each bit corresponds to an entry in the input data vector. The number of set bits in the mask equals the number of values that will follow in the current transmission, thus eliminating the need for individual indices and the stop-Byte.

Number of Bytes transmitted per period: $\left\lceil \frac{m+7}{8} \right\rceil + w \cdot \frac{m}{p}$

The effects of the presented variants on transmission efficiency when applied to one of the three frequency classes are summed up in Fig. 68 and Tab. 1. Aside from the emerged optimum solutions shown in italics in Tab. 1, in some cases others could have been considered.

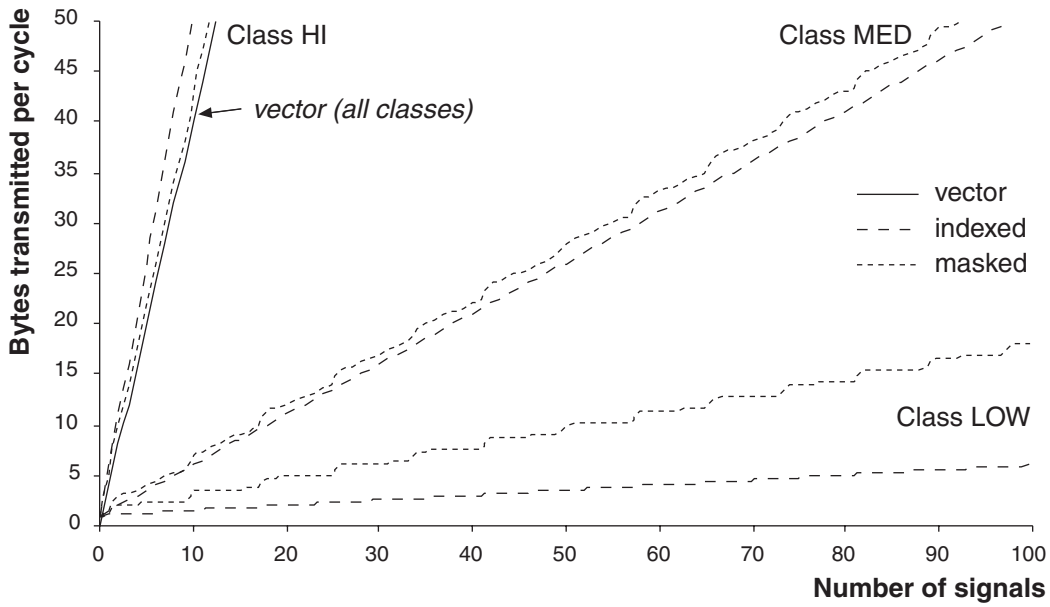

Figure 68: Input data distribution strategies

For reasons of simplicity and ease of implementation two of the variants were chosen and combined into a homogeneous solution: data in the highest frequency class is transmitted as a vector, data of the other classes in index-value

pairs. Because the implementation efficiency of the distribution mechanism is crucial, its particulars are further explained below.

Table 1: Relative number of Bytes transmitted

	Class HI $f = 1000\ Hz, p = 1$	**Class MED** $f = 100\ Hz, p = 10$	**Class LOW** $f = 10\ Hz, p = 100$
vector	*100%*	1000%	10000%
indexed	125%	*125%*	*125%*
masked	104%	135%	450%

The solution centers around a data structure with overlapping regions in the receiver node(s). The operative data vector, i.e., the one that makes input values available to subsequent processing stages, consists of dual-type elements (e.g., `float` and `int` values, both occupying 32 bits). The central part of this vector is shared among the receiver and the global readers, which really are event extractors and functional reactive blocks (Fig. 69). The central part contains the signal values belonging to the highest frequency class, which are transmitted without any further encoding *(HI* in Fig. 69). All the other signals, in many cases a majority, occupy the tail of the vector *(MED & LOW).*

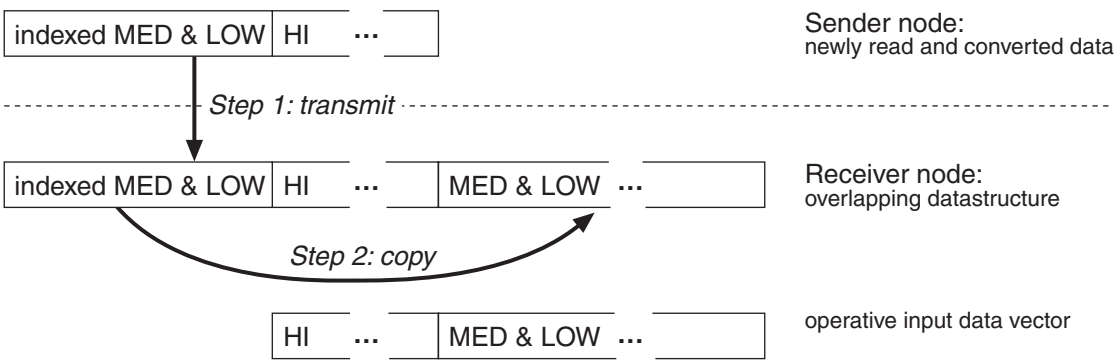

Figure 69: Input data distribution

This part of the vector is never transmitted directly, instead in each new transmission, some of these signals are included in the first part of the vector, together with their respective vector indices *(indexed MED & LOW).* Upon receipt, the new values are copied to their position in the operative input data vector (Fig. 70). The number of positions needed in the 'indexed MED & LOW' part of the vector is small (a typical size is 4) compared to the number of signals of type MED and LOW in the operative data vector. The explanation lies in the even distribution of input accesses with lower sampling rates across all periods in the access schedule.

The required number of positions is (definitions on p. 108): $\left\lfloor \frac{m_{LOW}}{p_{LOW}} + \frac{m_{MED}}{p_{MED}} \right\rfloor$

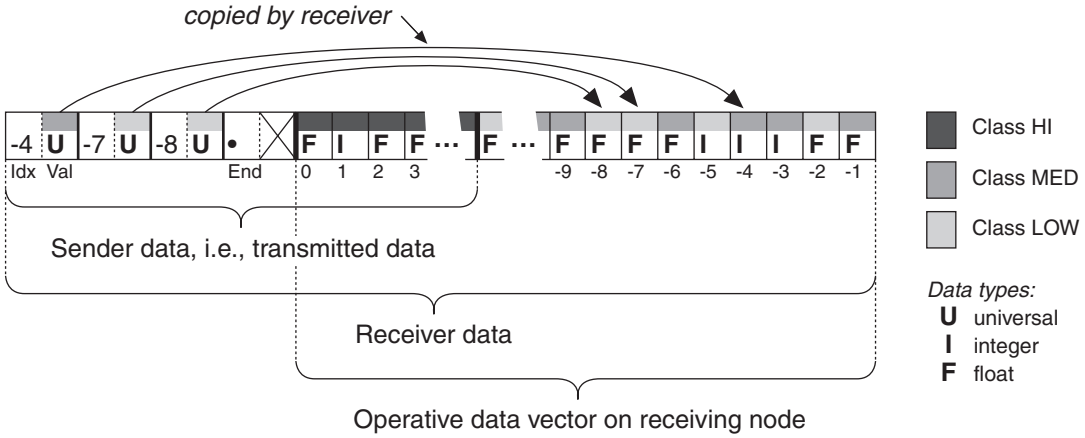

Figure 70: Input data copying

6.4 Event Extraction

The functional reactive block is driven by event messages that are generated when certain indications are discerned in the input data. In other words, events are extracted from the newest input data, which causes the generation of the pertaining event message (Fig. 63, p. 104, right). The tables that will be used by the extractor algorithm are generated from specifications, which contain conditions for each event (see section A.3). Event specifications may comprise time events, discrete events related to binary signals and signal patterns, or events associated with threshold values in quasi-continuous signals.

The connection of event messages with binary signals recorded as such from discrete sensors is straightforward. When the signal changes, one of two possible event messages is generated (separate messages are associated with the transition from zero to one and vice versa). Since events are often associated with more than one signal, predefined signal *patterns* (from the input specification) are used instead of just signals in the event extractor specification. The occurrence of unspecified patterns has no direct effect, but since the change is internally remembered, the previous event message is again generated when its pattern occurs. This behavior is useful if only one of the two events associated with a one-bit pattern is needed in the functional model, like the push of a button.

For the other type of events (i.e., those buried within continuous signals), the situation is less evident. Event messages associated with continuous signals are generated when the signal exceeds or falls below a specified limit. The limits can be set arbitrarily within a signal's range, however, their specification must conform to certain conditions. Specifically, a hysteresis is required to permit

proper switching of the extractor algorithm, which requires limit specifications be made with two values in ascending order (the extractor algorithm is given in section 7.1.2). If an event message of only one transition is needed by the functional model, nonetheless both limit values must be specified, the unused one marked as `VOID`. The specification for the example in Fig. 71 is on p. 193).

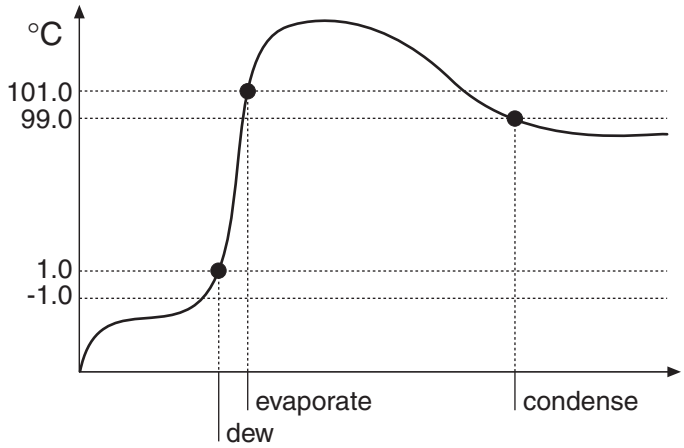

Figure 71: Events in a continuous signal

The event extractor maintains the correct ordering of subsequent events, i.e., as long as the information transport from the processes in the environment maintains the continuity of events, no arbitrary message sequences will be produced. Furthermore, the event extractor is also capable of switching between a number of different priority schemes at run-time. The currently active scheme is chosen from within the functional model, where all information regarding system status and desired system behavior is available.

A close correspondence of event extractor components with their associated functional components results from the tight links of events and their event messages with the behavioral context model (see section 5.5.1). On the other hand, the actual task of extracting events can be seen as an abstract activity, recurring in every conceivable system. Event extractor specifications therefore need only combine signals, their patterns and thresholds, with event messages.

6.5 Output

System output is generated when functional blocks issue action messages. These messages are converted, queued, and output before the current sampling period ends (Fig. 72). In distributed implementations, several functional blocks may be located on individual nodes, so their action messages must first be transferred to and collected in output nodes.

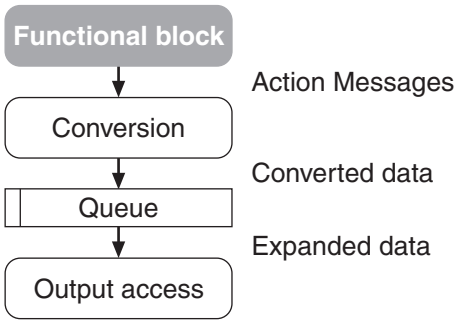

Figure 72: Output data flow

Contrary to the input, system output load cannot be distributed statically, as the number of outputs depends on the specific transition structures encountered in a cascade of the functional model (see section 5.5.3), as well as on the occurrence of multi-stage hardware accesses. On the one hand, multi-stage output is needed for the same reasons as for inputs, namely because a peripheral device must first be set up before output data can be written. On the other hand, a single action message may itself expand to a sequence of different output accesses. Consider the example of a positioning actuator. A new position of the actuator is selected by issuing a single action message with the position as an attribute, although the actual setting of the position consists of a number of separate steps: preparatory trajectory data include position and acceleration parameters, followed by commands to initiate the actual movement. While the behavior could be described explicitly as part of the functional model, uncluttering the functional model from this type of protocol is desirable.

Expanded output accesses are written to one of two queues. The first queue contains accesses that may be written directly and as soon as possible. Accesses with timing constraints, which in this case are delays that must be maintained between write accesses on a device, are written to the second queue (Fig. 73). While both contain the previously measured access time, the queue for delayed accesses also holds the delay relative to its predecessor element. Element insertion, which involves a linear search and moving of the elements in the rear, is usually not a problem since the queue is very short. Its length mainly depends on the number of items that cannot be dealt with in the same processing period that generated them, i.e., accesses with long delays. Because these should be handled explicitly in the function model, there is a limit on maximum delay time in the specification.

Both queues are emptied by a simple output run-time scheduler. The delayed queue has priority as long as its front element can be deployed directly, else delays are accomplished by inserting direct accesses (Fig. 73, right).

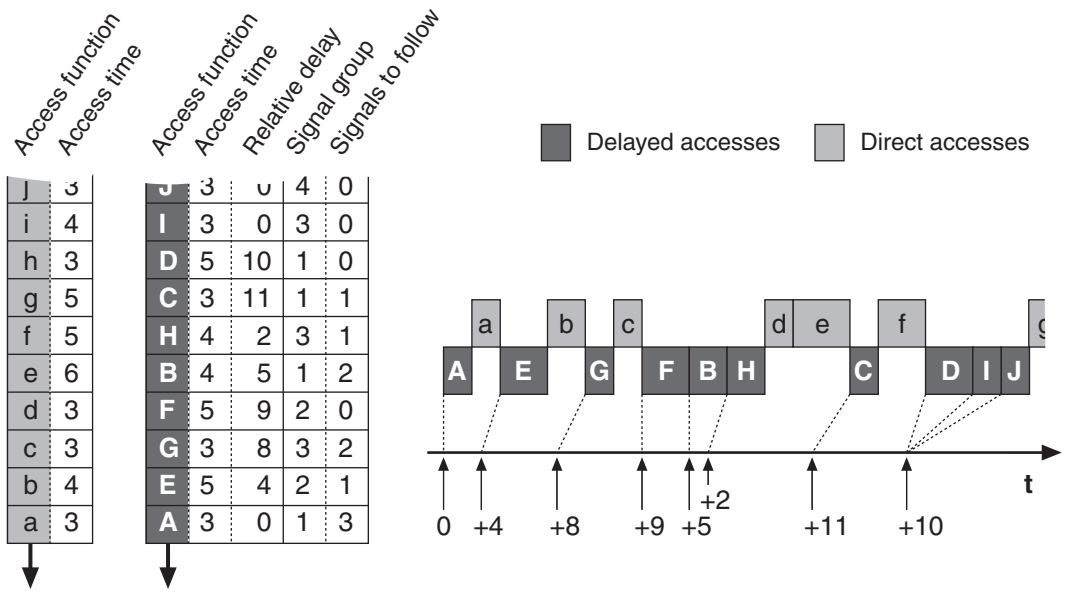

Figure 73: Output access queues and schedule

Stubs for calling the conversion and/or expansion routines from the functional model are generated using the same hardware-oriented specification as for the input schedules, the profile data in this case also including information on access expansions if required (see section A.2). With this information, together with the set of action messages from the functional model (see section 5.4) upper bounds on the processing demands of the output task can be found. A static table with the relevant values is computed for use by the run-time scheduler.

6.6 Communication

Since no conversions or any other modifications are required, the communication interfaces amount to the simple queueing and forwarding of high-level communication messages to and from the functional block (Fig. 74).

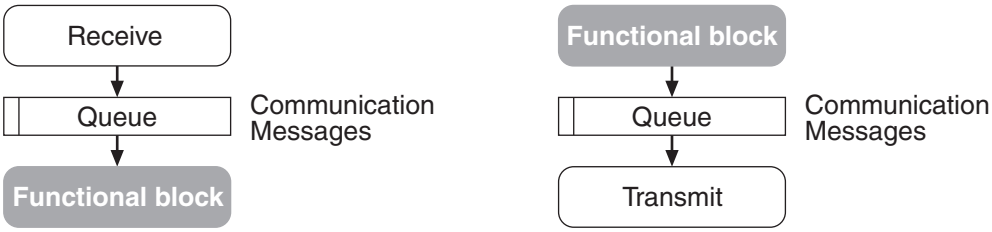

Figure 74: Communication data flow

7 Implementation

The need to ensure coherence between high-level specifications and executable code has been a long-standing obstacle to applying models and specifications in the area of real-time embedded systems. The reason is that most of the time the step from abstract descriptions to implementations is not taken automatically, at the least it involves manual intervention. Because in general, implementations contain much more detail than is captured by specifications, practitioners are reluctant to abandon a program after careful adaptation only because some part of it needs updating after changes in the specification. Instead of generating a new version, they adapt the program manually to the changes, which leads to programs gradually diverging from the high-level descriptions.

Therefore, the high-level source baseline ought to be the *only* description used in the development of the parts of a system that can be described on a high level of abstraction. To be useful, it must be linked to the lower levels in the same way as program source code is linked to object code, namely by automatic translation that is complete and does not require manual adaptation. This implies also that manually written parts remain untouched and that the execution behavior of the implementation at the low level is maintained, which is only possible if the generated code complies to definite properties that predetermine its structure and execution behavior. These properties are independent of the particular functionality called for in the high-level descriptions.

On this prerequisite, variations in the generated code that are required for simulation and for various target environments are composed automatically from the same configuration, with the component architecture assuring interface consistency of any partial implementation with its complete counterpart. The code generators yield code that can be implemented as three distinct tasks with run-to-completion behavior, either together on a single processor or separately on different processing entities. On a single processor, they operate in sequence

in every period. The input task with its uniform execution time on each invocation presents a constant load, while the time demands for the processing task (which includes the event extractor and the reactive CIP machine), as well as for the output task depend on the occurring events and hence the cascades that must be run through. Performance data obtained by measuring execution times of the functional code itself is used in the local extraction-processing cycle to make best use of the available processing time in each period (Fig. 75).

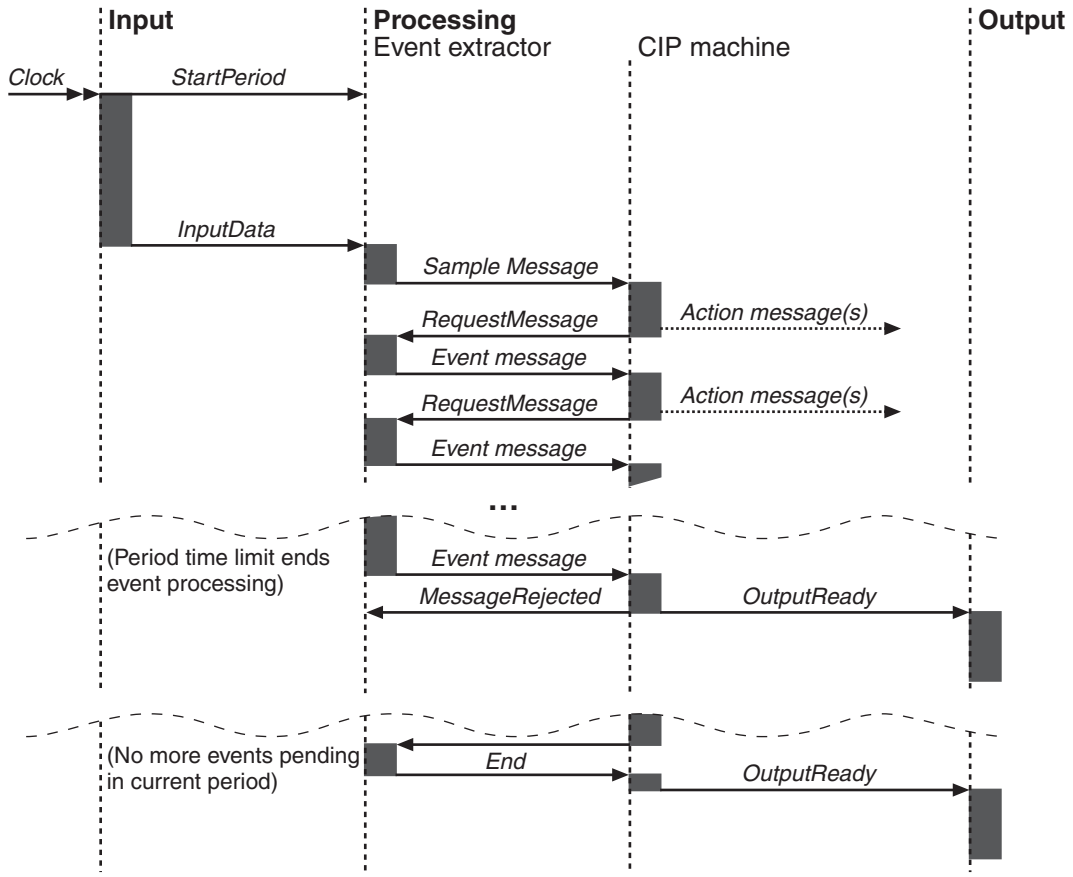

Figure 75: Execution sequence

An execution sequence proceeds as follows: the input task is triggered by a clock pulse with a period conforming to the highest sampling rate specified (see section 6.2). At the start of each period the processing task's local clock is reset. After the input task completes accesses and data conversions, control is handed over to processing. Processing deals with single event messages, each of which is requested from the extractor algorithm. This is repeated until the remaining time slot has become shorter than the worst-case execution time associated with the latest extracted message. Then, the extractor is notified and control handed over to the output task. The same happens when all events have been processed *before* processing time expires. In both cases, the output task then proceeds to convert and output the previously buffered output messages.

7.1 Code

The production of source code for real-time systems is practically always undertaken with language subsets, non-regarding the actual choice of language. Even when not enforced by regulations of the application domain (cf. the guidelines for the automobile industry [MIS98]), the use of language subsets is a necessity in projects with multiple contributors. For languages such as C with its notoriously long list of flaws, the definition of a less precarious subset is most important (see section 2.2.3). It becomes even more so if code written by hand and generated code must be combined into coherent applications with deterministic properties. The major stumbling blocks are dynamic data structures, recursion, and loops with possibly unbounded behavior. These features are not permitted, which preserves well-defined execution sequences, a precondition for automatic profiling.

Restrictions in the use of a language by barring all unnecessary constructs from the toolbox is also a valuable means to enhance program comprehension, an especially rewarding measure in embedded systems with their characteristic time-periodic execution.

Memory. Memory is allocated statically, because dynamic memory management (for example using the `malloc` routine built into C) implies several risks. Allocations can fail, and being able to deal with allocation failures in a running system increases complexity. Aside from possible memory leaks, allocating and freeing leads to memory fragmentation, which is the cause for the allocation routine's non-deterministic behavior.

The static approach allows allocation, but not freeing, whereby allocation is performed solely during initialization. If allocation is performed only once, a simpler routine without the overhead of block headers can be used instead of a language's normal allocation routine. Another advantage is that this routine can be disabled after completing the initialization.

7.1.1 Functional Code

The code that is generated from functional CIP models is efficient and deterministic. It is built as a collection of nested switch statements, which reflects its genuinely static properties, or in a table-based variant that introduces the separation of code from data. The code is truly reactive in that it remains passive until one of its input functions is called. Upon such a call, a complete cluster transition is performed, after which the reactive block returns to its inert state. Aside from a number of housekeeping functions (e.g., those needed to advance

the functional model's own delay timers, or to perform read operations from internal channels), the interface provides one specific function for each input message specified in the model. Output functions must be defined in the environment and bound to respective function pointers during system initialization. The functional block refuses to run if any of these are left dangling.

Processing time is present neither in the functional model nor in the connection specifications. It is only by examination of the generated code that the performance on a specific target can be evaluated. Exhaustive performance measurements, which are extremely difficult to achieve with non-restricted code, get easy when the legion of possible paths through the code is limited. Because the performance profile includes *all* possible execution paths of the deterministic code, the procedure gives the necessary assurance that an implementation does perform according to the requirements.

If these measurements reveal processing bottlenecks, a way out can be sought by distributing the work load onto different processors, or by softening the requirements on processing associated with certain events. In this case the measured data together with the event priorities permit quantitative statements regarding the performance degradation. It must be emphasized that exhaustive performance measurements on the final target are far superior to the vague assurance that can be obtained from the combination of timing assertions in a high-level model and (inevitably) partial testing.

The generated code is compliant with the postulated restrictive implementation demands, i.e., parallel structures present in the functional model are cast into a single sequence with deterministic behavior. Aside from generating the modules that implement the model's functionality, a more abstract representation of the entire model can also be produced. This representation can be used for creating other types of implementation, such as the table-based variant employed in dynamic reconfiguration (see section 7.3). Performance analysis is also based on the same representation. Each specified event message sits at the root of possible paths through its pertaining cascade, for which a collection of code sequences (minus the action messages) is defined. This code is the executable counterpart of a unit's interaction trees (Fig. 58, p. 96). The worst case execution path included in this collection must be found by measuring processing times on the target itself. Another excerpt of the abstract model lists only the sets of action messages belonging to each event with reference to the same possible paths. These are used to determine upper bounds on the processing needs of the output task. Aside from employing the data thus obtained to verify that and how well an implementation will fit on a specific target, the same measurement results are also used by the processing task's run-time scheduler.

The functional code structure is explained using the motor controller example introduced in section 5.3.2. Its implementation unit (containing a single cluster) accepts event messages and is capable of producing action messages as shown in Fig. 44, p. 83. The permissible traces in the input stream are determined by the structure and interoperation of the receiving processes.

The transition structure of the example's *Controller* process (Fig. 51, p. 88) shows the motor being either in state *off, running,* or *stopping* (the motor runs and stops under continuous control). Its transition triggers are the *switchOn/Off* pulses from the switch process, the *Standstill* and the periodic *Sample* event messages for the controllers from the environment. Its outputs consist of pulses *spinup, spindown, standstill,* to the other processes and action messages *MotOn, MotValue, MotOff,* to the environment.

Two different controller algorithms are placed within the transitions triggered by the *Sample* event and associated with states *running* and *stopping,* respectively. It follows from the diagram that only one of these controllers can be active at any time, thus effectively limiting the required processing requirements. The transitions leading to these two states contain the initialization parts of the controller algorithms.

The time demands for processing the event messages in the motor controller example show the processing limits of a specific implementation (Fig. 76). The periodic controllers triggered by the *Sample* event message present a recurring load in each period; they are scheduled first. If the worst case paths are taken, only one additional event message can be handled during a single period. Considering the time scale and the improbability of these events arriving at the same time (i.e., in the same sampling period), this may still be acceptable.

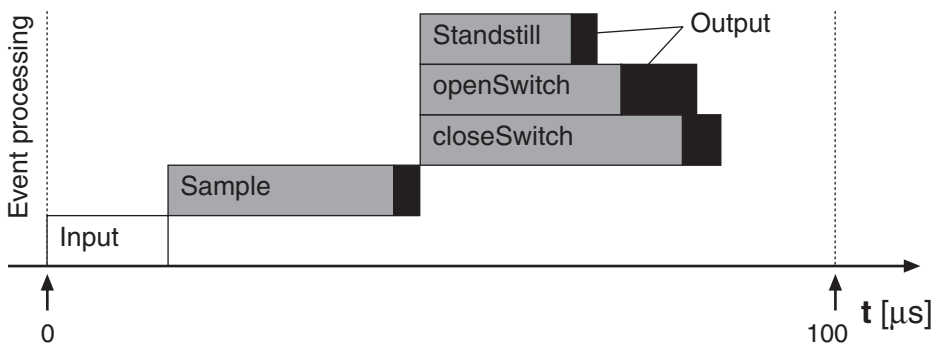

Figure 76: Motor controller: processor time demands

7.1.2 Connection Code

Input. The static access schedule for system input is based on the relation of the chosen frequency classes (e.g., f_{HI}, f_{MED}, f_{LOW}) and implemented as a switch-statement with f_{HI}/f_{LOW} cases (see section 6.2). Another switch-statement of the same size is generated for converting, filtering and verifying the newly acquired raw data. The two switch-statements are independent functions, so they can either be deployed sequentially in a single task, or in different tasks if required by specific implementations. The timing behavior is statically defined and known when the schedule is created, since the schedule itself builds on the time demands of individual accesses (the performance of the low-level access routines must be measured beforehand and included in the access specification, see section 6.2).

Event extraction. The event extractor is an efficient implementation of a table interpreter. Since it deals with two kinds of events that are either associated with discrete or with continuous input signals, it consists of two parts (Fig. 77). Each part has two linear data structures, one containing the static information (i.e., event message function pointers and their associated patterns or threshold values), the other holding information about the actual state (i.e., the last discrete data value seen at this input with the specified *Signal_ID*, or the region a continuous signal is currently in). A group of entries in the continuous table is delimited with the smallest and largest numbers representable in the chosen format (-∞ and ∞ in Fig. 77, `-FLT_MAX` and `FLT_MAX` in the case of `float`).

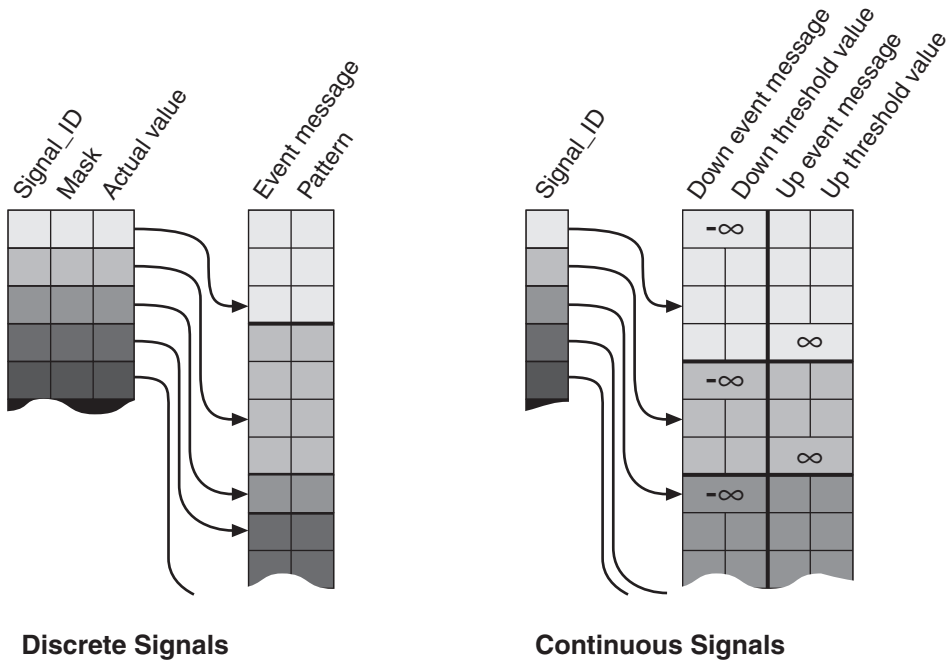

Figure 77: Event extractor data structures

When the event extractor runs, the list of discrete events is searched sequentially until a change of signal value is detected. An event message is generated, if the new value matches one of the specified patterns. For events that depend on continuous signals, two comparisons are needed (one against each of the threshold values limiting the region the signal has been in so far, see section 6.4). Aside from the *Sample* event message that always has the highest priority, event priorities are based on their location in the active list (with the list order defined in the event specification).

Output. System output is asynchronous and performed in several stages. In a first stage, action messages originating from the functional model are buffered locally to allow for possible asynchronicities of the output system. The majority of output signals results in a single hardware access, but sometimes expansion to several accesses with time lags in between may be necessary. For this reason, in a second stage two different buffers are provided (see section 6.5). In the case of a simple output signal, a plain FIFO buffer is used, while for multi-stage accesses, an ordered queue with relative delays is necessary. As in the case of input access schedules, delays are realized through insertion of accesses to other devices, though in this case, a simple dynamic technique must be used. The schedule is produced by reading from the two buffers in a round-robin fashion, either until they are empty, a new timer tick occurs or, in case of the second buffer, a first element is encountered with a delay time that has not yet elapsed.

7.2 Deployment

In many embedded applications a collection of singular tasks has replaced monolithic programs with the advantage of smaller scale and simpler structures. Modularity and dynamic behavior at the task level should permit system compositions involving new parts as well as legacy code. The drawbacks of these techniques in hard real-time systems are countered with more efficient hardware in the case of speed considerations and testing in the case of unpredictability concerns. In the more cost-sensitive embedded systems segment the claim for additional resources is not well received, because it is mandatory that these systems use the slowest possible processor and a minimum of memory. For these applications, progress made in hardware development must result in lower cost or reduced power consumption, rather than compensate implementation deficiencies. Predictable response times, vital in safety-critical systems on the other hand, are hard to achieve without static methods using components with deterministic properties.

7.2.1 Scheduling

A task's workload is characterized by its execution time. If the workload will be used for scheduling purposes, it is the worst-case execution time that must be considered. This, however, is a pessimistic estimation or measurement. Although the pre-deployment guarantee afforded by such approaches is highly desirable, most of the time the required processing time will be inferior to the worst case, affecting scheduler efficiency and thus leading to poor processor utilization. This is especially true when the difference between the worst case and normal execution time is great. Because it is difficult to measure worst-case execution times accurately, deadline violations must still be detected and handled in those systems where scheduling is based on these.

Dynamic Scheduling. Dynamic scheduling deals with tasks in an arbitrary sequence and unknown start times, where the various tasks operate independently and where interactions are not clearly defined [LiL73]. Solving the interaction problem is postponed and these issues are handed down to a later stage (i.e., run time). All types of dynamic scheduling are punished with considerable overheads aggravated by conceptual difficulties such as resource depletion and priority inversion; besides, their essential shortcomings with regard to satisfying timing constraints have also been shown [XuP93]. This is especially true for preemptive techniques that create an infinite number of possible schedules for a given problem and introduce additional conflicts with regard to resource usage.

Static Scheduling. Static scheduling, on the other hand uses detailed prior knowledge of the tasks' processing requirements to find a feasible schedule [XuP93]. While this technique can guarantee the correctness of its solutions, it is overly pessimistic regarding asynchronous events, which tend to be scheduled too often and therefore waste processing power. Even an inadvertent combination of both methods cannot do away with this disadvantage, as truly asynchronous tasks still have to be scheduled many times only to verify that the associated event has not yet occurred.

In both these approaches, input and output activities are implemented as tasks and scheduled in the same way as computation. Generally (except for exclusion and precedence relations), no assumptions about the nature of the tasks actually running on a target are made. Since such knowledge is available and the tasks' code confirms to strict limitations, a combination of static and dynamic scheduling techniques is possible that allows for high processor utilization, at the same time guaranteeing low input jitter and predictable behavior (Fig. 78). The overhead of a complicated scheduler is avoided, because there are only three basic tasks. No preemption is used and the remaining dynamic decision

within the processing task involves a mere table lookup that hardly introduces any overhead.

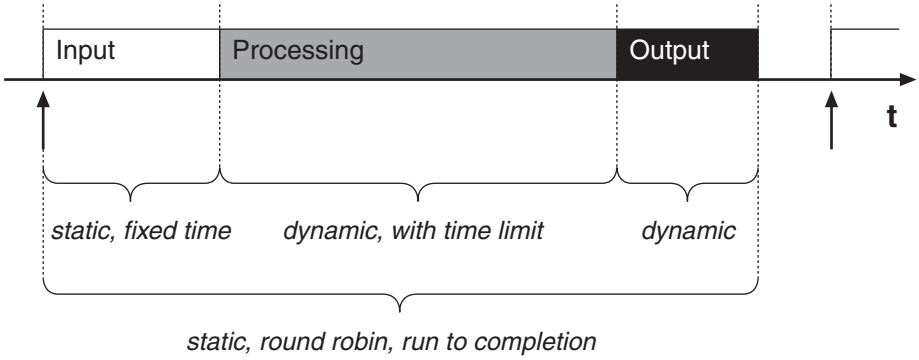

Figure 78: Global and local schedules

Run-time Executive. Even for a minimum implementation with three different tasks, some kind of a real-time, multi-tasking operating executive is needed to keep track of available system resources (CPU time, memory, etc.), and allocate or release them on demand. Its job of controlling and coordinating activities is greatly facilitated when the tasks present in a system cooperate, implying that

- their execution time demands are bounded and deterministic, i.e., once scheduled, they can be allowed to run to completion,
- they can (and must) be deployed in a prescribed sequence.

On this condition, a simple executive is sufficient to run the well-coordinated code segments produced by the generators. Because all tasks in the system are allowed to run to completion, a dispatcher based on non-preemptive round robin multi-tasking is appropriate. With this type of scheduling, the executive can be kept small and highly efficient, and it is perfectly possible to develop a custom piece of software. This approach avoids royalties in high-volume commercial applications and makes the executive's source available for verification. Furthermore, in systems with limited program storage space such as 8-bit microcomputers, small code size and a simple call interface are essential.

7.2.2 Processing Scheduler

When the input task yields the processor, the event extractor generates the *Sample* event message for those quasi-continuous processes running at the highest frequency available in this system. From then on, the processing task repeatedly requests single event messages from the extractor. Each message triggers a complete cluster transition in the functional reactive block, and for every message there is a worst-case interaction tree that it may trigger. The scheduler

exploits the fact that worst-case behavior is the exception and message processing will usually not exhaust its allotted time slot. The processing task keeps track of the current time and also records the time demands of all pending outputs, i.e., the ones associated with ongoing event processing. Whenever a new event has been extracted, the expected processing time in the table is compared to the remaining time balance of the current period. If it conforms, the event is fed to the reactive block and the procedure repeated until an event occurs whose execution would not fit any more. If completion cannot be guaranteed, processing of a message must be deferred to the next period. It is important to note that in this case the event message is withdrawn and not put in a queue, because this particular event may not be present in the input data the next time the extractor runs (this characteristic, combined with sufficiently low sample frequencies can also be used as a low-pass filter, e.g., for input key debouncing). Not having a queue for events at the same time effectively prevents processor overloads and is also a prerequisite for switching between different event priority lists.

In the period shown in Fig. 79 (using measurement data of Fig. 76), processing for the event message *Standstill* would not fit into the remaining time budget in the current period and is rejected. This type of scheduling is achieved with practically no overhead, and it is worthwhile because worst-case execution times are hardly ever exhausted. Thus, in reality more events can be dealt with than predicted by the static analysis. The slack time nevertheless remaining can be exploited by non real-time tasks, which must be preemptable.

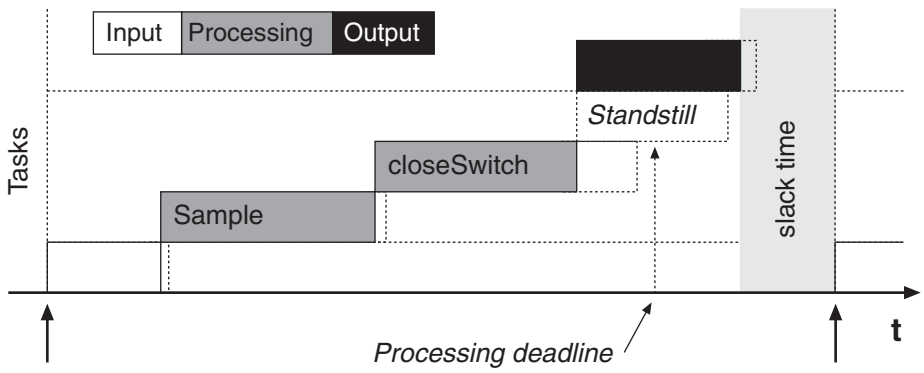

Figure 79: Processing task schedule

7.2.3 Implementation Variants

During development, different kinds of implementations are required to meet the demands of simulation and partial target installations. If these can be constructed from a single set of functional models and connection specifications,

the task of maintaining coherency of versions and variants is greatly facilitated. The same capability helps in choosing an implementation architecture adapted to the performance requirements towards the end of a development. Performance, hardware availability and cost considerations can thus be resolved without having to settle early on a specific hardware topology. Basically, all variants fall in one of the categories shown in Fig. 80.

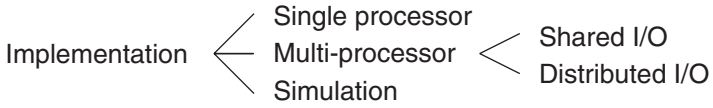

Figure 80: Implementation variants

A complete implementation involves the following tasks:

1. clusters, the possibly parallel components of the functional model must be assigned to implementation units; generated code consisting of the functional reactive CIP machine within a CIP shell is then assigned to a specific task,
2. the worst-case execution time for processing associated with each possible input message must be measured on the intended target and tables compiled for use by the run-time scheduler,
3. code for attaching functional reactive blocks to the peripherals must be generated from the connection configuration, thus including the low-level access functions or macros previously tested and profiled,
4. buffered communication mechanisms must be established between functional processes on different processors,
5. event extractor tables must be generated from the event configuration.

Target Implementations. In single-processor implementation, the three basic tasks are scheduled sequentially, according to Fig. 78. Inside a main loop that executes periodically with the system's highest sampling rate, both the processing (see section 7.2.2) and output tasks (section 7.1.2) perform their own loops (Fig. 81, left). Even a dual-processor variant illustrates the benefits of distributed implementations, in that processing and output may overlap (Fig. 81, right).

In a distributed implementation, it is costly and inefficient to fit every processing node with its own data input and output mechanism. With the component architecture set up accordingly, these nodes can share input and output devices based on particular nodes that handle peripheral data input and output for the entire system. Generally, multi-channel devices can be used more efficiently by dividing the available channels among various functional components.

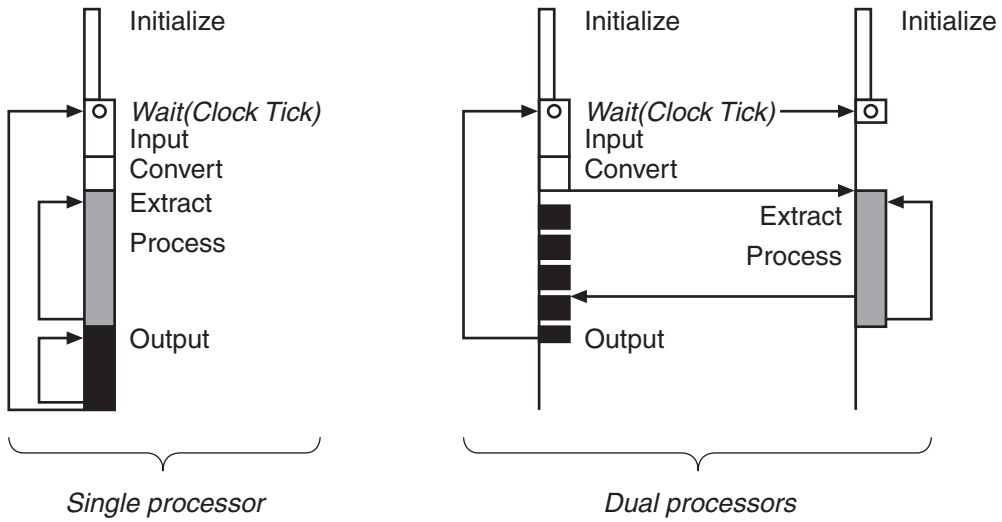

Figure 81: Single and dual processor implementations

Simulation. A special type of implementation is required for simulation, e.g., using Matlab/Simulink [MAT99]. It is marked by the need to incorporate the entire control system into a single function, even if the target system will be distributed. While the academic conjecture of using vectors to convey input and output values holds nicely, there are a number of other problems that must be addressed. Namely, if stiff integration solvers are applied to the simulation model, simulation time must be expected to run backwards. Therefore, input values need to be buffered before they are fed to the event extractor, or else the context models in the functional model could get out of step with the simulated reality, leading to context errors. Likewise, all output values must be updated even when no changes occurred since the last call. This is handled by additional functions that are only present in the simulation implementation (Fig. 82).

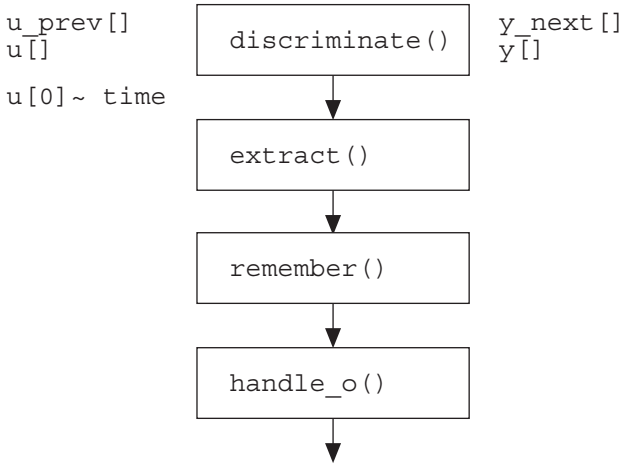

Figure 82: Simulation stages

The goal of a simulation is to validate as many parts of the final implementation as possible. It is thus important to use *exactly* the same code that will be present in the later target system. For the same reason, input values, which would *all* be available in each new 'sampling' period in the simulation, must be sampled using the same intervals chosen for the final installation. Therefore, the extractor algorithm and the tables it uses are also present in simulation implementations.

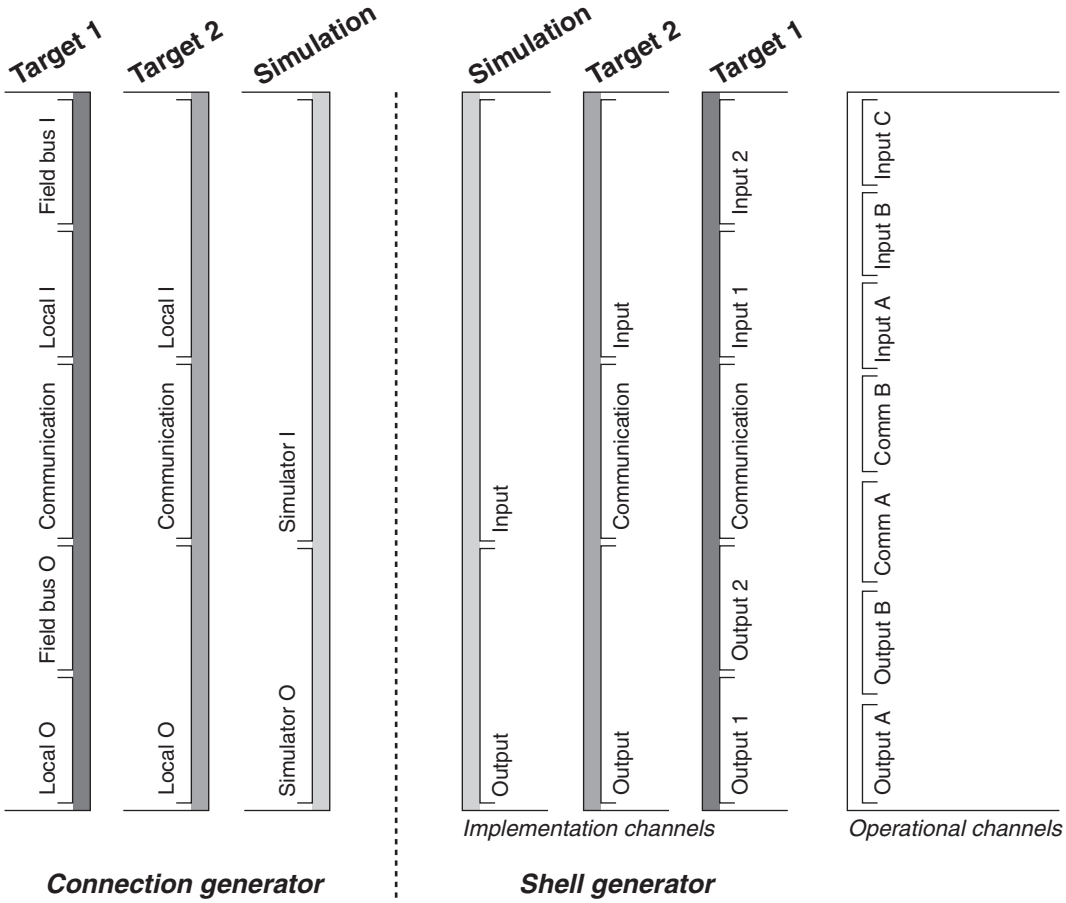

Figure 83: Interface implementations for different modes of operation

Identifying Suitable Implementations. To identify suitable implementations, the following steps should be taken:

1. choose a set of envisaged implementation architectures (which must be contained in the component architecture); example variants are shown in Fig. 83,
2. for each variant:
   - generate complete implementations,
   - measure the timing performance of the worst case paths in the functional block,

- establish performance data of input, conversion, event extraction, and output (i.e., use data calculated by the generators for input and output, and perform timing measurements of conversion and event extraction),

3. compare measurement results with the timing demands set forth in the requirements and choose the optimum variant.

7.3 Dynamic Reconfiguration

Dynamic changes of configurations while systems are up and running have become commonplace in general purpose applications in a workstation environment. Today's workstations can adapt themselves to the changing needs of their human users with minimal delay and yet maintain performance requirements. In such reconfigurations new behavioral mechanisms become operative without affecting the currently active behavior. The flexibility achievable by dynamic reconfiguration ranges from creating and destroying tasks on a single processor to reconfiguring entire task assignments in distributed systems. Despite the potential of dealing dynamically with functional entities that is offered by many modern operating systems and programming languages, these possibilities are only reluctantly applied in the field of real-time embedded systems. The reason is that dynamic methods entail the use of techniques that are adverse to the foremost goal in real-time systems, namely deterministic behavior. Still, these methods would be advantageous especially in systems that perform under a number of distinct modes of operation. Examples of this can be found in flight control, where ground handling, take-off, cruise, and landing represent the different modes, or in communication systems that in one mode of operation service a large number of connections exclusively, or dedicate a part of their resources to administrative tasks under low load conditions in a second mode. Since in purely static solutions the inactive parts unnecessarily block resources, dynamic reconfiguration may be indispensable for economical or technical reasons.

7.3.1 Static Approach

Reconfiguration has two main areas of application. One is concerned with the on-line evolution of systems that must be available over long periods, while the other type of reconfiguration attempts to optimize resource utilization. Early attempts of the second type were mainly motivated by hardware limitations, a prominent example being the first control system for the Space Shuttle [SpG84]. In this system, code overlays were loaded from tape to fulfill the

needs of specific modes of operation that correspond to the different phases in a single mission. Still today, optimal resource utilization is achieved through a number of predefined configurations that form groups of functional components needed in specific modes of operation. Reconfigurable systems in this sense, with or without the inclusion of programmable hardware, consist of optimum algorithms within an adaptive component architecture. Systems with these properties still run in situations where resources are tight, and/or they perform better than the set of only partially active algorithms within a fixed architecture. The switching between various configurations, while preserving consistency of the application data, presents an additional charge compared to the routine obligations of real-time systems. Therefore, advantages can only be expected from careful implementations.

Configurations are commonly defined explicitly, using dedicated languages. These languages deal with components and their instantiations, but abstract from component behavior. Because there is no connection between the level where the various configurations are assembled and the lower application domain, an additional step is required to identify dependencies and analyze consistent configurations off-line [FeL98]. This step may introduce inconsistencies caused by redundant information in the functional model and the configuration definition. Moreover, it is very difficult to manually find consistent configurations that satisfy the performance requirements of all tasks. This problem can be evaded when a system's functionality is described on an abstract level and configurations are based directly on this functional model – where they are implicitly present anyway.

Dynamic reconfiguration can be implemented either using features offered by an operating system or from within a programming language. If measures are taken at the language level, they need to blend with memory management in general and hence become dependent on implementation details of the language concerned. This may lead to memory fragmentation and non-deterministic behavior, both major problems in systems that are required to run over extended periods. In practice this situation is avoided by the sole use of static memory assignment, or with preallocated memory pools of a fixed size. On the other hand, dynamic changes of system behavior that are achieved through an operating system presuppose the use of dynamic scheduling techniques. In these techniques, a setup's schedulability is based on static analysis. If the assumptions made for the analysis do not hold at run time, system behavior becomes non-predictable and timing constraints may be violated.

In the presented approach, implementation variants are built into the code generators, with the abstract models remaining completely untouched. One

such variant is concerned with the mapping of functional modes of operation to the varying configurations of an implementation. The goal is efficient use of the primary resource memory while maintaining deterministic behavior and optimum performance.

7.3.2 Reconfiguration Strategy

The reconfiguration strategy is built on the information about distinct modes of operation present in the functional model. This approach evades inconsistencies between the functional model and explicitly specified configurations, and it is also more efficient than low-level approaches that disregard the functional model entirely. Still, the two primary goals can be reached:

1. only code of currently active components is running, and
2. temporarily unused data is eliminated to lower net memory requirements. If a rise of resource demands in one component is complemented by a decrease in another, reconfiguration immediately benefits an entire system.

Static Code and Dynamic Data. Dynamic reconfiguration aims at extending mechanisms that optimize processor utilization to also use memory as effectively as possible. Given the static environment and the necessity to avoid the use of memory management at the language level, a promising option is to separate code and data wherever possible. In the running system, the code will remain permanent, whereas data tables can be adapted to the current mode of operation. To this end, the nested switch statements of the functional code are given up in favor of tables and an interpreter engine. Separate tables are generated that correspond with specific cascade subsets and hence with the demands of the pertaining mode of operation. These tables can be handled dynamically at run time. As has been noted, the connection implementation must remain stable where it is concerned with the hardware environment. Thus, even in implementations with different input schedules (which optimize processor utilization), only the event extractor tables are suitable for reconfiguration.

Compact Storage for Off-Line Tables. Fields of the state machine tables have the types most adapted to their use by the state machine interpreter, i.e., `int` and pointer types. It is not necessary to store temporarily inactive tables in this format. Instead, tables not permanently required are stored in compressed form and loaded only upon entering the relevant mode of operation. The compression is achieved by exploiting the fact that the name spaces of most table fields, i.e., processes, states, transitions and so forth are small and occupy only a few bits. It would be possible to access data in the compressed tables directly during execution, but only at a great cost. Because the tables will be referenced fre-

quently, it is worthwhile to expand them into a representation that is easily accessible by the target machine. The need for persistent state management for temporary active cascades depends on a system's mode switching policies. If modes may be changed only when all processes in a cascade are in a defined state, the state vector does not need to be remembered. Otherwise, processes must be re-initialized with the values from the preceding state vector.

Run-Time Mechanism. In addition to a fully static implementation that must deal with event and communication messages alone, a change in the mode of operation is signalled to the scheduler just like a conventional (action) message. The purpose of this message is to trigger changes in the table as required for this specific new mode. This chore is treated no differently from the execution of a cluster transition.

7.3.3 Example

In the robot example (its component architecture is shown in Fig. 37, p. 73), a number of specific modes of operation dictate the work load that must be dealt with by the various system components at different times (Tab. 2). For example, mode 1 calls for precise movement and high resolution perception only, mode 2 requires the robot to move along a trajectory with high speed, but low precision, performing only simple gripper operations, etc. In a naïve static approach the load variation in the components relating to different modes of operation results in a low average utilization of processing and memory resources.

Table 2: Robot modes of operation

Mode	Motion	Handling	Navigation	Perception
1	Precision	none	Basic	HighRes
2	Fast	Coarse	Best	LowRes
3	Standstill	Fine	none	HighRes

In cases like this, where disjoint modes of operation are well defined and known in advance, a well adapted implementation can improve resource utilization considerably within the given static environment. In the following, improvements through dynamic reconfiguration for functional and event extractor components are investigated.

The components *Head, Motion, Handling, Navigation* and *Perception,* together with their corresponding event extractors all exhibit highly variable load characteristics that depend on the current mode of operation. The functional component *Motion* is examined in more detail for variations in its cascade

dimensions and the influence of these variations on the rest of the system. *Motion* is placed in its own implementation unit as a single cluster; its interaction net is shown in Fig. 84.

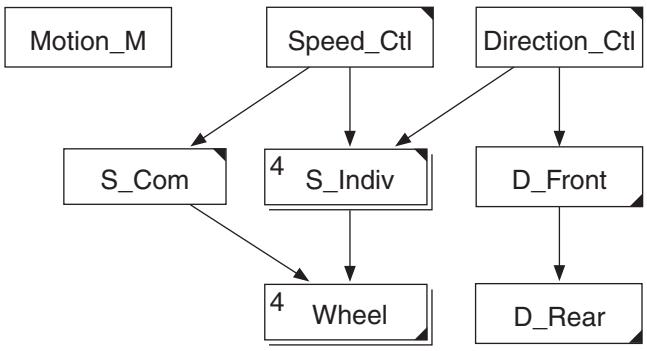

Figure 84: Robot: interaction net (Motion)

Any temporarily active cascade subset is contained in the cluster's interaction net. A cascade subset's extent directly influences the required processor and memory capacities for a given mode of operation. Considering the multiplicities of the processes that deal with the four wheels, the requirement variations are substantial (Fig. 85).

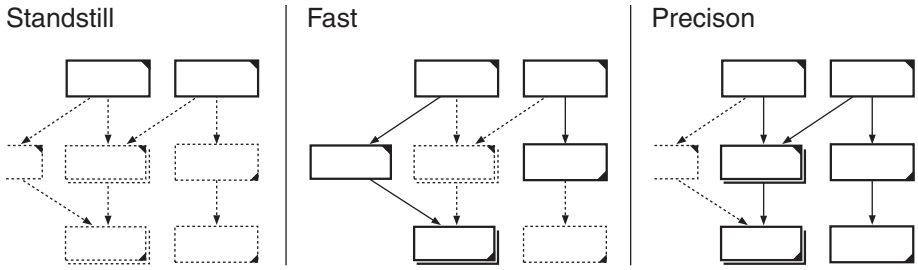

Figure 85: Robot: cascade subsets (Motion)

Cascade variations affect the event extractor tables insofar as processes with their own capability to receive messages may become inactive in certain modes of operation. In the cluster *Motion* this concerns the common and individual wheel speed control processes *S_Com* and *S_Indiv* (which are mutually exclusive). The table size varies from maximum in mode 1 *(Precision* for cluster *Motion)* to minimum in mode 3 *(Standstill).*

Requirement Variations. Resource requirements are summarized for the entire system and shown in Tab. 3 and Fig. 86 for memory (in Bytes) and processing time (proportions over one period), respectively. The first row in Tab. 3 lists the permanent requirement of each component, followed by specific additional needs per mode of operation. Implementations that make use of these overlaps

save 45% on memory requirements (using a maximum of 2250 Bytes at any one time instead of 4090 Bytes), the minimum requirements being dictated by mode 1.

Table 3: Modes of operation and memory requirements

Mode	Head	Motion	Handling	Navigation	Perception	per mode
1…3	750	200	50	50	150	1200 (permanent)
1	100	900	0	50	0	**1050**
2	70	300	200	200	200	970
3	20	0	750	0	100	870

While no reconfiguration is needed to profit from the variable demands in processor capacity (Fig. 86), it is apparent that not all components could be accommodated in one processing period if their maximum requirements from different modes of operation were to coincide. Because there is a possible trade-off of processing time granted to any component with the number of messages this component can deal with in a single period, the savings cannot be easily ascertained.

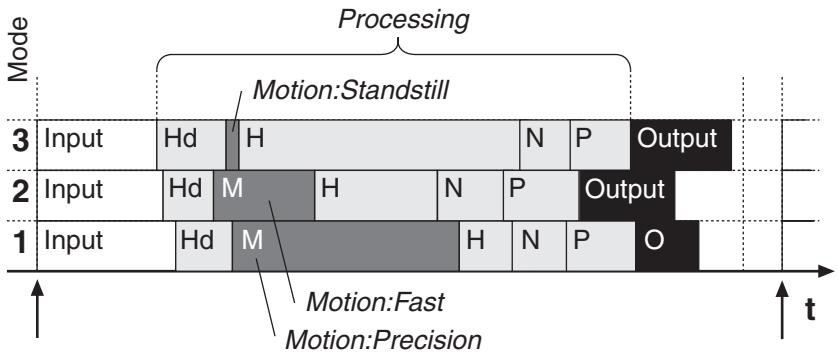

Figure 86: Modes of operation and processor utilization

Run-time performance. A sample situation that illustrates processor loads in the vicinity of a change in the robot controller from mode 3 to mode 1 is shown in Fig. 87. Functional processing time has been summarized (light gray) to point out the role of the component *Motion* (dark gray). It can be seen that the transition from *Motion* being practically idle to requiring a substantial amount of processing does not affect the overall performance. Only the period where the actual mode change takes place is dominated by the memory manager at the expense of other processing. At least, this routine is deterministic and the instances when it will be called are foreseeable and under control of the functional model and its modes of operation.

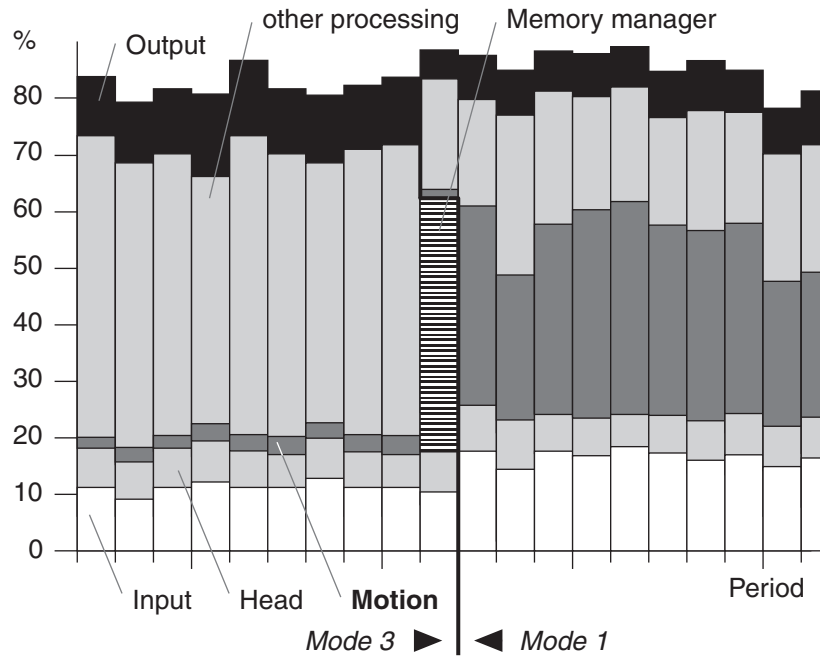

Figure 87: Processor utilization at mode change

As shown in Fig. 88, the same mode change increases *Motion's* dynamic memory partition from zero to 900 Bytes, as required in the new mode. During the period in which the transaction takes place only the permanent requirements are satisfied.

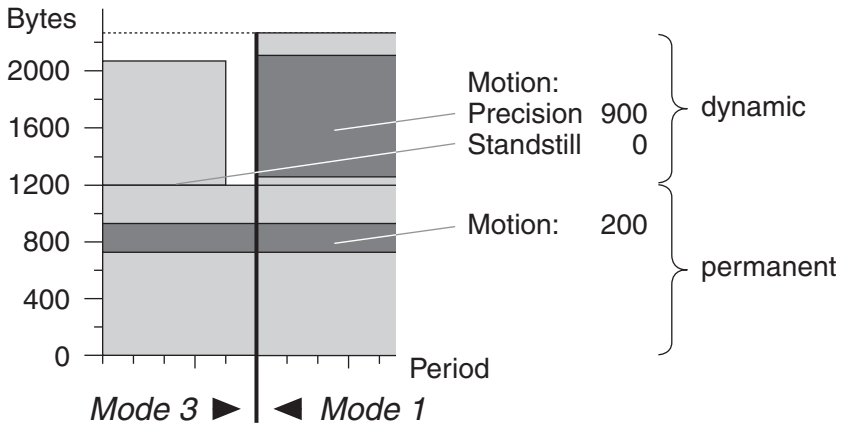

Figure 88: Memory allocation at mode change

8 Hybrid III

8.1 Hybrid Vehicles

The history of hybrid vehicle drive systems reaches almost as far back as the history of motorized vehicles itself. Aside from the combustion engine, some early car models were equipped with an additional electric motor to improve the rather poor acceleration [Pie09]. However, this early form of hybrid drive systems, along with the purely electric drive were soon superseded by stronger combustion engines that were not only reliable, but also economical thanks to mass production. Not until private vehicle traffic started to explode towards the end of the century and because of growing environmental concerns did these concepts experience a renaissance. Purely electric vehicles still cannot equal the ease of use and the virtually unlimited operation range of conventional cars, mainly because of the unrivalled energy density of fossil fuels, so that now hybrid cars have reappeared and even make it to full scale production of important manufacturers [Hub97].

In Switzerland, traffic is responsible for roughly one third of the total energy consumption. Aside from the railways that are operated electrically, this comprises 98% of all fuels consumed. Furthermore, with an assumed average energy efficiency of 22%, internal combustion engines prove to be the consumers with the lowest energy efficiency altogether [BEW94]. As a consequence, higher energy consumption implies a greater pollution burden especially in this field. For mobile applications, the chief advantages and disadvantages of electrical versus propulsion systems based on internal combustion engines relate to the energy conversion and storage, where they exhibit quite contrasting characteristics. In the case of the combustion engine, the high energy density of fossil fuels is the main benefit, its insufficient energy efficiency the major drawback (mainly caused by the inadequate use of the engine under part load). In comparison, the chief advantages of electric drives consist in the wide range of

speeds at a steadily high torque, combined with its high energy efficiency across the entire operation range. The batteries needed as storage devices, on the other hand, are poor performers, they are awkward to handle and present additional problems when they have to be disposed of. Hybrid concepts seek to combine the positive properties of the two approaches and thus help reduce the energy consumption and associated harmful effects of private vehicle traffic.

As the attribute suggests, hybrid vehicles have more than one way of storing and converting energy. The most well-known concepts involve combinations of electric motors with internal combustion engines or gas turbines (Fig. 89). Apart from conventional batteries, additional (short-term) storage devices used to absorb brake energy and to provide boost power under peak load may be installed, e.g., capacitors, hydraulic accumulators or flywheels. In principle two basic concepts can be discerned. The two drive systems act simultaneously in parallel hybrids, while they are concatenated in serial hybrids [Bur92].

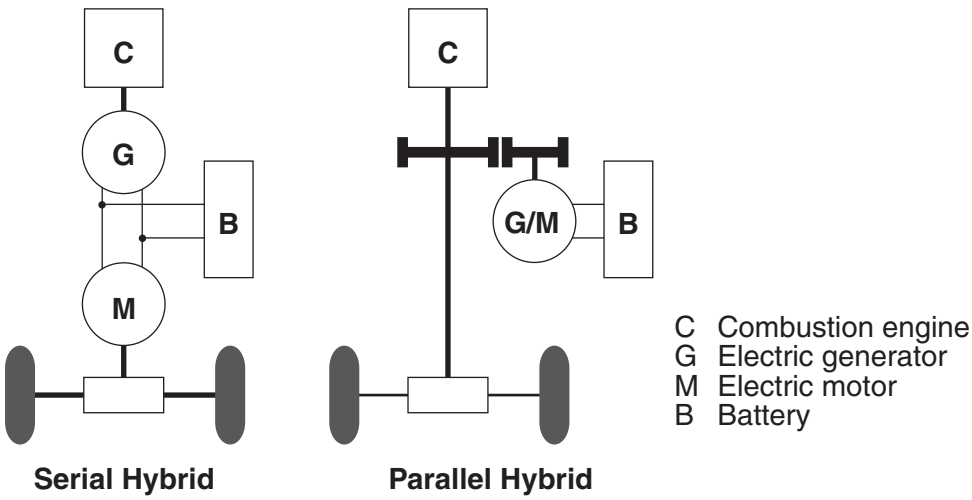

Figure 89: Hybrid concepts

8.1.1 Hybrid Concepts

In *serial hybrids* a combustion engine drives a generator to produce electric energy that is either fed directly to the drive motor(s) or stored in capacitors or batteries for later use. Because the vehicle is driven by the electric motor alone at all times, this motor must be dimensioned according to the desired top speed, cruise speed and acceleration. If an advantage is to result in comparison with conventional drive systems the energy efficiency of each component must be optimized and the control strategies for the energy flow chosen carefully. Serial hybrid designs are attractive because they can contain several electric motors that are sometimes placed directly into the wheel hubs.

Parallel hybrids have two essentially separate drive systems that can act upon the vehicle simultaneously. The electric drive in this case has only auxiliary functions and must mainly compensate weaknesses of the partially loaded combustion engine. In many of these concepts, the additional weight compensates amply for the lowered fuel consumption, making reduced exhaust emissions their only advantage. The most simple of parallel hybrids can be attained with two drives acting on different vehicle axes, in which case they operate independently as two entirely distinct propulsion systems. On the other hand, if they are coupled to a common shaft, the auxiliary motor can also take on the role of electric starter for the combustion engine. In such concepts, the possible disadvantage of the electric motor having to rotate even when not in use may be countered by additional clutches with which to separate the two drives from each other.

8.1.2 Drive Systems and Energy Storage Devices

Putting together different drive systems and energy storage devices means to strive for a compromise with the aim of combining the benefits each element has to offer. A short comparison of the key advantages and disadvantages summarizes the criteria that must be applied in the selection and evaluation of a specific concept; these issues are discussed in greater detail in [Vez97].

If the *energy density* of the storage devices is compared, fossil energy is at a great advantage. The calorific value of regular gasoline amounts to approximately 10 kWh/kg, which is in striking contrast with the 30 to 100 Wh/kg in batteries available today. The *cruising range* is directly dependent on it; for vehicles with internal combustion engines it varies from 500 to 800 km, electric vehicles reach between 40 to 150 km. On top of that is the difference between simple refueling and lengthy recharge times, an important point in daily use. Both these disadvantages could disappear with the further advent of fuel cells, but since these have not yet left the research centers, only conventional accumulator cells are examined.

When the *energy efficiency* is compared based on crude oil as the primary energy source, instead of just comparing the on-board converters (internal combustion engine, electric machine), neither of the concepts shows any special advantages at rated load. At part load, however, the situation changes drastically, because electric motors exhibit a very high energy efficiency across the entire range of operation, whereas combustion engines suffer significant performance degradations when not fully loaded. Since a car's energy demands only very rarely reach rated load in normal traffic, this presents the major imperfec-

tion of even state-of-the-art private vehicles. Conditions are similar with regard to *engine torque.* Combustion engines perform well only in a narrow operation range, namely above a certain rotational speed. Electric drives, on the other hand, can deliver maximum torque even at small rotational speeds.

The *weight* of a propulsion system and its mounting dimensions are lower for electric solutions, but this is only true if the accompanying energy storage device is left aside. *Emissions* generated by road vehicles are for the most part noise and exhaust gas from the combustion process. Electric vehicles produce little noise and no exhaust gas (locally). Therefore they actually could help in ameliorating the air quality of big agglomerations in the future.

Short-time *energy storage* devices permit the recovery of the vehicle's kinetic energy while braking. Additionally, they can equalize peak energy demands and thus help improve the efficiency of the combustion engine. Unfortunately, most of the conceivable solutions exhibit only unsatisfactory overall energy efficiency, limiting the possibilities to mechanical storage devices. In general, these are flywheels with a high energy density and direct mechanical coupling to the drive system. Its installation is very complicated; it necessitates a continuously variable transmission with an extended setting range and the associated elaborate control mechanisms [End96]. At the same time, a flywheel as short-term energy storage device allows for intermittent use (duty cycle operation) of the combustion engine, i.e., mechanical energy production and consumption actually are completely separated. The vehicle is driven by the periodically recharged flywheel, thus the combustion engine is operated in intervals only, but always in the range of its optimum efficiency.

A propulsion system for an economical clean air car can be built with a combination of these elements, and without giving up the main advantages of conventional vehicles: unlimited cruising range and autonomy.

8.2 The Project Hybrid III

Hybrid III is the latest member of a family of hybrid cars developed at ETH since 1990. As well as in the preceding projects, the intention was not to build a completely new kind of car, but rather to strive for optimum energy efficiency and for a substantial reduction of polluting emissions, using methods from the boundaries of today's technology. The main emphasis in this project is on the propulsion system, so that a commercially available vehicle was to be fitted with a new drive system. This novel drive system is a joint development of several laboratories at ETH, and industrial partners [DHE93]. It features a gas-

oline engine, an electric motor, a flywheel storage device, batteries, and a continuously variable transmission. Various clutches connect the gasoline engine, the flywheel and the transmission, while the electric motor sits directly on the drive shaft. This assembly allows for all possible operation modes of a parallel hybrid, including duty cycle operation. Its most remarkable achievements are twofold: an improvement of the combustion engine's efficiency in extreme underload operation, the prevalent situation in urban traffic, and best possible use of the available energy, because recovery is possible mechanically as well as electrically. In urban traffic the vehicle is driven by the flywheel via the newly devised continuously variable transmission. Because the normal setting range of 1:5 is traveled through twice, the much wider range of 1:20.5 becomes available. With this expansion it is possible to translate high flywheel rotor speeds to small running speeds of the vehicle and vice versa.

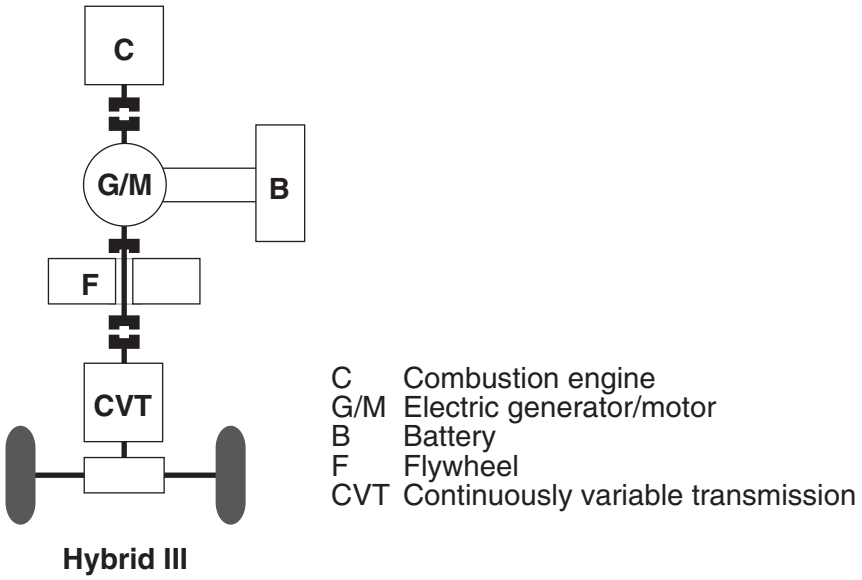

Figure 90: Hybrid III drive system

The main features of the concept are:

- duty cycle operation: whereas engines in conventional cars run continuously and within undesirably broad rotational speed and power ranges, the combustion engine of Hybrid III is used intermittently and only in the vicinity of its optimum operating point,
- when braking, Hybrid III recovers this normally lost energy onto its flywheel with high energy efficiency,
- state-of-the-art exhaust gas cleaning combined with careful thermal balance management reduce harmful effects on the environment,
- shorter distances, namely within urban areas, can be covered electrically, hence completely exhaust-free.

8.2.1 How it Works

Whenever the flywheel has slowed down and needs recharging, the gasoline engine is engaged. It works intermittently, i.e., it accelerates the flywheel with full power to the desired rotational speed, after which it is immediately disengaged and turned off. Because the engine itself has been stripped of all unneeded inertia, starting and stopping can occur within a few revolutions. Operating ranges with unfavorable energy efficiency are therefore eliminated. On highways the system operates without the flywheel, because the load is sufficient to assure near-optimum consumption anyway. Exhaust-free driving in urban areas is possible up to approximately 60 km/h with the electric motor alone. In this operating mode the flywheel helps cope with peak demands that would exceed rated power output of the electric drive system.

8.2.2 Results

Measurement results confirmed the high expectations: all test bed measurement cycles showed reductions in energy use of up to 40%, in spite of the additional weight of more than 300 kg. Energy consumption in the typical urban measurement cycle ECE-15 amounts to 4.7 l/100 km for Hybrid III compared to 8.9 l for the same car with a conventional drive system [Die96].

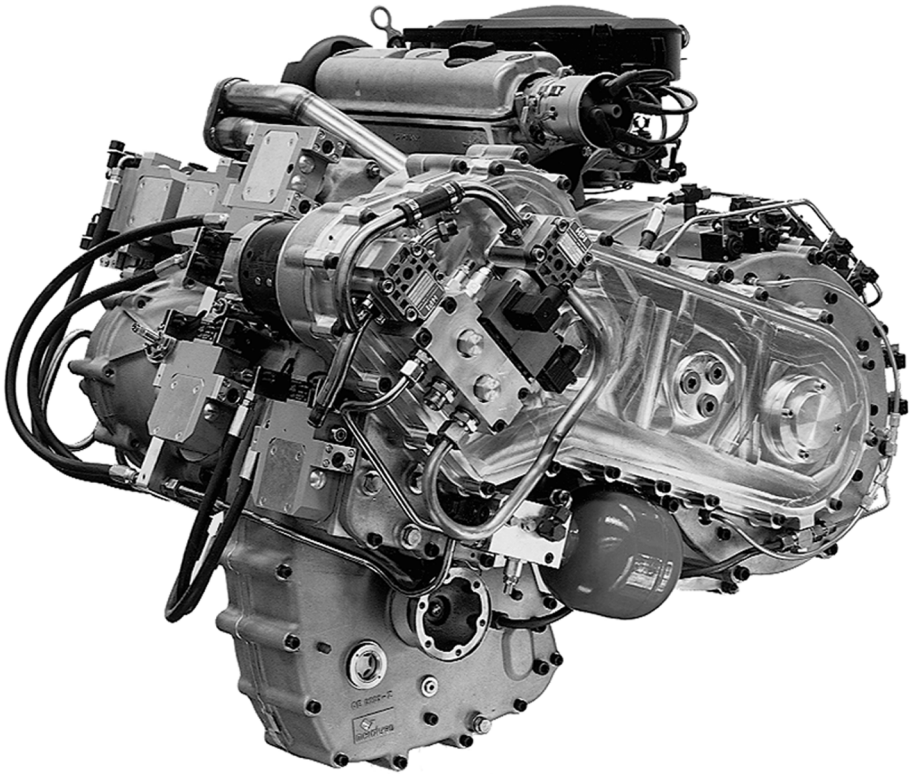

Figure 91: Hybrid III drive train

8.3 Hybrid III Components

The components of Hybrid III were designed and built by the individual academic and industrial project partners in different places. Subsystems became operational with partial control systems that were identical to the one finally installed in the vehicle itself. A detailed description of the mechanical design concept is contained in [End96].

Internal Combustion Engine. The internal combustion engine is a standard 1.3 l gasoline engine that has been adapted to the special needs of Hybrid III. Thanks to a modified control strategy the optimal specific fuel consumption matches full-load operation across the entire range, so that the intermittent charging of the flywheel always happens with optimum efficiency and minimal consumption [SGC96]. Other modifications affect the utmost reduction of the moment of inertia and the thermal comportment during duty cycle operation. The latter measures are necessary to protect the catalytic converter from undesirable temperature excursions.

Electric Motor. The electric drive consists of the electric machine [Kün95], power electronics and traction batteries, a battery charger, the vehicle power feed, and a braking resistance [Vez97]. There exist four main electrical operating modes with vastly different requirements (starter, on-board generator, propulsion, brake). Maximum permanent power is 6 kW for acceleration (limited by the batteries) and 16 kW for braking. The higher value is only available in conjunction with the braking resistance, which is itself integrated into the vehicle cooling circuit.

Transmission and Flywheel. The transmission and the flywheel are the key mechanical components. The centre-piece in the i^2-transmission is a continuously variable chain converter. Additional fixed gear ratios and a number of hydraulically actuated clutches permit to travel twice through the converter (equivalent to having two such converters in series) and thus to attain the necessary high setting range of 1:20.5. The device features an energy efficiency magnitudes above customary continuous vehicle transmissions, even when operating at feeble loads. The flywheel weighs 49 kg and its moment of inertia amounts to 1.039 kgm^2. It is designed for operation at a maximum rotor speed of 6000/min.

Thermal Balance and Exhaust Gas Cleaning. The combination of many different elements with dynamically changing cooling and heating needs demands a lot of the car's thermal management, with the intermittent operation of the combustion engine under heavy load presenting a special challenge. All driving elements, together with the power electronics and the components for exhaust

gas cleaning, are integrated into a heating/cooling circuit. Aside from the catalytic converter, a number of heat exchangers and latent heat registers are needed. Heat flow is controlled by means of electrically operated pumps and valves.

In contrast to conventional solutions, where temperatures and gas concentrations remain within restrained bounds, Hybrid III's exhaust gas cleaning must be capable of dealing with the varying concentration ratios caused by the intermittent operation of the combustion engine. Whereas the temperature in the custom-made catalytic converter is stabilized adequately with heat exchangers and a double-skin exhaust manifold, a catalytic material capable of tolerating the greatly varying oxygen concentrations had to be sought at first.

Vehicle Interface. The Hybrid III propulsion system is built into a slightly modified Volkswagen Sharan that has retained all its standard equipment. The vehicle interface unites an especially designed multi-functional display and a drive selector lever with force feedback. All the other sensors and actuators do not differ from the ones found in conventional cars, effect of an early design decision. Therefore, anyone accustomed to driving should have no problems with this hybrid model. The most notable difference is an addition to the drive selector lever. Because the conventional hydraulically operated vehicle brakes were left untouched, the lever doubles as input device to control braking via energy recovery. Position and force feedback is used to indicate the availability of the various energy converters and storage devices such as the flywheel, the electric converter and machine, the accumulators, and the braking resistance. Contradicting user commands can likewise be resolved by manipulating the lever position. The other notable enhancement is the graphic display. In the default mode it serves as the car's standard dashboard, but with its various other operation modes it can also double as a measurement value readout. An arbitrary number of graphic screens can be defined with arrangements of signal and inner state displays. Engineers driving the car can choose dynamically from among these pages to focus on individual areas of interest.

Control System. The five principal components that make up the propulsion system for the Hybrid III car obviously must be united for the final installation in the vehicle, where they act together under the coordination of the same regime. The same is true for simulation implementations, for which all components must be contained in a single functional block that integrates into the simulation environment. Nevertheless, these components form coherent entities all by themselves and as such they were to be developed and tested. Subsystems became operational with control systems that were identical to the one finally installed in the vehicle itself. Partial systems unite elements that have

their own, independent processor capacities in the control system. Therefore, a solution had to be sought that could integrate these different operation modes without imposing too many changes when converting from one to the other.

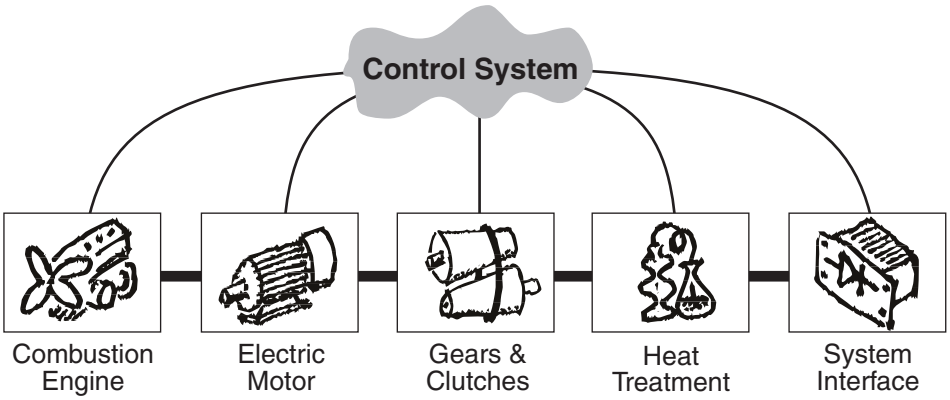

Figure 92: Hybrid III system structure

9 Hybrid III Control

It is advantageous in any development if the architecture of the control system reflects the component setup of the machine itself. The control system for the Hybrid III car is composed of five singular, loosely cooperating blocks, structured in just the same way as the propulsion system itself (see section 8.3).

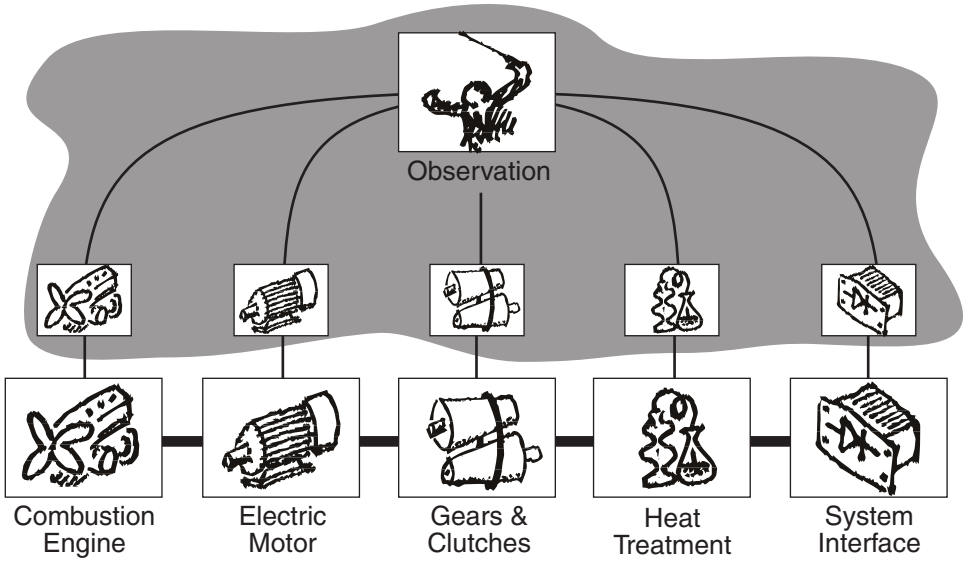

Figure 93: Operational structure of the control system

These so-called clusters break up the whole into partial systems, which are easier to handle. They form widely autonomous subsystems that are capable of executing complex operations all by themselves. Groups of functions that are united in the mechanical construction are assembled within the control system as well, thus enabling the separate development and testing of complete parts in accordance with the entire structure. In the same manner, clusters also permit a hierarchical system decomposition, restraining any direct communication of leaf elements with each other.

An additional cluster with higher level functionality monitors and coordinates the individual subsystems. Input and output devices are located within the scope of a specific cluster, namely the one which controls the mechanical component where said devices are installed. The interface to the vehicle extends into the mechanical construction by means of rugged, distributed satellite boxes miniaturized especially for vehicle use. Another added cluster deals with the actual peripheral devices and distributes the acquired data among the functional subsystems. In the same way it collects output data from the subsystems and emits them to the machine.

This representation abstracts completely from lower level functionality, but it is this structure that lies at the heart of what specialized engineer groups use in the development of their respective part of machinery and pertinent control system. By choosing a star topology another advantage has been gained: because each of the working clusters needs only a single and uniform connection, it is perfectly possible to set up a partial system without any interface changes to the cluster in question and for any specific test bed installation.

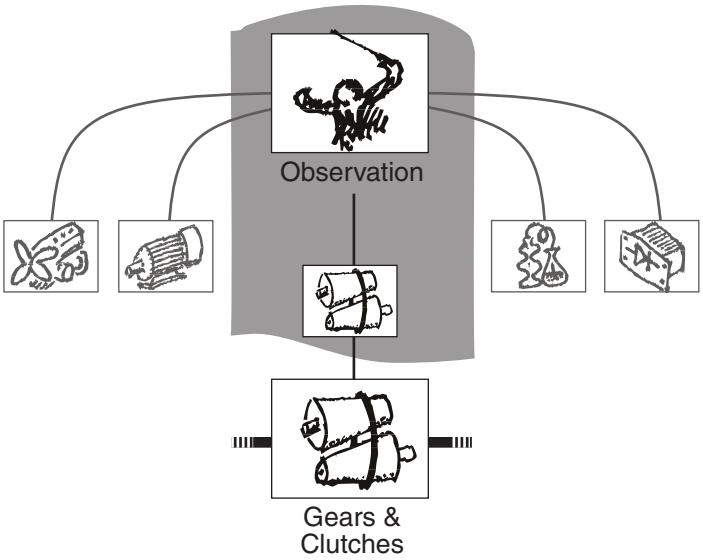

Figure 94: Partial system

Hybrid III comprises the following clusters:

O	Observation
C	Combustion
E	Electrical
M	Mechanical
H	Heat Treatment
S	System Interface
IO	Input & Output.

9.1 Concepts

In a functional implementation the cluster partition and interconnections outlined above must be adapted to the features existent in a hardware environment, namely the processors and the peripheral interface. Notably the star structure cannot be accomplished easily for an arbitrary number of working clusters. In the chosen implementation it is replaced by a double ring, ensuring that all partners can be reached in two ways. While the operational structure guarantees interface consistency of any partial with the full implementation, assigning separate processor nodes to all the clusters guarantees that performance will equally remain constant in both types of implementation. Furthermore, a single cluster's performance can be improved with additional local resources (processing capacity, memory, dedicated interfaces etc.)

It would be costly and inefficient to fit every cluster with its own data input and output mechanism. The supplementary cluster IO, hidden from the operational structure, handles peripheral data input and output for the entire system. This cluster is also included in the ring structure and in addition conveys input data to all the other clusters simultaneously through an efficient hardware broadcast mechanism BC.

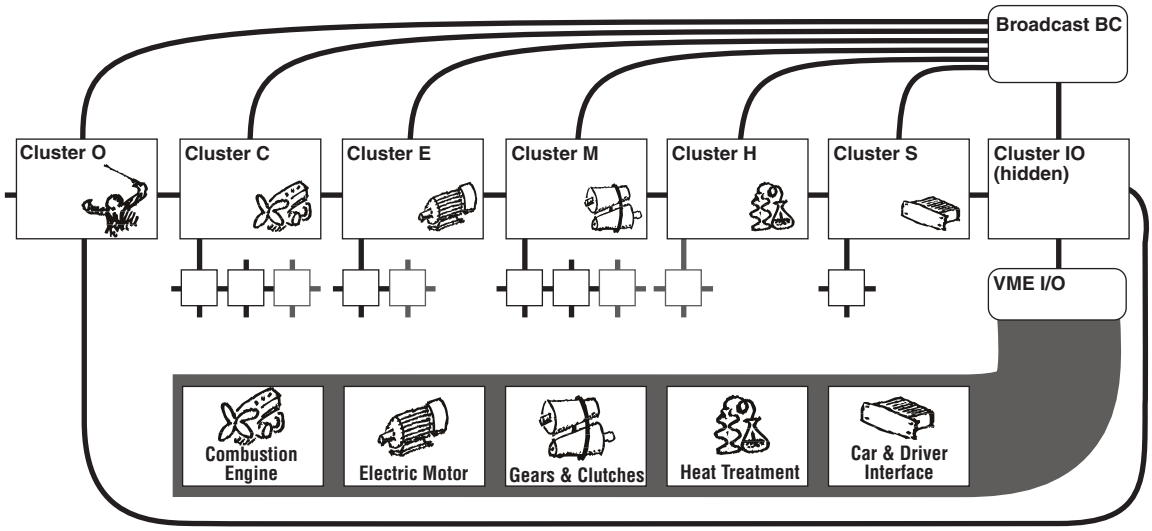

Figure 95: Implementation structure of the control system

The respectable number of sensors and actuators (c. 150 sensors and more than 80 actuators, numbers increasing for test bed installations) and the restricted space in a vehicle impose the use of efficient solutions for data input and output. On the hardware side, the connection with the components in the environment is based on the industry-standard VME bus, complemented with a mezzanine-bus used together with distributed connector boxes (satellites) that

were especially developed for this application. Software for various hardware arrangements is generated from high-level configurations.

Global functions (system operating modes, error handling and the like) are controlled by the *system master* placed above all the others and situated within the cluster Observation O; typically all the slow activities with reference to the whole installation are handled here. At the same time the cluster supervises the overall system communication and is in charge of the interface to the developer's workstation as long as it is required. An additional processor node (Host) operates buffers needed for decoupling the workstation interface itself. This node can also be configured for its second use, namely to monitor and display ring traffic selectively.

All the working clusters[1], i.e., the ones associated with a piece of machinery (C, E, M, H, and S), contain a *submaster* to execute all the clusters' local functions. Therefore fast operations remain within the scope of a single cluster, where they can make use of locally added resources if necessary. The detailed realization of the vehicle's control strategies (associated with the operation of clutches and transmission ratios) focuses on the cluster Mechanical M and is therefore located within this cluster's *strategy master.* It controls those fast operations that, owing to circumstances, may concern the entire system.

Sometimes the functional membership contradicts the constructive realities. Because the control system is modeled on the machine construction, all elements are connected with those clusters to which they belong constructively. The imposed structural restrictions require a *remote service* mechanism. This service makes externally situated elements accessible to those clusters where they belong functionally. Along with the above, clusters typically contain an arbitrary number of processes for the actual control functions, by far the bulk of a cluster's chores. All these processes are defined and specified by the specialists' teams within a modeling environment.

9.1.1 Processors

The computers used in the control system are transputers [Inm89], 32-bit CMOS devices that incorporate processor, memory and communication link interfaces for direct connection to other transputers on a single chip. Multiprocessing systems can be constructed from a collection of transputers operating concurrently and communicating through links. System design is simpli-

1. Hybrid III clusters are actually processor nodes that may contain a functional reactive block comprising several CIP clusters in one implementation unit.

fied by the use of tasks[1] as standard software and hardware building blocks. The possibility of dividing a complex chore into individual tasks is ideally tailored to our system demands; each block is controlled by a group of tasks on an independent computer, which communicates via links with his partners. As far as the hardware is concerned, a link is a DMA machine that operates alongside the processor and coprocessor components.

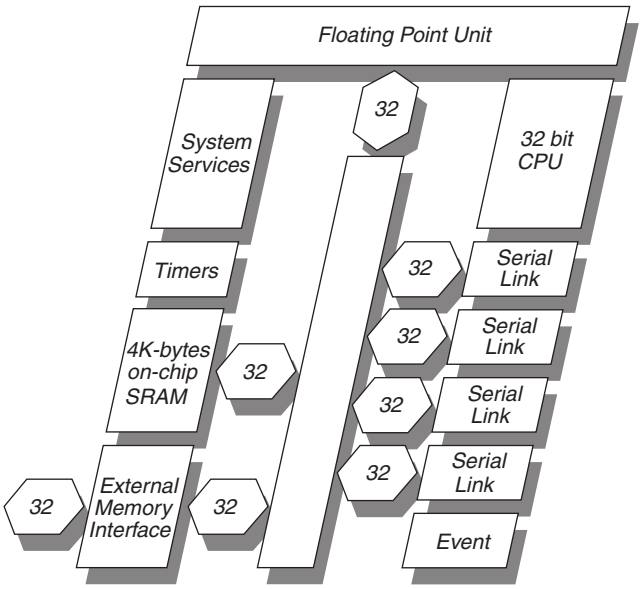

Figure 96: IMS T805 transputer block diagram

The hardware architecture stands on the firm grounds of Hoare's CSP [Hoa85] and directly supports communication and task scheduling. Therefore, applications can be placed on single or multiple transputer installations without the need and the overhead of an operating system. Procedure call, task switching, and typical interrupt latency times are in the sub-microsecond range. Type T805 transputers, which feature an internal 32-bit wide bus, reach c. 15 MIPS (million instructions per second) at a clock frequency of 30 MHz. The built-in 64-bit floating-point unit is capable of performing 2 MFLOPS (million floating-point operations per second) concurrently with the processor. The maximum communication speed attainable on the links that provide connections to other transputers is 20 Mbit/s.

Aside from the special cases concerned with peripheral hardware interfaces, the transputers charged with the functional work load are located on TRAMs (TRAnsputer Modules), small sub-assemblies of transputers and memory with a simple signal interface profiled in modular sizes. They are mounted on carrier boards that are electrically and mechanically compatible with the VME bus,

1. Schedulable transputer processes are termed tasks to distinguish them from CIP processes.

making some of the links available on the backplane. The modularity and minimum real estate required provide the flexibility of adding more modules or replacing existing modules with faster ones according to emerging needs.

9.1.2 Communication

The operational star topology forms the base of all communication in the complete system. Clusters therefore find all the required interfaces even in partial implementations, notably in the various test bed installations. Messages from one cluster to another are asynchronous and can only be sent along predefined data stream networks. Because there is no direct communication among the working clusters, only two such networks are needed. The *collector net* is used for all data sent from any of the working clusters and destined for the cluster Observation O, the *distributor net* transmits data in the other direction; the latter covers both point-to-point and broadcast messages to all clusters simultaneously.

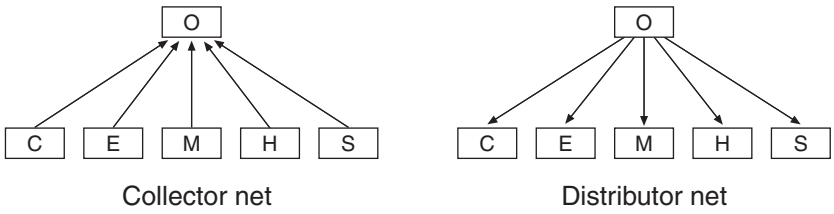

Figure 97: Data stream networks

As outlined, the hardware topology is completely different and must take into account the transputer link feature and especially the limited number of available links. Links are serial connections via two one-directional signal wires, using a simple protocol. The protocol permits an acknowledgement to be transmitted as soon as the receiver has identified a data packet. The acknowledgement can thus be received by the sender before all of the data packet has been transmitted, which more than doubles the attainable peak data rate.

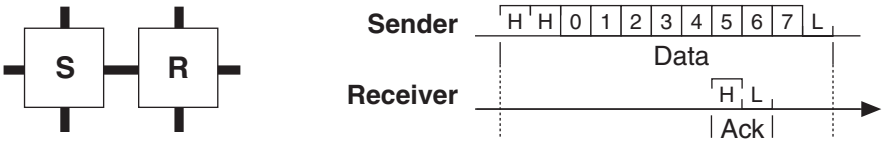

Figure 98: Point-to-point connection between transputers

Because single transputers have only four links, alternative solutions have to be sought for the star connection as well as for the broadcast mechanism. On the

hardware topology the hierarchical cluster structure is implemented as a bidirectional ring. A special *communicator* task on each cluster handles ring traffic from and to this cluster and forwards all the other incoming messages (see section 9.4 for a description of clusters and their tasks). The ring mechanism is also used for output data and all in all for any kind of traffic in the system.

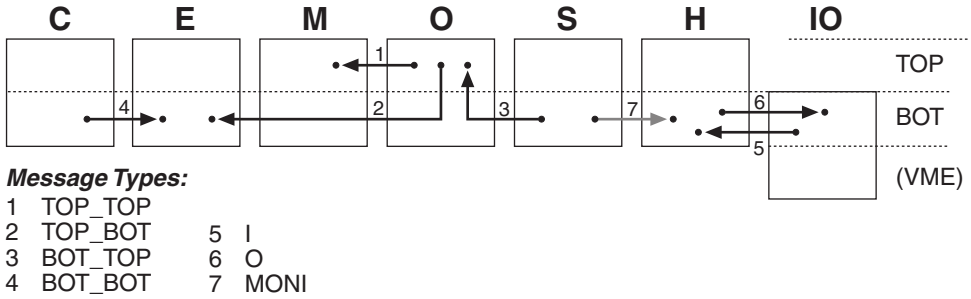

Figure 99: Message levels

Protocols for point-to-point and broadcast connections are agreed upon globally. Aside from designating the message origin and destination clusters, these protocols also incorporate fields that denote the message type as a pair of levels. The levels are either `TOP` or `BOT`, corresponding to the levels of abstraction used in the models and configurations. Messages can be received on the same level as they were sent, as is the case when they are transmitted from one functional component to another, but they can also change levels, usually to switch between different behaviors at the receiving end, or for error indication. Additional message types are used for output and monitoring purposes, as well as for a reduced input functionality in case of failures in the input broadcast mechanism.

The ring as presented in Fig. 95 provides a bidirectional connection of all clusters with one another, even though for the price of a few hops. It could principally cover all communication needs, including the distribution of input data. However, input data distribution via the ring has two disadvantages. The first one is a loss of synchronicity of the working clusters, as they would receive broadcast data in a cascade-like manner. The other, more important drawbacks are inadequate transmission speed, caused by the necessary retransmissions on every transputer along the way, and the possible overloading of the ring. Thus, mainly efficiency reasons call for a custom-built hardware solution for the broadcast mechanism. The sender transputer's output is split to up to eight receiving transputers. Their acknowledges are collected and then a single generated acknowledge is transmitted back to the sender. This way, synchronous transmissions with a high bandwidth are possible, and the arrangement remains completely transparent for both the sender and the receivers.

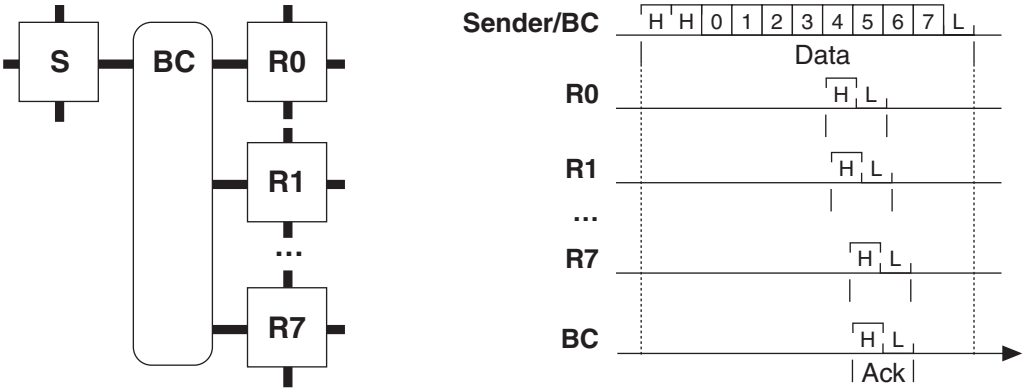

Figure 100: Link broadcast mechanism

9.1.3 Input and Output

Input and output signals are not globally available and belong to a specific cluster just as the respective sensors and actuators. For this reason there exists another structural conflict, this time between the operational structure and the realities of the peripheral hardware. Generally these are multichannel devices and it is inevitable to distribute the available channels among various clusters for reasons of space constraints and economic use. The problem of making a peripheral device accessible to different users is worsened by the various timing requirements imposed on the input signals. It is desirable to reduce these to just a few frequency classes (e.g., 1 kHz, 100 Hz, 10 Hz).

To balance the work load on the cluster IO and on all the other clusters for that matter, a static schedule is used to read carefully chosen groups of input signals at a given point of time. Together with the signals of the highest frequency class that are read in every cycle, a selection of signals of the other classes is processed. The large number of signals and complicated restraints on the hardware accesses make automatic code generation from configurations a necessity. Input and output are standardized. No interrupts are used, so that all inputs are periodic with fixed intervals (i.e., synchronous), whereas outputs occur immediately after a cluster has produced a new output value (asynchronous). Still, a burst of output values is associated with the preceding group of input values; particularly they are located together between two ticks of the input system (Fig. 101).

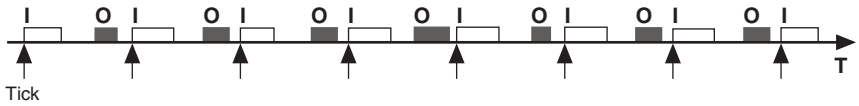

Figure 101: Input and output cycles

An individual cycle begins with the cluster IO acquiring a group of input data (Fig. 102). The raw data are filtered and converted to physical units, then forwarded to the working clusters. All clusters receive an identical copy of the newly acquired and converted data simultaneously through the broadcast mechanism. Sometime later, after the working clusters have finished their processing and calculated new values, output data from each cluster arrive at the cluster IO.

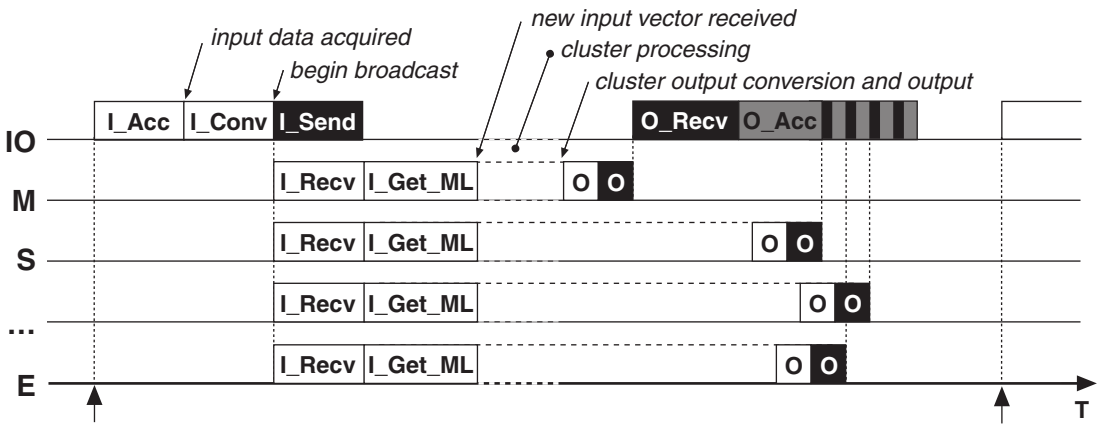

Figure 102: Synchronous input and asynchronous output

If these data do not already conform to the correct hardware formats and can therefore be output immediately, they are expanded and output in portions via dedicated buffers. The algorithms as well as related filter and conversion routines are complemented by sending and receiving mechanisms in the clusters involved. They are standardized and use particular data structures in each specific implementation that are generated to measure from configurations.

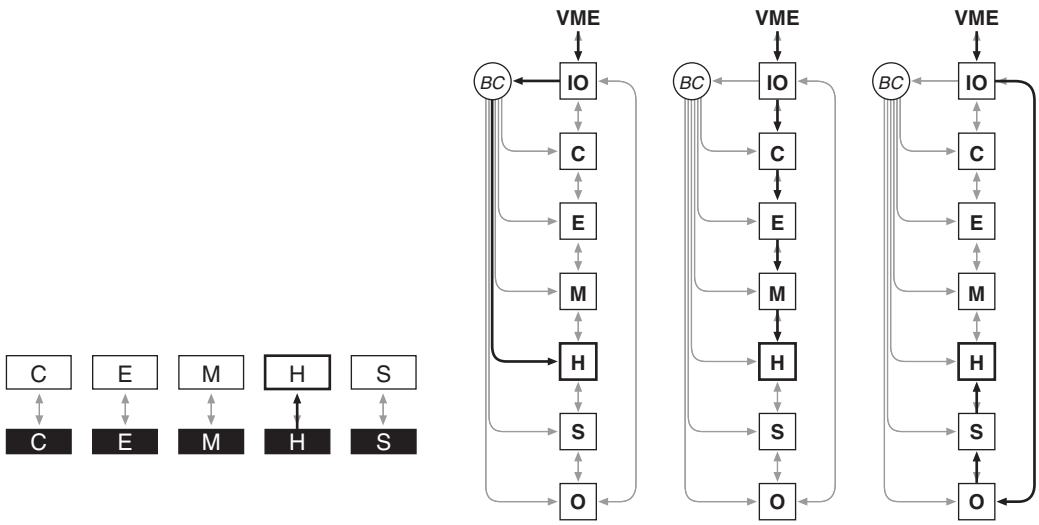

Figure 103: Data input path variants

Input and output data routing depends on global settings that reflect the overall system status. The default is to use the broadcast mechanism for data input and a preset direction for all ring traffic. In the presence of connection or single node failures alternative paths provide a limited redundancy. As an example different paths for data input of cluster H is shown in Fig. 103. The first variant illustrates normal operation using the broadcast mechanism, the other two present alternatives in case of a broadcast failure. Analogously, the ring is used in either direction for the transmission of output data.

9.1.4 Bus Systems

The use of the M-module mezzanine bus [VIT91] on top of the VME bus [VIT87]) for all input and output peripherals opens up possibilities for the use of a great variety of commercially available parts. The small card footprint and simplified bus interface facilitate manufacturing of additional more specific components. On the other hand, the VME bus gives cause for concern, because it is the only element that is not easily expandable in the system as a whole. If need be there remain two ways out: on the one hand, the local and dedicated application of processing power (the use of signal processors for the power electronics was projected from the beginning), on the other hand, direct peripheral connections of a cluster for critical signals (which were introduced in the course of the development for some continuous controller signals linked to fast mechanical components).

9.1.5 Mechanical and Electrical Installation

Evidently the control system in a vehicle should occupy a minimal amount of space only. Energy consumption must be low and the system must comply with demanding electrical and mechanical operating conditions. A small ruggedized 19" card cage with integrated regulated ventilation contains all components aside from the decentralized amplifier and signal conditioning units, termed satellites. These distributed peripheral installations are housed in diecast aluminum enclosures.

Component groups comply with the drive system architecture and allow for easy set up of test bed as well as of the final configurations. For the same reason, all connectors are equipped with plugs. The enforcement of a rigid concept for power distribution and for the connection of peripheral devices extends the good electromagnetic shielding inherent in the closed metal compartments to the entire plant. A high power factor dictated the choice of converters, power switches and other energy dissipating devices. For example, the various circuit

breaker solenoids present are of a special vacuum-type that combines fast closing times with minimal holding current.

Power feed voltages are generated centrally from the vehicle's standard 12 VDC wiring, using highly efficient DC-DC converters that are capable of dealing with the considerable input voltage excursions of the on-board power supply. The power supplies for all peripherals can be switched under control and the supply voltages on each satellite are continuously monitored during operation. The entire installation falls into three areas that are DC-isolated with a single central ground connection. For compatibility reasons the control unit is operated from 12 VDC even when stationary in laboratory tests.

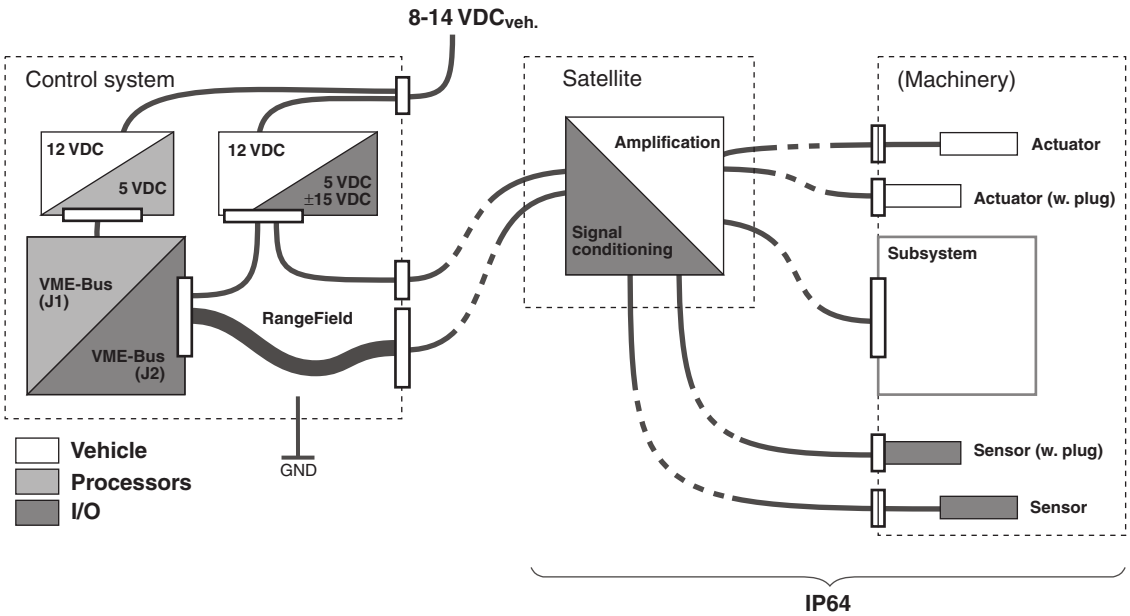

Figure 104: Cabling and galvanic isolation

The control system is turned on from an external switch (ignition key). It cannot be switched off the same way, because loosing power while in motion would equal a loss of (propulsion) control for this 'drive-by-wire' vehicle. Therefore, the control system will only shut down when the car is stationary and all systems have entered their required stable states. Turning off the ignition key does, however, initiate the necessary sequences that precede such a shut-down.

9.1.6 Code

As is the case in the development for any embedded system, work on the drive system for Hybrid III entailed the repeated assembly of parts that themselves are still incomplete and change continuously. In any complex project this situa-

tion easily ends in a nightmare without the continual assistance of efficient tools. Unfortunately, today's software community still does not have any integrated solutions at its disposal, thus creating the necessity to draw from different sources and to combine the obtained results.

The sequencers for the input and output systems as well as for the functional code, the ring mechanism, buffer code, etc. are playing the role of system software, as it were. These program parts serve as a foundation for the generated code, or they make use of generated tables. While being well structured and comparably short, they exhibit the typical high demands on efficiency and machine-intimacy. For this reason they were coded manually. Algorithms for continuous control present neatly confined subproblems and were developed within an especially suited environment (Matlab/Simulink [MAT99]) and then integrated as ready-made components. The actual event-triggered control functions as well as the overall functional structure are taken out of the hands of programmers entirely. The use of high-level configuration and modeling methods permits developers to work in a problem-oriented way and have tools do the actual coding.

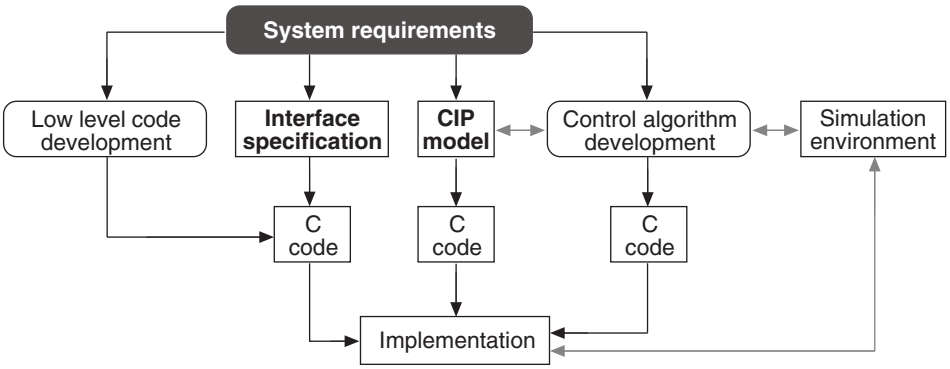

Figure 105: Hybrid III software development strands

The finished application is composed of several strands that can be dealt with independently of each other (Fig. 105). Because in a later development phase changes occur only in the central strands, where coding errors are eliminated entirely, the total number of errors is reduced, making remaining logical mistakes stand out more clearly for the developers to concentrate on.

9.2 Methods and Tools

The development strands shown in Fig. 105 suggest the use of separate tools to achieve solutions in the different problem areas that lead to an implementa-

tion. Because often work on a control system has to begin when the rest of the machine is in the design stage itself, and also because it would be too risky to directly try untested code on the machine, simulation plays a key role. Therefore, implementations not only have to be generated for the target system (or for parts of the target system), but also for inclusion with a model in a simulation environment (Matlab/Simulink). It is important that these different implementations be based on the same configurations. Only then it becomes possible to switch from one to the other at any time during the development process, and only then the unity of the approach remains intact by avoiding a split up of the project into different variants.

9.2.1 Working Environment

The project groups charged with developing singular components of the Hybrid III drive system worked completely independently and at different locations. Aside from the workstation, they disposed of a downgraded version of the control system with the needed processing resources as well as input and output facilities (Fig. 106). Coordination among the various groups was achieved by relying heavily on central repositories. To keep track of the different test bed configurations, the use of a data base was envisaged that included the machinery in the environment, the necessary parts needed for connecting these elements, and the software configurations. As work progressed, this intention was somewhat redimensioned to contain only elements relevant to the control system.

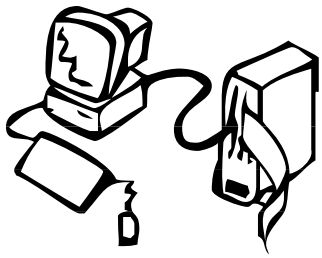

Figure 106: Development setup

Interfaces of both the mechanical parts and the control system were ready for later assembly from the beginning. Therefore, tests with various combinations of elements could be set up with only minor changes to single components. No modifications to the underlying program structure are necessary to operate a reduced version of the control system with any number of processing nodes as well as input and output facilities (cf. Figs. 107 and 109).

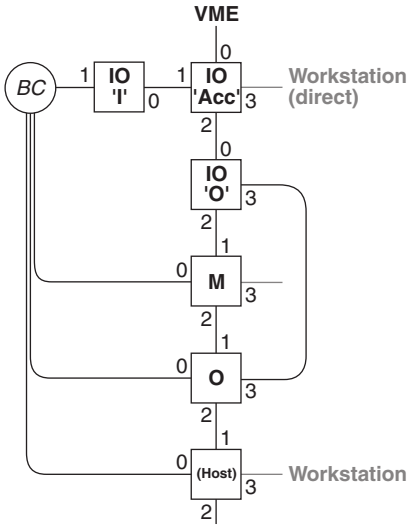

Figure 107: Partial implementation

All required adaptations are made on the architectural level, where dummy communication partners must be substituted for nonexistent clusters. The use of a sole workstation to operate and monitor any setup is a need for the installation in the vehicle itself, where limitations of space and available energy become important. On the day when development and testing is brought to an end, exactly the same control system structure continues to be used, only without a workstation attached. Apart from modifications to the drive selector lever required to easily operate the hybrid vehicle's features for energy recovery, the user interface then corresponds to the one in any conventional car.

9.2.2 Configuration and Modeling

A hybrid drive concept entails a considerable increase in complexity compared to conventional vehicles. This is especially apparent in the complicated interplay of the great number of components, which elude the imagination of any developer. An adapted representation of discrete states and continuous processes therefore becomes important already during the planning phases. Ideally, all preliminary work should be integrated smoothly into the later application, an impossible claim without the aid of suitable tools.

The starting-point for the choice of an appropriate working environment was the status quo. How do engineers arrive at solutions for their process control issues? As a rule, they are occupied as much with the coding of algorithms as with solving their actual problems. During the development process, and especially in the case of the inevitable later modifications, the questions of *what?* and *how?* become completely indistinguishable. Aside from burdening these

engineers with additional work they are not qualified for, and which is regarded by many as a necessary evil, the situation becomes especially cumbersome in the presence of errors and malfunctions. Detecting if their cause lies within the logical structure, or whether it is just a coding error, becomes a problem in itself. Consequently, the main focus was on taking away the chores of programming from the developers, and at the same time to separate logical from mere coding issues. Furthermore, the need for a strict separation of functional from connective questions was evident from the beginning.

At the outset of the project it became clear that no commercially available solution could satisfy these demands. It was therefore decided to enhance a then newly conceived solution for modeling system functionality with a complementing suite of tools that would facilitate the difficult task of embedding this solution in a dynamically changing hardware environment. Code generation from models and configurations are central to both these approaches.

Connection. The connection of an environment with those parts that realize a system's functionality, as well as all the other aspects of embedding such a functional solution within the context of a specific hardware architecture, clearly raises different questions than the functionality itself. It requires the knowledge of a computer scientist who can deal with hardware-related problems and with the low-level routines that handle task schedules, communication etc. In this field the mingling of concepts with the actual coding seems inevitable and is also less critical, because it touches upon the actual qualification of the engineers involved. It was therefore envisaged to create a stable collection of low-level tasks whose behavior would be adapted to the emerging needs mostly with modifications in the data supplied and not by changing the code itself. For efficiency reasons this approach had to be abandoned in the course of the project and replaced by a suite of tools that generate custom-made implementations according to configurations. The tools `acc`, `evt` and `dsp` deal with input and output accesses, event extraction, and display related features, respectively (see appendix A).

Functionality. The functional behavior is modeled as a collection of finite state machines in the modeling environment CIP Tool. It is based on a formal model [FMN93] and supports the development of distributed embedded systems. Its compositional paradigm is oriented towards the real objects in the environment. Constructive steps lead to complete systems that are defined by composition. The tool's capability to generate deterministic applications that run without further modifications in the connective environment was a key factor for choosing CIP Tool from the considered options. *Processes,* the CIP finite state machines, either model parts of the machinery in the environment,

or they sit on top of these models and accomplish the desired functionality through interaction. Synchronous groups of processes, termed *clusters,* are the elements that will later be distributed onto processing entities to achieve a feasible implementation. The operational structure of many control systems ideally matches this type of modeling through composition.

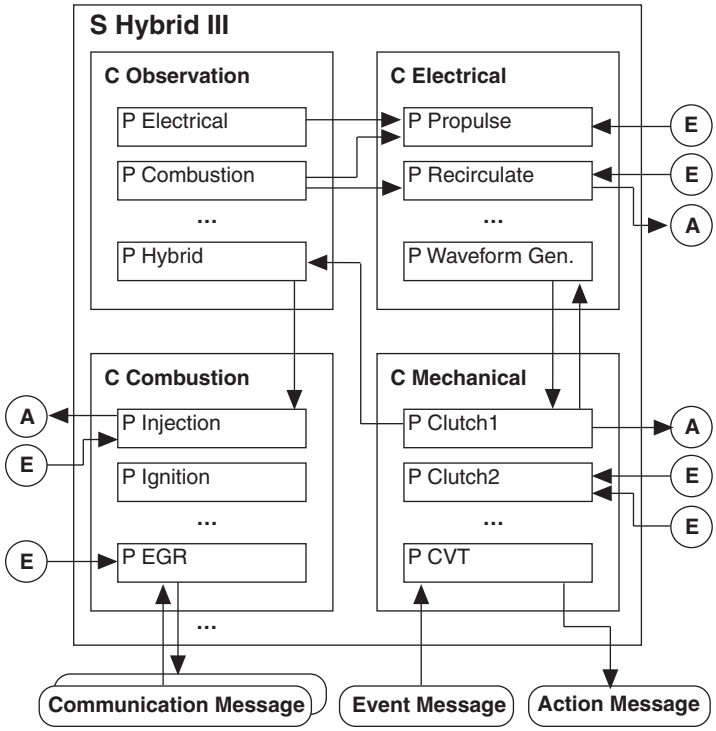

Figure 108: Hybrid III project structure (excerpt)

9.3 Hardware Implementation

9.3.1 Control System

The control system contains the power supplies for both the processors and the peripherals, a 14-slot double height cage, of which the first twelve slots are accessible via the J1/P1 connector of the VME bus [VIT87], and the additional two are reserved for dedicated processors with their own peripheral connections. In place of the J2 connector, the first five slots feature an additional backplane where the transputer link connections are routed. Except for the one transputer with direct access to the VME bus that sits in the first slot (node IO_ACC of the cluster IO in Figs. 107 and 109) most of the other transputers reside on TRAM modules; nine of the available 24 TRAM slots on four possible carrier boards are occupied. Two transputers are used in the role of dedi-

cated processors with M-module input and output interfaces independent from the VME bus. One of these handles display functions as a slave processor in cluster S, the other is in control of fast mechanical subsystems belonging to cluster M. The remaining seven slots can accommodate carrier boards for up to 28 M-modules, 26 of which have been used in the full configuration to basically provide the following input and output possibilities:

80	binary inputs (used: 71),
68	analog inputs (used: 53),
10	rotational speed measurements (used: 8),
5	angle of rotation measurements (used: 2),
64	binary outputs (used: 56),
24	analog outputs (used: 12).

All peripheral connections are accessible on a range field behind the compartment backplanes, from where they are routed to a set of flush mount and coded connectors, each satellite having its own data and power cables. The entire control module is pluggable and is easily removed for maintenance or replacement.

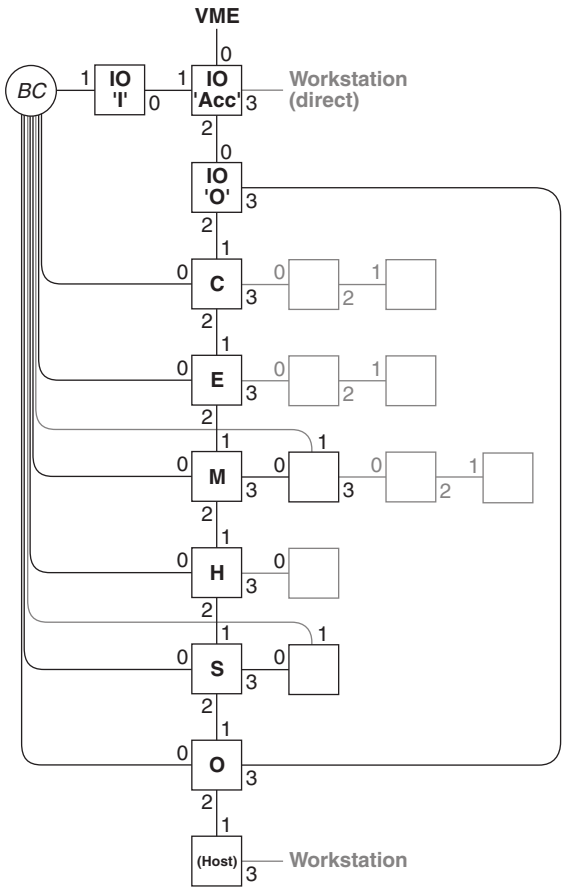

Figure 109: Complete implementation

The cluster dealing with inputs and outputs is critically tied to the bottleneck VME bus and was therefore enhanced with two more transputers. Three nodes take part in the chores of the cluster IO: one deals exclusively with bus accesses, another converts and filters input values and forwards them to the other clusters, the third maintains output data queues and operates the ring interface.

9.3.2 Peripherals

Peripheral connections are added up in satellites, ten of which are placed in the vicinity of machine components and associated with a specific cluster. Here, sensor and actuator connections are ranged to interface lines of the control system using converters, amplifiers and DC-isolation for signal conditioning. Thus, satellites contain a variety of functions that can nevertheless be categorized in specific groups, an ideal starting point for a modular system with freely combinable elements. Unfortunately none of the existing comparable schemes complied with our needs, especially not with the tight space constraints.

Satellites are small (250 x 75 x 75 mm), have a massive case with excellent electromagnetic shielding and protection level IP64, so that they can be placed anywhere in the vehicle. Any one of them contains the same base card with connectors for the various power supplies, for the signal connections from and to the control system, and for all kinds of sensors and actuators. Eight slots can hold up to three stacked interface S-cards each, of which 15 different versions have been developed. Power electronics are optimized for low losses to save energy and to permit operation in the closed metal casings. The peripheral elements themselves represent a collection of all possible ways of connecting such devices, automotive and from industrial control; S-cards are used to integrate them and condition their signals for the control system interface.

9.3.3 Additional Hardware

Wherever possible, the use of commercially available components was preferred over new development. New developments were necessary for the satellites, for all of the S-cards, for the broadcast element, as well as for some of the M-modules. Two of these modules that implement rather particular functions are presented in the following sections.

Rotor Speed Measurement M-RPM. Rotational speeds and angles of rotation are among the most important measurement data in the control of a drive system, and there are plenty also in Hybrid III. Digital ways of performing these measurements involve counting pulses or determining the periods of the pulse

train generated by a sensor. Unfortunately, the precision attainable at a specific measurement rate directly depends on the rotational speed itself (or vice-versa). At slow speeds, a high resolution at a steady measurement rate is only possible at the expense of a high number of pulses per rotation. The shortest response times in digital speed control lie at a theoretical 1.5 pulses, with practical values tending to be 5-200 times higher. The drive system's gear and clutch operations that must be controllable precisely at all speeds would make rise the necessary resolver resolutions as well as the signal frequencies involved. Available high-resolution resolvers have several disadvantages. They need an open shaft end for mounting, their elevated pulse rates might present transmission problems, and they are not rugged enough to survive in the harsh conditions (dirt and high temperatures) prevalent at the intended mounting positions.

A different solution had to be sought, with the potential to use hard-wearing sensors requiring only a low number of pulses per revolution, and evenly high precision across the entire measuring range. The approach makes use of the fact that mechanical systems exhibit no abrupt speed changes due do inertial forces. The recent measurement history therefore can be used to ameliorate the current result [Tru96]. The method fills the period between two pulses detected by the sensor with a generated stream of N pulses (Fig. 110). Based on the preceding measurement Z_k, a prediction of the point in time for the next pulse is made. The actual time of arrival then gives rise to a correction that will be used for the following prediction at $Z_{k+1} := Z_k + (N-P_k)$. The generated pulse stream corresponds to additional gear teeth on the coding wheel, synonymous with a virtual wheel with N-times as many teeth as there actually are. Requirements concerning coding wheels, sensors, transmission and data evaluation decreased considerably using this dedicated M-module.

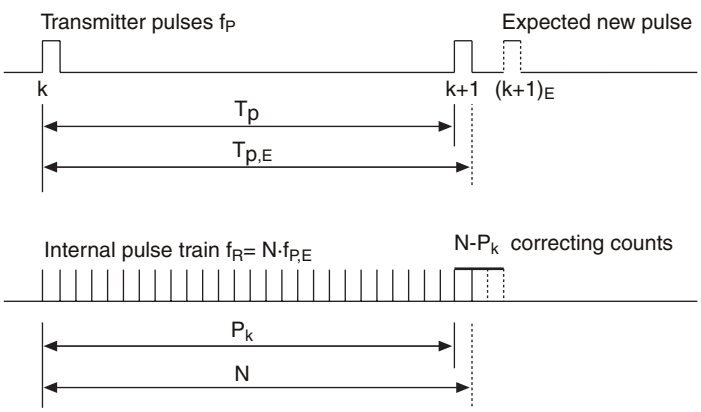

Figure 110: Principle of rotor speed measurement

Internal Combustion Engine Control M-ICX. This module can operate four cylinders of a four-stroke internal combustion engine and independently con-

trols fuel injection and ignition point timing directly. System output expense is reduced to the periodic updating of new angle and timing values. The two necessary inputs needed by the module at a high data rate are transmitted as pulse trains directly from a neighboring M-RPM module: ϕ (crankshaft angle) and σ (sync pulse, every two rotations).

9.4 Software Implementation

9.4.1 Communication

High level communication is performed on the grounds of the predefined data stream networks *collector net* and *distributor net* (see section 9.1.2). Because the implementation targets range from single processor (actually an inverse functional implementation that satisfies the interface requirements of the simulator) to a truly distributed system, these nets are implemented very differently (Fig. 111).

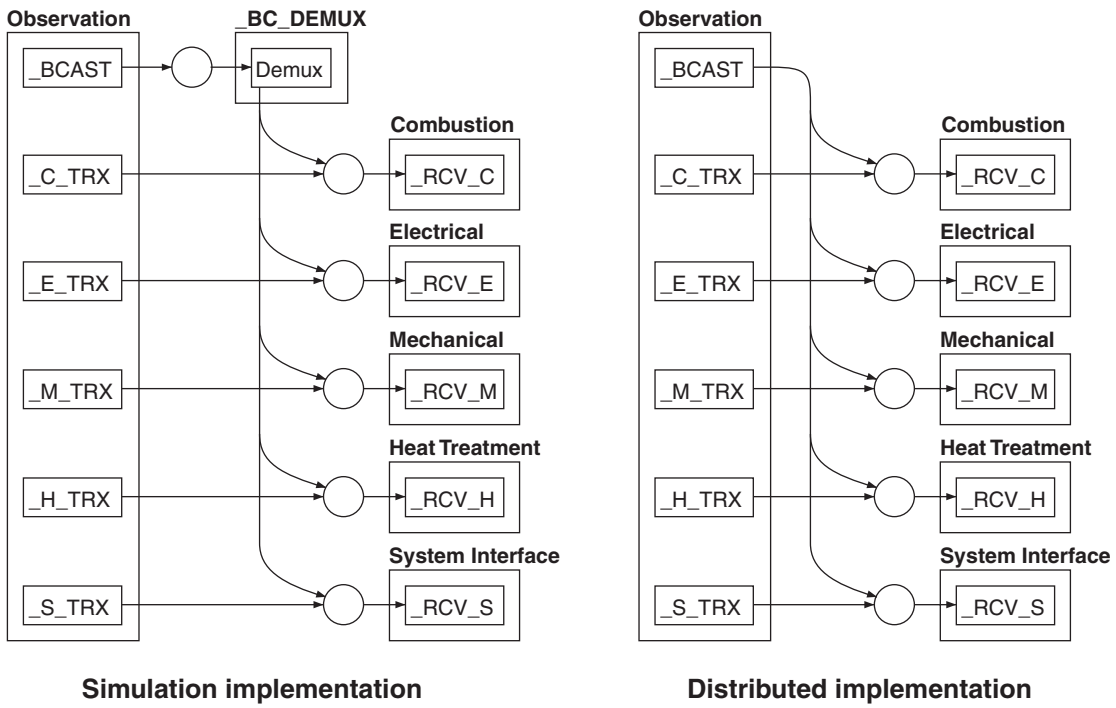

Figure 111: Distributor net implementation variants

Functional code generation is performed for implementation units that include an arbitrary number of a system's clusters. For simulation, a single implementation unit is defined that embraces the entire system, whereas distributed implementations have one unit per processor node. Communication networks that

lie completely within a single implementation unit are not visible at the interface and their implementation is taken care of by the generator. The collector net with its sole point-to-point connections poses no problems, whereas the broadcast facility of the distributor net requires special consideration. While the underlying protocols of the distributed implementation support broadcasting, it must be explicitly modeled for the unified simulation. The necessary demultiplexer is integrated in the model in its own cluster (Fig. 111, left), and this cluster is just exempted from code generation for the distributed cases (right).

9.4.2 System Software and Connection

The two all-embracing connective mechanisms (ring and broadcast) are present on all clusters with respective tasks. Numerous *buffers* decouple global and local actions from each other. Code for all these tasks is identical throughout the entire system. Furthermore, the non-generated code of all the working clusters is identical as well, and grouped around the single *CIP* task where the actual control functions run. The control functions themselves are generated from the functional model. Aside from buffers and multiplexers, there exist other processes, which may or may not be present on individual clusters. The *communicator* operates the bidirectional ring interface. The *broadcast receiver* accepts fresh data packets and updates the global input data vector. The *tick generator* drives local delay timers within the functional block. A single *monitor* task keeps track of communication performance and cluster responsiveness at a low level. The *adaptor* task is needed by all clusters to perform low-level initialization and to respond to global change directives issued by the monitor. Finally, the *abyss* is where misguided connection attempts are routed to.

There are three types of clusters:

Cluster O. Since this unique cluster sits at the top of the operational star topology, its communication interface is characterized by an additional necessary demultiplexer and dedicated buffers for the incoming messages from each working cluster (Fig. 112). While a copy of the input data is available on the cluster, no events other than a timer tick must be dealt with, therefore no event extractor task is present. As in all clusters where generated code is used, the structure is centered around the CIP task.

It is on this cluster that the low-level monitor task runs continuously to ensure system integrity; any non-responding nodes are quickly identified and in some cases appropriate measures taken. Because the ring can be operated in both directions, a single failure does not block all communication. Likewise, a fail-

ure of the broadcast mechanism leads to considerable performance degradation (because only a subset of the input data can be dealt with, which must be passed over the ring), but it does not cripple the input system entirely.

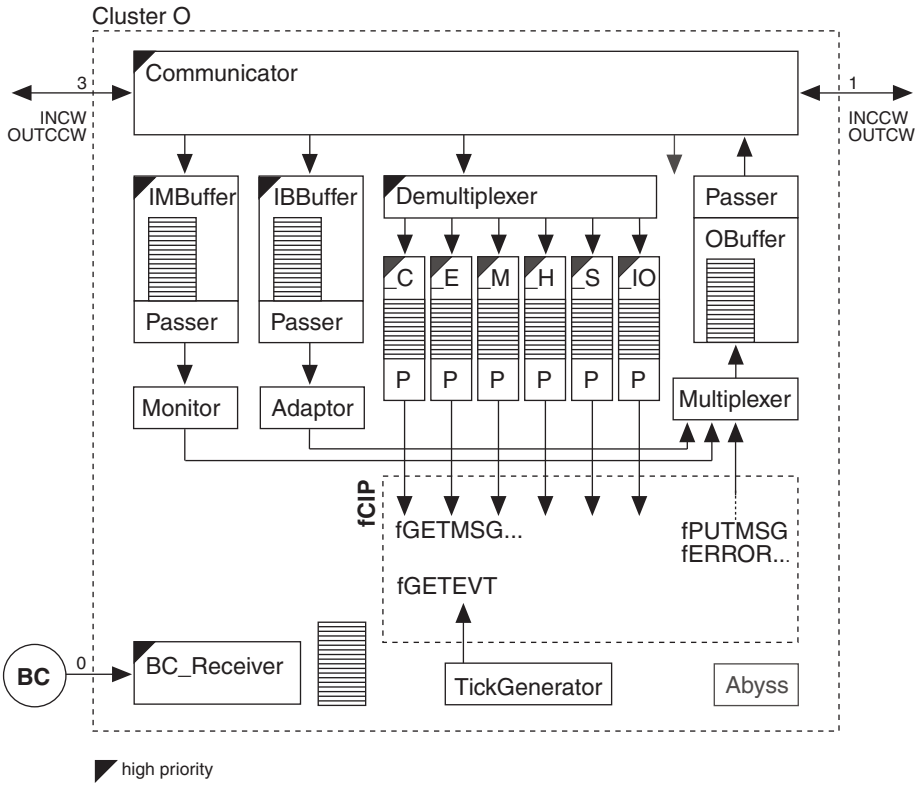

Figure 112: Task structure of cluster O

Cluster IO. This cluster is composed of three dedicated nodes, with the node ACC that performs the accesses to the VME bus playing a major role (Fig. 113). The system's central clock task is also situated here, granting priority of due inputs over pending outputs. To ensure its unhindered operation, high priority buffers are provided in the adjoining nodes. Data contained in the output buffer on node O has been converted and expanded previously by the *output converter* and can be written immediately. Raw input data is passed to the input node I, where conversion, filtering and plausibility testing are performed before a new data package is sent by the broadcast sender.

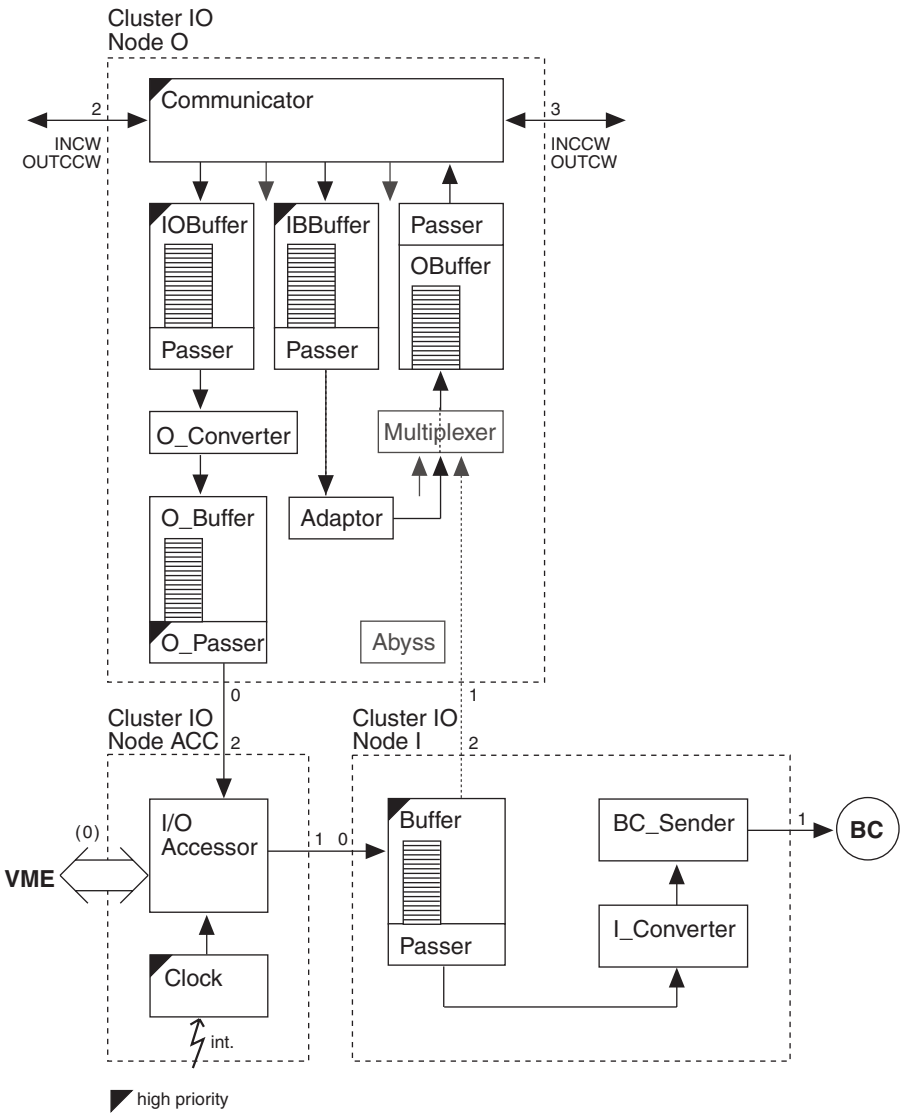

Figure 113: Task structure of cluster IO

Working Cluster. The structure of the clusters C, E, M, H, and S is identical, their two main components being the CIP task and its complementing *event extractor* (Fig. 114). Input data are used in two basically different ways. Either numeric values are used directly, as in the many control algorithms, or they are examined for events contained in a quasi-continuous data stream. These can then be used to trigger the CIP functional reactive block. In the case of binary signals it is sufficient to search for matches of predefined patterns with the newly acquired inputs. If events are contained within continuous signals, the quasi-continuous input sequences must be compared with threshold values. The use of generated tables allowed for an efficient implementation of this otherwise tedious and time-consuming process.

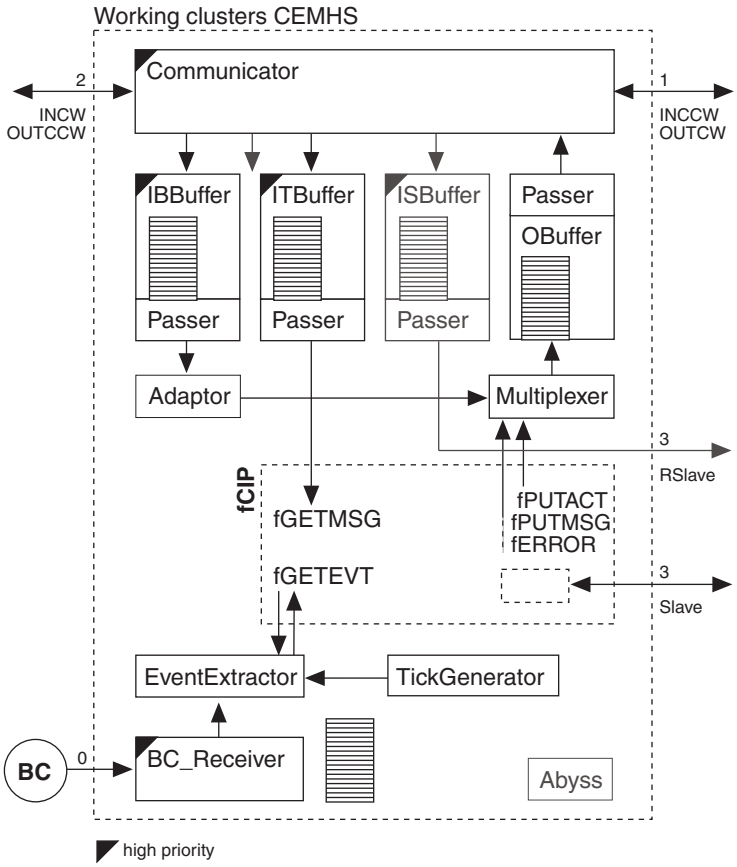

Figure 114: Task structure of clusters C, E, M, H, and S

A cluster's functionality is achieved by combining the outcome, i.e., the generated code, of the functional CIP model and the event configuration provided by the specialists concerned with the respective part of machinery.

Slave. One or several slaves in a row can be attached to the principal working cluster nodes. There are two types of slaves, *plain slaves* are connected to the CIP task, or to another slave situated above itself (Fig. 115), whereas *ring slaves* (RSlaves, Fig. 116) are serviced directly from the ring. Plain slaves expand a cluster's capabilities locally and are owned by their respective masters. Ring slaves can be reached from the entire system with access to the ring, and they act solely as receivers. Because they are connected to a ring input buffer on their master nodes, they may only occupy the first position behind the master. A notable example is the display slave that is associated with cluster S (system interface), but must also be able to display data sent to it from any other cluster.

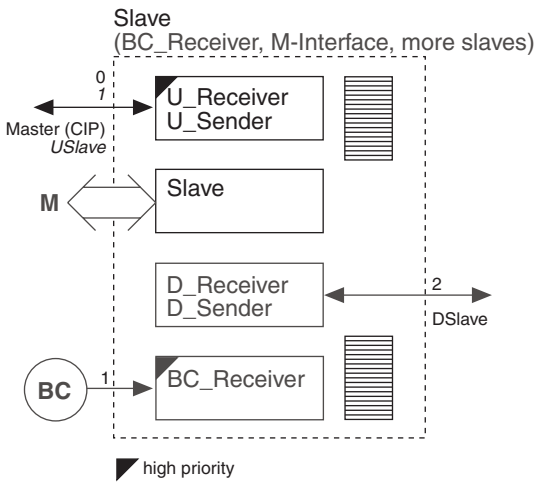

Figure 115: Task structure of a plain Slave

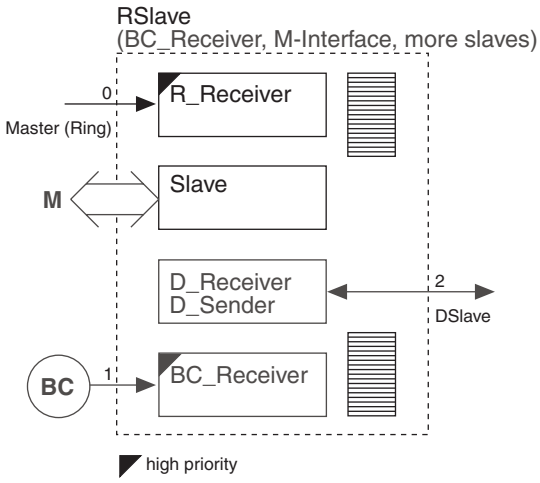

Figure 116: Task structure of an RSlave

The slave concept has proven effective because of the possibility to bring in needed additional resources late in the development process. As with all adaptable system properties, the interface to a particular slave is defined in a configuration. The inner workings of a slave can either consist in plain number crunching, which is the case when entire control loops are placed exclusively on a single processor, or it may contain parts of the working cluster's generated code. With a functional model aptly divided into asynchronous CIP clusters in anticipation of such necessities, the expansion of a simple cluster node to one with additional slaves becomes a straightforward and highly automated task.

While the interface to a slave's master is indispensable (consisting of either `U_Receiver`, `U_Sender` in a plain slave, or `R_Receiver` in a ring slave), some components are optional for both types of slaves: `BC_Receiver` (only

needed if the slave is to receive broadcast input data), an M-interface (if the slave has direct hardware access to local M-module data input and output), and an interface for more slaves (`D_Receiver`, `D_Sender`, if additional slaves are connected further down in the same row, Fig. 117).

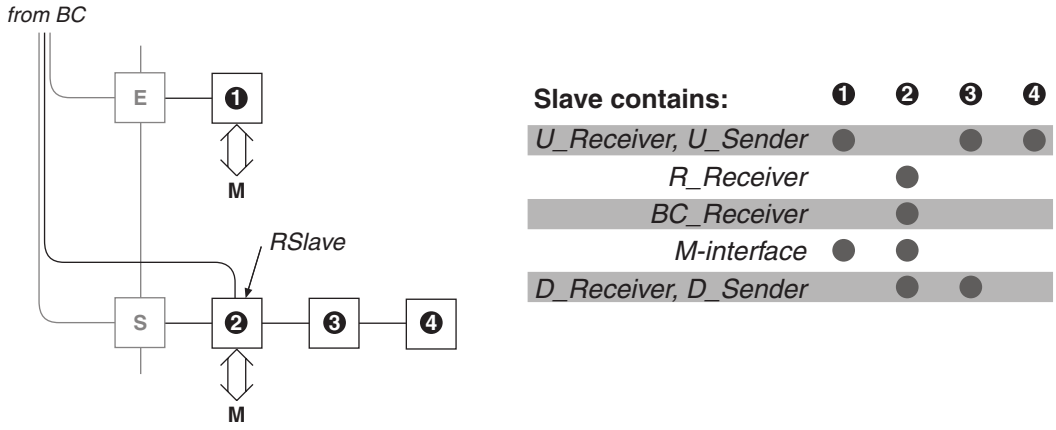

Figure 117: Configurations of master and slave nodes

This part of the project suffered the most from a lack of tools for tracking down errors and for analyzing the performance in a parallel environment. In order to make full use of the available system capacities, task structures and priorities at this low level had to be balanced carefully by hand. The benefit was that these components could be set up and tested early in the development process by just a few specialists. From then on the subsequent problem-oriented chores involved only high-level models and configurations that yielded generated code to fit into an otherwise stable framework.

9.4.3 Functional Code

The entire functionality of a cluster is contained within the single CIP task. It consists of the generated functional code, enclosed by a shell that tunes up the supplied signals and establishes a connection between the input data and the structure present in the functional model. The CIP task's output signals are handled likewise by the shell. After the required conversions they are buffered to be forwarded eventually by the communicator task.

10 Conclusions

Development efforts for computer systems invariably lead to a configuration prescription for a general-purpose processor and the hardware resources attached to it. In the case of embedded systems, this configuration, commonly known as object code, coerces the general purpose processor into behaving like the black box described in the original requirements. It is not easy to arrive at this configuration, and rather than dealing with a single problem, developers must solve a conglomerate of problems from different fields. Unfortunately, these problems are not independent.

'When we try to pick out anything by itself, we find it hitched
to everything else…' (J. Muir [Mui11]).

Nevertheless, the only feasible approach is to try and split up the task into separate problems, and in doing so attempt to reduce mutual interference to a minimum. Some of the overlap between problem areas can be controlled with the choice of a common base, such as the pivotal description of the component architecture. The other elements that help in keeping the various parts aligned are inherent in the code generators.

Within the problem areas themselves, development progresses between these two fixed points. Since there is no single way of describing the problem in all areas at once, a problem-oriented approach furthers the use of appropriate methods and tools for individual solutions. The problems that must be solved call for a diversity of representations, ranging from high levels of abstraction (for the functional model and the connection specification) to code itself (for operations within the functional model and the connection's device access routines). Problem areas clearly delimit the applicability of methods and descriptions, and define interfaces between them. However, for practical purposes, all this would be worthless without the last essential step, namely assembling the collection of parts to form coherent and efficient implementations.

The different representations that are given or needed to arrive at a functional implementation are shown in Fig. 118. Development aims at uniting the preconditions in the top row with the constituents of an implementation at the bottom. Modeling and specification, based on the pivotal component architecture, demand the bulk of development efforts. From there, code production is automatic. Temporal considerations enter the picture only after code can be generated from the CIP model and the specifications for events, as well as input and output. Then, performance measurements of the code itself are used to verify that an implementation performs according to the requirements.

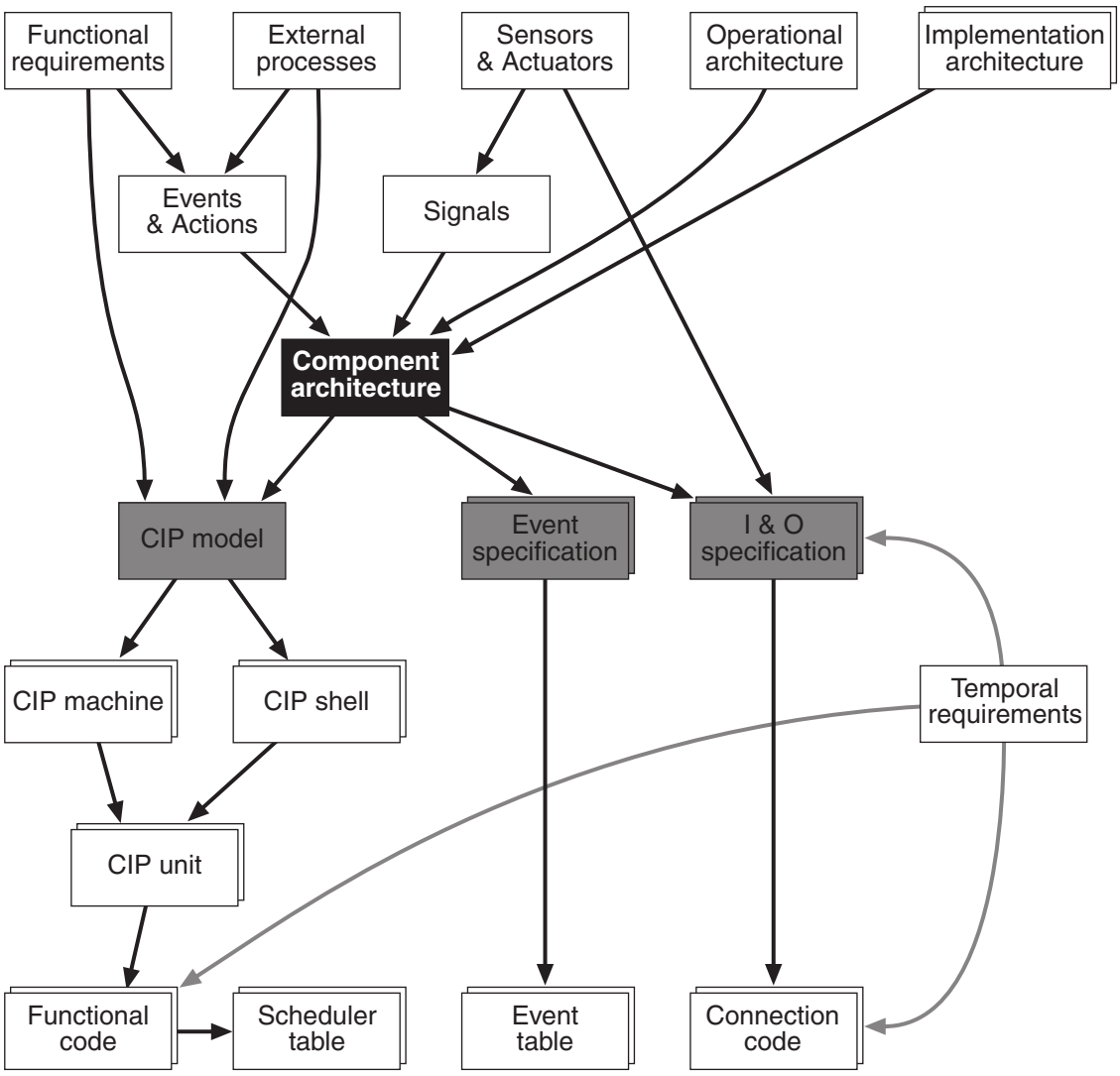

Figure 118: Dependencies

Primarily, Fig. 118 illustrates dependencies, and not development sequences, although development iterations will follow the same lines. The role of the component architecture as the pivotal description is apparent. It is only bypassed by the functional requirements and information about the behavior

of the external processes that are needed to construct the functional CIP model, and by sensor and actuator details used in the connection specification (actually in the guise of their low-level access procedures).

10.1 Complete Solutions

Like in no other area of computer engineering research, real-time embedded systems demand an extremely good knowledge and understanding of the actual application characteristics. Since applications in this field are tied to specific execution environments, or because legal obstacles are in the way, it is very difficult for researchers to gain access to the innards of *actual* real-time systems, i.e., the ones that are in practical use. Therefore, the research community largely fails to perceive many of the major real-time problems and opportunities in the real-time application world. Instead, most efforts concentrate on 'classical' problems, mainly excerpts of the entire problem field that are amenable to formal treatment and abstraction. Messy application details are excluded.

Developers in industry, on the other hand, are subjected to severe constraints on hardware cost, size, weight, and power consumption. The costs of software development in volume applications is negligible compared to hardware, even considering that they are generally higher for real-time in contrast with other applications. This makes developers in industry believe that they neither need nor can afford technology advances from real-time software research. Programs are written in assembly code and deployed with self-made executives or none at all (i.e., the application executes directly on the computer hardware and performs all resource management) [Rei97].

The motivation for the presented work was to bridge this gap. Rather than being cutting-edge, the chosen techniques have stood the test of time. The focus was on simplicity, and the main contribution consists in uniting different methods into a suite of tools with which coordinated progress from requirements to implementations becomes possible. The main traits are:

- development of *entire systems* is placed within a stable context based on decomposition (with the pivotal component architecture as its center),
- each of the collection of problems and the developers dealing with them are taken seriously (by allowing descriptions that are appropriate to specific problems),
- the final assembly leads to implementations that are efficient, easy to implement, statically deterministic, having known timing characteristics (through generated code with strictly defined execution properties).

The concept has been proven in a number of industrial and academic projects, where different partitions of the same model had to be connected to simulation models, to test beds for specific system parts, and to the real environment of the final target system. Simulating entire control systems against numerical models eliminates the need for possibly dangerous experiments with the target system and greatly reduces the initial operation phase by granting control system developers a lead.

10.2 Implementation

Introducing parallel entities into the design of control systems is a well established method when the focus is on specific aspects in a complicated system. While the idea of parallelism is helpful for disentangling complex problems in a conceptual stage, there is hardly any reason for the common practice to implement such a solution using different tasks when they all run on a single processing entity. If the conceptual parallelism cannot be exploited by parallel hardware, it is advantageous to replace it with a single corresponding sequential task. While this may not be easy or even feasible for a set of parallel tasks written in a programming language, it poses no problems for a code generator. Input and output generally use common hardware resources that often impose sequential use. With specifications and code generators, efficient solutions for hardware access become possible. The generators combine models and specifications from a high level of abstraction with low-level elements (behavioral descriptions and C code). Since deterministic code results from these efforts, what remains to be done is verifying that the total processing time is sufficient; there is no need for scheduling more than necessary!

The correlation of distinct modes of operation and resource utilization is known in advance for many embedded control systems. This a priori knowledge can be used to devise implementations that are well adapted to a specific operation environment. Optimizations are achieved through dynamic reconfiguration, and they either permit a reduction of the required hardware resources, or allow for this extra amount of processing power or memory that a certain component needs. In the presented approach, such optimizations are possible within the tight constraints of a static environment. Control remains at the language level, but without the need of making recourse to the dynamic memory management a language may provide.

10.3 Reuse

The usefulness of distinct problem areas is also noticeable with regard to software reuse. Tools let developers work with the abstract categories they are at ease with and which they consider for reuse. If abstraction is the prerequisite to software reuse [Weg87], then code generation closes the gap between an abstract model and its implementation. Automating the link between a high level of abstraction and the implementation eliminates situations where an otherwise fitting component cannot be used because of the ways it has been implemented. Namely, if abstract component definitions are associated with manually written code fragments, it is not sufficient to meet the architectural constraints. The code must be appropriate for the envisaged context as well, or only the component's structure could be reused. Code generation greatly increases the chances for component reuse without adaptations; it thus helps reduce the number of variants to monitor and maintain.

Appropriate tools favor the creation of detachable, self-contained components. Functional models that reflect the behavior of a component in the environment are especially fit candidates for finding generic solutions to recurrent problems. For example, a mechanical switch simply corresponds to its counterpart in the model, which will be used with the same confidence as its match. Customarily, elements needed to attach a functional solution to specific environments are difficult to reuse. Such components can be included unaltered in various environments only together with code generators [Tru99b].

10.4 Suggestions for Further Work

Further work is required to improve the integration with available repository management utilities. Instead of subscribing text files that must be imported in an additional step, subscription from within the modeling environment is envisaged. Whereas black box reuse is not attracting developer groups working within the same organization, it may well become an issue, if components are to be sold for reuse by someone else. To this end, suitable means to extract the essence of a component without revealing proprietary information must be investigated.

A Specification

This chapter describes the specification languages for the architectural pivot and for the connection (access specification for input and output; event extraction specification), as well as the generators and their use in creating the connection code. The description is meant to be more accessible than the specification syntax (see appendix B). The generators used to create implementations for system input, output and event extraction work on textual input files that contain configurations in the appropriate notation. They are C programs without an interactive user interface and as such platform-independent. Although the use of plain specification languages without the bells and whistles of graphic user interfaces has gone out of fashion, it is appropriate for the type of problems at hand[1]. Besides, the choice of this representation offered several other advantages:

- instead of struggling with the intricacies of programming a graphic user interface, efforts could be concentrated on solving algorithmic problems,
- program structures are based on tried and tested compiler technology,
- being platform-independent, the tools integrate smoothly into different existing development environments,
- a list representation is perfectly adequate to the catalogs of abstract signal names and hardware items that the connection problem has to deal with,
- because the specifications themselves are text files, the commodities of C preprocessors and the powerful features of modern program text editors and project management tools become available automatically.

1. This became apparent while working on a first generation of one of the tools that was programmed in an object-oriented environment with a fully interactive user interface. Because there the most suitable view of a configuration proved to be the searchable generated list summary, it was decided to adopt list formats also for the specifications themselves.

A.1 Architectural Specification

The architectural pivot is described with a notation that compares unfavorably against known and much more powerful 'real' architecture description languages (for an overview see [All97]). The limitations of the chosen notation make sense, because solutions attempted along the lines of the problem decomposition from chapter 3, and with code properties as described in chapter 7, effectively restrict the number of required different component and connector types. Furthermore, through the strict separation of concerns their behavior and ways of interactions are fixed on the architectural level.

A.1.1 Specification Language

The specification serves two purposes. It provides a clear view of a system's component architecture for its human developers and users, and conveys system-wide implementation properties to the various tools used in constructive development. Although no code is produced directly from the description, implementation details that will be needed by the constructive tools are included in the notation.

Keywords:

`VERSION`	specification version,
`CONSTANT`	constant definition,
`TYPE`	type definition,
`COMPONENT`	functional or connective component type,
`PORT`	component interface,
`MESSAGE`	port message,
`COMPUTATION`	component computation,
`CONNECTOR`	way of interaction between components,
`ROLE`	connector interface,
`GLUE`	connector glue,
`INSTANCE`	instantiation of component items,
`ATTACHMENT`	interaction topology association,
`VOID`	(reserved).

The syntax permits the use of:

- characters (case sensitive),
- numbers in the formats `int` (`1`), `float` (`1.0`), hexadecimal (`0x1`), binary (`0b1`), `int` and `float` can be negative,
- `,` as divider,
- comments (`/*...*/`) may be used where whitespace is legal.

A.1.2 Productions

The productions are described using the above keywords and the following terminal symbols:

`ident`	any valid C identifier,
`stdtype`	a C standard type,
`cardinal`	positive integer number,
`number`	number as defined above, at times `int` required,
`procedure`	function call with (actual or wildcard) parameters,
`string`	character string delimited with `""`.

Architecture production:

```
Version {Constant | Type | Component | Connector}
{Instance} {Attachment}
```

Version. Specification version in a free format (possibly dictated by a project management tool).

Production:

```
"VERSION" string.
```

Semantics:

```
VERSION version
```

Example:

```
VERSION "940503"
```

Constant. Constant declaration. The constant is available for use in definitions within the pivot specification itself and is also passed to client tools.

Production:

```
"CONSTANT" ident number [string].
```

Semantics:

```
CONSTANT const_name const_value "comment (optional)"
```

Examples:

```
CONSTANT TTICK 3125 "CIP timer tick (* 64 [us])"
CONSTANT I_VAL_SIZE 87 "number of input values"
```

Type. Type declaration for types used in component and connector interfaces. The item `type` appearing in the semantic description is either a standard type or a previously defined custom type. The declaration takes different forms for simple and structured types.

Production:

```
"TYPE" (ident | stdtype) ident [string].
```

Semantics:

```
TYPE type type_name "comment (optional)"
```

Examples:

```
TYPE int counter_t "counter type"
TYPE struct ASTRUCT aValue_t;
```

Production:

```
"TYPE" ("struct" | "union") [ident] [string]
"{" {([("struct" | "union")] ident | stdtype)
ident {"," ident} ";"} "}" ident.
```

Semantics:

```
TYPE tag  tag_name(optional) {
  tag(optional) type member_name1, ..;
  tag(optional) type member_name2;
  ..}
type_name "comment (optional)"
```

Example:

```
TYPE union {float f; int i;} fi_t; "input item type"
```

Production:

```
"TYPE" (ident | stdtype) ident
"["(number | ident) {"," (number | ident)}"]" [string].
```

Semantics:

```
TYPE type type_name [Size1, ..] "comment (optional)"
```

Examples:

```
TYPE int Hist_t [5,5] "field history type"
TYPE fi_t I_Data_t [I_VAL_SIZE] "input vector type"
```

Component. Component type declaration. Components are either functional or connective. Either way, a component's interface consists of an arbitrary number of ports capable of sending or receiving messages of a specific type.

Production:

```
"COMPONENT" ident [string] {Port} Computation.
```

Semantics:

```
COMPONENT component_name "comment (optional)"
port1 port2 .. functionality
```

(See below for an example of a complete component declaration).

Port. A component's port declaration. Aside from the direction of data flow, ports are characterized by the messages they produce/consume and the types of these message.

Production:

```
"PORT" ("IN" | "OUT") ident [string] {Message}.
```

Semantics:

```
PORT (IN | OUT) port_name "comment (optional)"
message1 message2 ..
```

Message. Declaration of messages that are associated with a specific port. Messages are typed and may be associated with interface functions. An interface function for messages of an input port needs to be called to deliver the message. In the case of output ports, the interface function is called by the component itself, i.e., it must be defined in the component at the other end of a specified connection. The keyword `VOID` supplied as `function_name` declares a message an untyped signal.

Production:

```
"MESSAGE" ident (VOID | (ident
ident "(" ident {"," ident} ")")) [string].
```

Semantics:

```
MESSAGE message_name function_name
return_type (param_type, ..) "comment (optional)"
```

Computation. A component's computation declaration. For functional components, `unit_name` is the name of the CIP implementation unit that represents the constructive solution of the component. For connective components, the `unit_name` is the name of the compilation unit.

Production:

```
"COMPUTATION" ident [string].
```

Semantics:

```
COMPUTATION unit_name "comment (optional)"
```

Component example:

```
COMPONENT  CIPM  "CIP functional machine"
  PORT IN  iPort "1st input port"
    MESSAGE  on  C2_on  void (int) "ICmU0A on unit 4"
    MESSAGE  off C2_off void (void)
  PORT OUT oPort "1st output port"
    MESSAGE  on  C7_on  void (aValue_t)
    MESSAGE  off C4_off int  (void)
  COMPUTATION  MotorController "DC motor controller"
```

Connector. Connector type declaration.

Production:

```
"CONNECTOR" ident [string] {Role} Glue.
```

Semantics:

```
CONNECTOR connector_name "comment (optional)"
role1 role2 .. interaction
```

(See below for an example of a complete connector declaration).

Role. Declaration for the roles of components that will be associated by a specific connector. Roles could contain the message types that a component must be able to deal with when attached to a specific connector, as well as permissi-

ble traces defined on the set of messages. Since the roles are inherent in the connectors used in the component architecture, roles are merely descriptive.

Production:

```
"ROLE" ident string.
```

Semantics:

```
ROLE role_name "role description"
```

Glue. The connector glue declaration has only descriptive character. It could hold connector properties in a style of a more general architecture description.

Production:

```
"GLUE" string.
```

Semantics:

```
GLUE "glue description"
```

Connector example:

```
CONNECTOR Simple "one to one unidirectional connector"
  ROLE Caller "initiates function call"
  ROLE Callee "performs computation"
  GLUE "Caller invocation sequence is preserved"
```

Instance. Instantiation of components and connectors. Types would be more useful if the component architecture supported parameters. Even if an express functional component type is required for every instance (because functionality, ports and messages depend on the context), explicit instantiation has its uses for the more universal connective components and connectors.

Production:

```
"INSTANCE" ident ":" ident.
```

Semantics:

```
INSTANCE instance_name :
(component_name | connector_name)
```

Example:

```
INSTANCE CIP_Machine: CIPM
INSTANCE I_CIP: Simple
INSTANCE O_CIP: Simple
```

Attachment. Connector attachment declarations are only descriptive.

Production:

```
"ATTACHMENT" ident "." ident "AS" ident "." ident.
```

Semantics:

```
ATTACHMENT
instance_name.port_name AS instance_name.role_name
```

Examples:

```
ATTACHMENT CIP_Machine.iPort AS I_CIP.Callee
ATTACHMENT CIP_Machine.oPort AS O_CIP.Caller
```

A.2 Input and Output Specification

The purpose of this specification is to arrive at a deterministic and optimized implementation from a description that is based on the hardware items on the one hand, and on the input and output signals as used in the functional model on the other hand. In the running system, input values can be retrieved from a vector where each signal is updated periodically according to its timing requirement and with practically no input jitter. Access macros are provided to guarantee type integrity. Likewise, output signals are sent using generated macros.

For reasons of optimal utilization it may be advantageous that the resources needed to perform a system's input and output operations be centralized. Oftentimes, this results in a single such specification for an entire system, although several distinct collections with separate specifications are also possible. Additional data structures are generated that are required for the transmission of input and output data in distributed systems. The choice of a suitable means of data forwarding depends on the hardware architecture and the preferences in a specific implementation. Various transmission schemes are supported, such as point-to-point forwarding and broadcast via dedicated links or shared memory.

A.2.1 The Generator 'acc'

Specification data from a textual input file (defaults to `IO.config`) is used to create the files `IO.c` (containing the generated code), `IO.h` (the pertaining header file), and `IO.lst` (a summarizing hardware and software configuration description for documentation purposes).

Call Syntax:

```
acc [input_file]
```

A.2.2 Specification Language

The specification language reflects a hardware-oriented view of the system configuration. Rather than generate the entire code from an abstract description, it serves as a means of integrating already tested and profiled low-level code routines or macros into the overall structure of a universal input and output system. This purpose is reflected in the input language, which incorporates arbitrary function calls including parameters, but does not itself permit the specification of functionality. Keywords, identifiers and numbers are complemented by a set of wildcards that are needed for parameter substitution.

Keywords:

`VERSION`	specification version,
`SAMPLING`	sample frequencies,
`INCLUDE`	file inclusion,
`CONSTANT`	constant definition,
`INIT`	initialization,
`CARRIER`	carrier board,
`MODULE`	mezzanine module,
`HI, MED, LOW`	input signal,
`OUTPUT`	output signal,
`PATTERN`	mask and pattern definition,
`VOID`	(reserved).

The syntax permits the use of:

- characters (case sensitive),
- numbers in the formats `int` (`1`), `float` (`1.0`), hexadecimal (`0x1`), binary (`0b1`), `int` and `float` can be negative,
- function- (or macro-) calls with parameters,
- wildcard characters for various automatically substituted parameters:

`.b`	carrier board address (base),
`.s`	mezzanine module number (slot number `0...3`),
`.c`	'channel setup value' (for the actual channel),
`.n`	'channel setup value' (for the next channel of same device),
`.hd`	raw data (hardware data),
`.sd`	converted data (software data),
`.l`	local static storage,
`.t`	temporary storage,
`.m`	mask (for binary signals),

- `,` as divider,
- comments (`/*...*/`) may be used where whitespace is legal.

A.2.3 Productions

The productions are described using the above keywords and the following terminal symbols:

`ident`	any valid C identifier,
`cardinal`	positive integer number,
`number`	number as defined above, at times `int` required,
`procedure`	function call with (actual or wildcard) parameters,
`string`	character string delimited with `""`.

I/O production:

```
Version Sampling {Include | Define | Init} {Carrier}
```

Version. Defines the specification version in a free format.

Production:

```
"VERSION" string.
```

Semantics:

```
VERSION version
```

Example:

```
VERSION "590202"
```

generates the line

```
#define VERSION "590202"
```

Sampling. Used to declare the sampling frequencies (in [Hz]). Strictly speaking, the relation among the three frequency classes `HI`, `MED`, and `LOW` are defined. It is the responsibility of the caller in the implementation to match the highest (or base) sampling rate with the one defined for class `HI`.

Production:

```
"SAMPLING" cardinal cardinal cardinal.
```

Semantics:

```
SAMPLING HI_Freq MED_Freq LOW_Freq
```

Example:

```
SAMPLING 1000 100 10
```

The sampling frequencies are set to 1 kHz, 100 Hz, and 10 Hz for the frequency classes `HI`, `MED`, and `LOW`, respectively (using a base sampling rate of 1 kHz).

Include. Inserts a preprocessor directive (with optional comment).

Production:

```
"INCLUDE" string [string].
```

Semantics:

```
INCLUDE "file_name.ext" "comment (optional)"
```

Example:

```
INCLUDE "MLib.h" "mezzanine module definitions"
```

generates the line

```
#include "MLib.h" /* mezzanine module definitions */
```

Constant. Preprocessor constant declaration.

Production:

```
"CONSTANT" ident number [string].
```

Semantics:

```
CONSTANT const_name const_value "comment (optional)"
```

Example:

```
CONSTANT OCmA_MAX 0x800 "max. value for ICX outputs"
```

generates the line

```
#define OCmA_MAX 0x800 /* max. value for ICX outputs */
```

Init. Declaration of initialization function calls for global settings as well as for individual carrier boards and single modules.

Production:

```
"INIT" procedure {procedure}.
```

Semantics:

```
INIT Init_fct_1(p1,p2,..) Init_fct_2(p1,p2,..) ..
```

Example:

```
INIT vme_address_modifier(0x3d)
```

A call to this function will be inserted into the system initialization function `IO_Init()`. The statement may be followed by an arbitrary number of other initialization calls.

Carrier. Declares a carrier board and its base address. Carrier board declarations can also be used for declarations concerning cards with their own functionality, i.e., additional mezzanine module components are optional. When a carrier does not contain any modules, its own functionality is optional.

Production:

```
"CARRIER" ident number [string] {Module}
{(Input {Pattern}) | (Output {Pattern)}.
```

Semantics:

```
CARRIER name base_address "comment (optional)"
  MODULE ..
  HI .. PATTERN ..
  OUTPUT .. PATTERN ..
```

Example:

```
CARRIER R__VA26 0xaaa800 "this card is a joker" MODULE
```

This declares the carrier board `R__VA26` having base address `0xaaa800`, together with the given comment.

Module. Declares a mezzanine module including its position on the carrier board (slot 0…3). Modules are related to the specific carrier board that precedes the module declaration. A single module may offer both input and output functions.

Production:

```
"MODULE" ident cardinal [string].
```

Semantics:

```
MODULE name slot "comment (optional)"
```

Example:

```
MODULE R__M331 0 "16ch AD mezzanine module"
  MED ..
```

This declares the mezzanine module `R__M331`, in slot `0` on its carrier.

HI, MED, LOW (Input). Input signal acquisition can be a lengthy process that depends on the kind of signal and especially on the peripheral hardware items involved. To cater for the majority of possible combinations, signal inputs are defined as sequential processes like in Fig. 119. Some of the stages are optional.

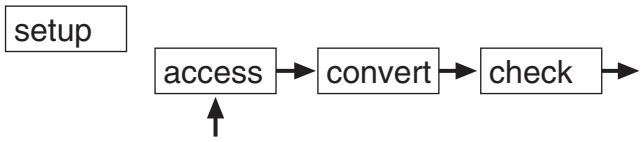

Figure 119: Input phases

Inputs are qualified by their frequency class and by the following information:

Table 4: Input configuration items

signal name
simulation model interface vector index
setup value
setup function call, with execution time and delay time [µs]
access function call, with execution time [µs]
conversion function call
check function call

Production:

```
"input class" ident cardinal number
  procedure, cardinal, cardinal
  procedure, cardinal
  procedure
  procedure.
```

Semantics:

```
( HI | MED | LOW ) name simline ch_setup
  I_Setup_fct(..), T_setup, T_wait
  I_Acc_fct(..), T_acc
  I_Conv_fct(..)
  I_Check_fct(..)
```

Example:

```
HI ICmP03 22 101
  M34_st_set_mode(.b,.s,.n),3,7
  M34_rd(.b,.s,.hd),3
  I_F(.sd,.hd,0.000488,-0.4)
  VOID
```

This specification describes an input signal of class `HI` with initialization value `101` and the three setup, access, and conversion function calls. The keyword `VOID` indicates an empty check function. Function parameters are either constants or wildcard characters that will later be substituted by the generator. The execution and delay times needed in generating the access schedule must be known beforehand for all functions (i.e., determined by profile measurements). In a simulation implementation, this input signal would be connected with the simulating environment via vector item `22`.

Output. Similar to the sequence defined for data acquisition, outputs are seen as consisting of at most three separate stages, all of which are optional but for the access call itself.

Figure 120: Output phases

To declare an output signal, the following information is required:

Table 5: Output configuration items

signal name
simulation model interface vector index
setup value
conversion function call
setup function call, with execution time and delay time [µs]
access function call, with execution time [µs]

Production:

```
"OUTPUT" ident cardinal number
  procedure
  procedure, cardinal, cardinal
  procedure, cardinal.
```

Semantics:

```
OUTPUT name simline ch_setup
  O_Conv_fct(..)
  O_Setup_fct(..), T_setup, T_wait
  O_Acc_fct(..), T_acc
```

Example:

```
OUTPUT O__A04 14 5
  B_I(.hd,.sd)
  VOID
  ID_M33_wr_data(.b,.s,.c,.hd),3
```

This describes an output signal with initialization value `5` and the necessary conversion and access function calls. The setup function is empty. In a simulation, the signal would be connected via vector item `14`.

Pattern. Patterns, together with the related masks, belong to a specific binary input or output signal. The mask name (and its value) is followed by an arbitrary number of pattern names with their values. The mask name will also be used as a prefix to create the final names for the patterns.

Production:

```
"PATTERN" ident number cardinal {cardinal}
ident number {ident number}.
```

Semantics:

```
PATTERN prefix mask_value simline(s)
pattern1 value1 pattern2 value2 ..
```

Example:

```
PATTERN  OHaW1 0x0003 4 5
off 0  close 0x0001  open 0x0002
```

The mask `OHaW1_MASK` is defined including the patterns `OHaW1_off`, `OHaW1_close`, and `OHaW1_open`. Since two bits are set in the mask, two simulation interface lines must be specified (`4`, `5`), higher order bits first.

A.3 Event Extraction Specification

Each working node in a distributed system (or the sole one in a single processor installation) possesses its own event extractor mechanism. This specification has the advantage of preserving the close relation of event messages and their names to a node's functional model. It dissociates the node in question from the entire project and makes it easier to adjust to local changes. This is especially rewarding when different teams work with partial installations on their own test beds. Besides, many events are only of use for a specific node, and broadcasting them would encumber communication channels unnecessarily.

A.3.1 The Generator 'evt'

Configuration data from a textual input file (defaults to `event.config`) is used to generate the header file `event.h`. This header file needs information from and works only in conjunction with the files `IO.c` and `IO.h` that are unique for a system (see section A.2.1).

Call Syntax:

```
evt [input_file]
```

A.3.2 Specification Language

The specification language includes a number of keywords to select a node's local properties, as for example the resolution of the local delay timers and the presence of slave nodes. The bulk of a specification centers around the events themselves. The generator is used to translate this information into standardized data tables that will be used by optimized routines in an implementation. Not only does generation of these data permit the enforcement of important data properties that would otherwise have to be checked at run-time, it also offers the advantage of a convenient and intuitive input format.

Keywords:

`VERSION`	specification version,
`NODE`	node identifier,
`TTICK`	the node's basic timer tick,
`DISPLAY`	switch to enable a node's display features,
`SLAVE`	switch for nodes with a plain slave,
`RSLAVE`	switch for nodes with a ring slave,
`BEVENT`	specifies a discrete (binary) event,
`CEVENT`	specifies an event contained within a continuous signal,
`VOID`	(reserved).

The syntax permits the use of:

- characters (case sensitive),
- numbers in the formats `int` (`1`), `float` (`1.0`), hexadecimal (`0x1`), binary (`0b1`), `int` and `float` can be negative,
- signal names (as used in the input/output specification),
- channel (or process) identifiers,
- event identifiers.

A.3.3 Productions

The productions are described using the above keywords and the following terminal symbols:

`ident`	any valid C identifier,
`cardinal`	positive integer number,
`number`	number as defined above, at times `int` is required.

Event production:

```
Version Node Timer [Display] [Slave] {BEvent | CEvent}
```

Version. Defines the specification version in a free format.

Production:

```
"VERSION" string.
```

Semantics:

```
VERSION version
```

Example:

```
VERSION "521101"
```

generates the line

```
#define VERSION "521101"
```

Node. This production is used to identify the node that the current event specification belongs to. In an implementation, local timers will be initialized with a random number based on this identifier to prevent the accumulation of expiring timers throughout the entire system. Additionally, identifiers are created by which the node is individualized in the global scope.

Production:

```
"NODE" ident.
```

Semantics:

```
NODE name
```

Example:

```
NODE CC
```

The node's identity is `CC`, which gives rise to the following definitions:

```
#define NODEID CC
#define _SAMPLER _SMP_CC /* this node's sampler proc */
```

Timer. Creates a constant needed for the initialization of the local timer interval counter. Delay timer ranges used in the functional model directly depend on this value, which unfortunately is implementation-dependent. With a basic timer tick interval of 64 µs, for example, the following values apply:

Table 6: Delay timer values

3125 ≈ 0.2 s
625 ≈ 0.04 s
125 ≈ 0.008 s etc.

Production:

```
"TTICK" cardinal.
```

Semantics:

```
TTICK interval
```

Example:

```
TTICK 3125
```

Display. The presence of this keyword indicates that the node uses global display features[1], in which case the required code fragments for communicating with the display node are inserted. This production applies only to distributed systems.

Production:

```
"DISPLAY".
```

Semantics:

```
DISPLAY
```

Slave. These keywords indicate if a node possesses additional private (`SLAVE`) or public (`RSLAVE`) slave nodes. If either keyword is present, code fragments for private or public communication with the slave node(s) will be provided, respectively. This production applies only to distributed systems.

Production:

```
"SLAVE" | "RSLAVE".
```

Semantics:

```
SLAVE | RSLAVE
```

BEvent. Events from binary input data are raised when ever the input has changed and one of the defined patterns (see p. 189) is recognized in the input data. This production establishes a connection between patterns and event messages that are understood by the functional model. Aside from the signal and event identifiers, a channel is specified. This channel's message alphabet must contain the used event messages.

Production:

```
"BEVENT" ident ident (ident ident {ident ident}).
```

Semantics:

```
BEVENT Signal_ID Channel_ID (Event Value {Event Value})
```

Example:

```
BEVENT ICoP01 Evt_Chan
  (P_rises ICoP01_normal
   P_sinks ICoP01_low)
```

Entries in two tables will be generated. The first one is the binary event control table containing references to the input signal, message channel, and binary event data table, as well as the signal mask `ICoP01_MASK`, the number of defined events and a variable field where the value that was last seen will be stored. The binary event data table contains the events themselves and their associated patterns.

1. Another tool (dsp), which is not described here, is used for specifying a system's display functionality. It is based on the notion of an arbitrary number of screens that are populated by instruments taken from a toolbox. The specification contains display properties, such as display formats, refresh rates, warning thresholds etc. Implementations are also created by a code generator.

CEvent. The occurrence of events depending on quasi-continuous input data is related to threshold values. These thresholds can be set within the entire signal range, although they must conform to certain conditions:

- the way the event extractor algorithm is implemented requires a hysteresis. Therefore, two signal values must be specified for each threshold,
- thresholds must be given in ascending order,
- even in the case where only one transition will give rise to an event, both values are mandatory! In these cases, the keyword `VOID` replaces a valid event name.

Production:

```
"CEVENT" ident ident (ident number ident number
                      {ident number ident number}).
```

Semantics:

```
CEVENT Signal_ID Channel_ID
  (DownEvent DownValue UpEvent UpValue
  {DownEvent DownValue UpEvent UpValue})
```

Example:

```
CEVENT ICmT03 Evt_Chan (freeze   -0.1  dew    0.1
                        condense 99.0  evaporate 101.0)
```

As for binary events, entries in two tables will be generated. The first is the continuous event control table (allegedly a slight misnomer…) with references to the input signal, message channel, and continuous event data table. This last pointer will be updated during operation to reflect the range where the signal is currently in. The continuous event data table contains an ordered list of events and their associated threshold values. The lower and upper limits are marked with `-FLT_MAX`, `FLT_MAX`, respectively, allowing for an efficient implementation of the extractor algorithm.

A.4 Code Generator Program Structure

All the generators are built along the same lines, even sharing some of the modules. As an example, the structure of the program `acc` is presented in more detail (Fig. 121). It consists of five modules:

- the scanner (`scanner.c`, `scanner.h`), extracts tokens from character sequences and emits error messages while reading the input file,
- the parser (`parser.c`), performs syntactic analysis and inserts read data into the symbol, access procedure and initialization tables,
- the input access distribution scheduler (`schedule.c`), computes the static schedule for data input, taking into account the different frequency classes,

- the module data (`data.c`, `data.h`), manages static data; moreover, `data.h` contains all global definitions and declarations,
- the file generator (`access.c`), performs final parameter substitutions and generates the output file; it also contains the main program.

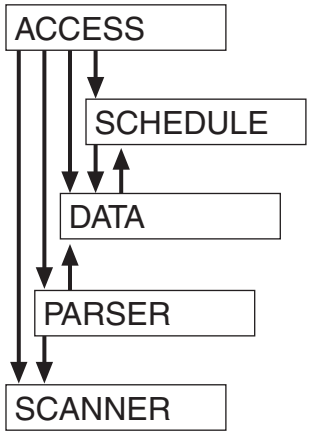

Figure 121: Code generator program structure (acc)

A.4.1 Data Structures

Symbol Table. The symbol table contains all addresses, constants, signals, masks, and patterns. The item types correspond to the elements indicated by keywords in the specification language.

`S_HEADER`	first element (empty)
`S_INCLUDE`	include file
`S_CONST`	constant
`S_CARRIER`	module carrier board
`S_MODULE`	mezzanine module
`S_CHAN`	signal
`S_MASK`	mask (of a binary signal)
`S_PATTERN`	pattern (belongs to a mask)

It is implemented as a list of lists with elements defined as follows:

Table 7: ST (symbol table) data fields

Field name	Description
`char *name`	symbol name
`char *rem`	remark or comment
`int a,b,c,d`	symbol data
`int type`	symbol type
`struct ST *up`	pointer to parent symbol
`struct ST *dwn`	pointer to child symbol (list)
`struct ST *next`	pointer to other symbols of the same type

Depending on the symbol table type, data fields `a...d` are used in specific ways, and they may also be empty. Items in parentheses are optional, their use depends on the signal type:

Table 8: ST (symbol table) data field usage

type	a	b	c	d
S_HEADER				
S_INCLUDE				
S_CONST	const value			
S_CARRIER	address	card number		
S_MODULE	slot number	slot id		
S_CHAN	max_hi_cnt idx	-1	HI	(I_Hist[]idx)
	-1	max_i_cnt idx	MED, LOW	(I_Hist[]idx)
	O_Acc index	O_Bin[] idx	OUTPUT	
S_MASK	mask value	HI, MED, LOW	I_Val.value[]idx	
	mask value	OUTPUT	O_ConvSend idx O_Acc idx	
S_PATTERN	pattern value			

Example Specification. The configuration in Fig. 122 contains three mezzanine modules (`M342`, `M331`, and `M313`) on two different carrier boards (`R__VA28`, `R__VA2A`). `IElI01`, `IElI02`, `IElI03`, and `OElA05` are continuous input or output signals, respectively, and `ICoP1`, `IElC2` are binary input signals including their masks `ICoP1_MASK`, `IElC2_MASK` and the patterns `ICoP1_high`, `ICoP1_low`, and `IElC2_ON`, respectively. The internal representation of this setup is shown in Fig. 123.

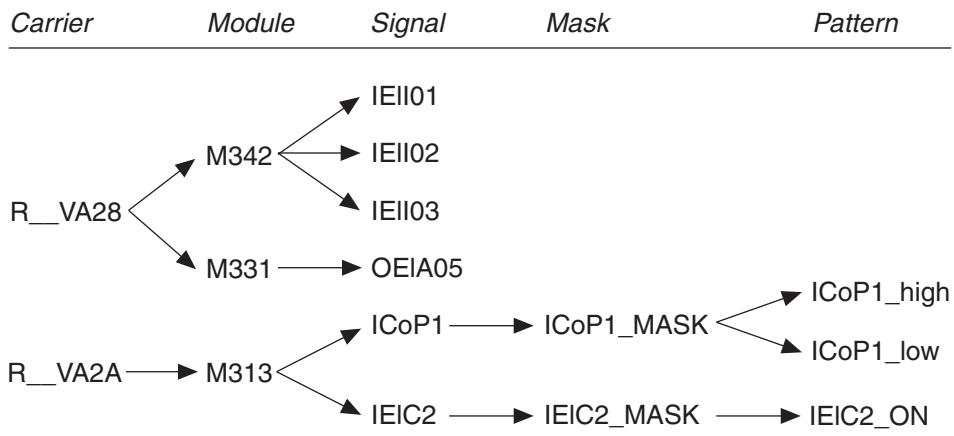

Figure 122: Example signal configuration

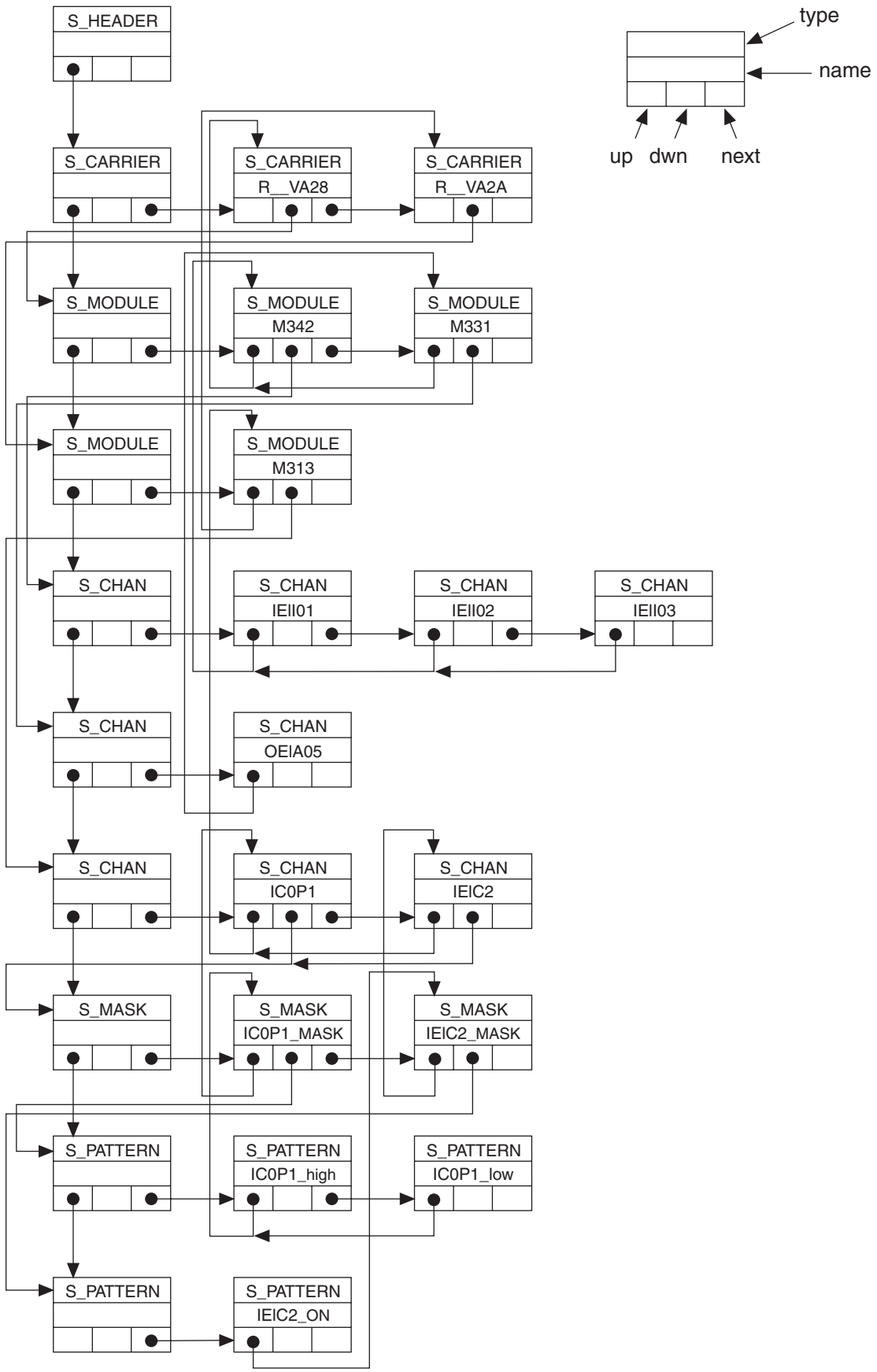

Figure 123: Internal signal representation

A.4.2 APs (Access Procedures)

An access procedure item (AP) contains all information needed to perform the actual bus accesses:

Table 9: AP (access procedure) data fields

Field name	Description
`char *setup`	setup call
`char *access`	access call
`char *conv`	conversion call
`char *test`	data check/test call
`int flag`	essential
`int s`	setup time
`int d`	delay time
`int a`	access time
`int setup_val`	setup value
`S_Ptr channel`	pointer to the channel in the symbol table
`S_Ptr mask`	pointer to the mask in the symbol table
`AP_Ptr next`	pointer to the next AP, if several APs are to execute in a sequence

The entry `mask` is only used for binary data and points to the pertaining mask, `next` is used for channels that need more than one access procedure per signal, e.g., all binary modules with several masks.

The AP elements are accessible through lists for each signal class (APLs).

Table 10: APL (access procedure list) list element

Field name	Description
`AP_Ptr p`	pointer to an AP
`APL_Ptr next`	pointer to the next list item

The first element in the list remains empty at all times; new elements are attached at the end of the list, pointed to by `APL_Last[]`. There are four lists:

Table 11: APL (access procedure list) lists

Variable	Description
`APL_Base[HI]`	APs of frequency class `HI`
`APL_Base[MED]`	APs of frequency class `MED`
`APL_Base[LOW]`	APs of frequency class `LOW`
`APL_Base[OUTPUT]`	`OUTPUT` type APs

A.4.3 IPs (Initialization Procedures)

IPs contain initialization functions that are stored in a linked list. An IP element consists of four fields:

Table 12: IP (initialization procedure) list element

Field name	Description
`char *routine`	initialization function call
`int value`	initialization value
`S_Ptr module`	pointer to module (used for 'private' initializations)
`struct IP *next`	next IP

The elements are stored in the same sequence they are encountered, i.e., the first initialization function read from the specification file will be executed first.

A.4.4 Variables

Freq. Frequencies are given in Hertz. Freq is of type `int Freq[3]`.

Table 13: Sampling frequencies

Variable	Description
`Freq[HI]`	Sampling frequency for type `HI` signals
`Freq[MED]`	Sampling frequency for type `MED` signals
`Freq[LOW]`	Sampling frequency for type `LOW` signals

Indices and Vector dimensions. A collection of variables containing the actual indices for the next element of various vectors. Upon completion of the syntactic analysis, these values correspond with the respective vector sizes for use in the generated files.

```
max_i_hist_cnt,/* I_Hist[] index count max value                */
max_hi_cnt,    /* BC_S_Data.value[] (HI) index count max value  */
max_i_cnt,     /* BC_R_Data.value[] (MED/LOW) index max value   */
max_o_bin_cnt, /* O_Bin[] index count max value                 */
max_o_cv_cnt;  /* O_ConvSend[], O_Acc[] index count max value   */

max_hw_cnt,    /* I_HW_Vec[] index count max value              */
max_ml_cnt;    /* .ML_x[] index item count max value            */
```

S_Actual. `S_Actual` contains the actual (i.e., the one that was accessed last) symbol in the symbol table.

```
S_Actual,      /* last (actual) accessed symbol                 */
S_actCarrier,  /* current carrier record                        */
S_actModule,   /* current module record                         */
S_actMask,     /* current mask                                  */
S_actPattern;  /* current pattern                               */
```

A.4.5 Access Distribution

Access distribution is performed in two stages. First, all signals that must be read in the i-th sampling period are selected and stored in the variable `slice`. Then the required accesses are combined with permutations and the best combination is chosen.

Selection of the Signals for One Period. A number of signals from frequency classes `MED` and `LOW` must be selected in each sample period that are to be read alongside the signals of class `HI`. The selection is crucial in order to achieve an even distribution of all accesses, i.e., if there are 10 such signals and 10 periods, one is read in each period (as opposed to reading all ten in a single period). The algorithm can be outlined as follows.

The total number of signals $Total[class]$ as well as the required sampling frequencies $Freq[class]$ are known from parsing. The signals of class `HI` are read in every period, `MED` signals only every $m = Freq[HI]/Freq[MED]$ periods, and the signals of class `LOW` are read only every $l = Freq[HI]/Freq[LOW]$ periods. The objective is to obtain an even distribution of all accesses through sampling periods 0 to $l-1$. The total number of signals that must be accessed per period amounts to $elem[class] = Total[class] \cdot Freq[class]/Freq[HI]$ (a remainder is possible).

In the hardware-oriented configuration input file, the signals are ordered according to their location on carrier cards and module slots. Because exclusion relations between signals on the same slot, i.e., the same mezzanine card are highly probable, signals are not inserted into the schedule in the same sequence as they are encountered in the configuration. By following this strategy most situations where two signals belonging to the same hardware item end up in the same sample period can be avoided from the start, thus reducing the number of conflicts that must be resolved later.

A signal of type $class$ is read in every $Freq[HI]/Freq[class]\,th$ period. Special care must be taken to accommodate the possibly remaining signals. The array $counter[class]$ holds values between 0 and $(Freq[HI]/Freq[class]) - 1$. An additional signal is copied, if $rest[class] > counter[class]$. This technique has the disadvantage of accumulating a remaining signal of each class `MED` and `LOW` in the last scheduler period, which is detrimental to the distribution algorithm (an optimized permutation, see *Access Distribution*, below). The efficiency of this algorithm may depend on the presence or absence of a single additional signal. The problem is solved in the generator by explicitly altering the contents of $counter[class]$ in such a way that the remaining signals of class `LOW` are inserted into those slices where signals of class `MED` will be inserted with a very low probability only.

Variables and Procedures. The signals that must be dealt with in a specific period are contained in variable `Slice`.

```
APL_Ptr Slice;
```

`Slice` is a list of pointers to APs. The first element in a slice is always empty. A number of procedures facilitate the administration and the use of slices:

```
Slice_Init()
```

must be called to initialize a slice and all connected structures.

```
Slice_New()
```

generates the next slice.

```
Slice_next_AP()
```

searches for the next free element in a slice. Elements whose slots have been allocated to another signal are skipped. One of three search methods is chosen with a parameter:

- search next element, starting at the actual position,
- search next element, starting at the beginning,
- search next element from the actual position to the end of the list (no reset at the end).

Access Distribution. The best distribution of all accesses can be found by searching through all permutations. Unfortunately, there exists no feasible inductive way to achieve this.

A reasonable distribution can be accomplished in $O(n^2)$ with a simple heuristics. First, the dependencies between different accesses are represented in a directed acyclic graph. The only dependencies to account for are possible access clashes to the same module within a single slice. In order to catch these conflicts, it is therefore sufficient to create a list of accesses for each module. First, the variable `w.start` that indicates the earliest possible start time is initialized to `0` for all the first elements in a slice, along with the slice timer `time`. Then the `PathCycles` for each element in a serial schedule are calculated, i.e., the time (in ms) remaining until the end of the schedule is reached. The schedule itself is then obtained with the following algorithm:

```
1.  time= 0
2.  WHILE (not all elements inserted)
2.1   IF (there exist elements with w.start <= time)
2.2     THEN take element w with highest number of PathCycles
2.3     ELSE take element w with lowest start time
2.4   insert element w into the global schedule
2.5   w.start= MAX(w.start, time)
2.6   time= w.start + w.wait
2.7   w.next.start= w.start + w.wait + w.delay
```

The heuristics is to be found in lines `2.1` to `2.3`. A signal `w` is chosen (`2.2`) whose first access is executable immediately (`start <= time`) and which needs the most accesses to complete, i.e., the one that will cause the greatest number of gaps in the form of delay time between individual accesses. The earlier it is inserted, the easier it will be to fill the gaps. If there is no such signal left (`2.3`), then the signal that produces the smallest gap is chosen. Because its next access cannot be scheduled directly, a delay instruction of duration `w.start-time` must be inserted to satisfy `2.5`. In `2.6`, the slice time is updated to after the execution of `w` has completed, and `2.7` calculates the start time for the next instruction.

B Syntax

B.1 Architecture

```
SCANNER pvt;

CHARACTERS
eofC = {0X}.
blankC = {020X} | {09X} | {0DX}.
any = {0X..0FFX} - eofC.
letter = {"A".."Z","a".."z"}.
digit = {"0".."9"}.
binary = {"0","1"}.
hex = digit + {"a".."f","A".."F"}.

TOKENS
eof = eofC.
void = "VOID".
ident = (letter | "_") {letter | digit | "_"}.
cardinal = digit {digit}.
number = (["-"] digit {digit} ["." {digit}]) |
   ("0X" | "0x") hex {hex} |
   ("0B" | "0b") binary {binary}.
string = "" {any} "".
stdtype = "void" | [signed | unsigned]
   "char" | "short" | "int" | "long" | "float" | "double").

COMMENT FROM "/*" TO "*/" UNNESTED

LITERALS
"VERSION", "CONSTANT", "TYPE", "INIT",
"COMPONENT", "PORT", "IN", "OUT", "MESSAGE", "COMPUTATION",
"CONNECTOR", "ROLE", "SOURCE", "SINK", "GLUE",
"INSTANCE", "ATTACHMENT",
",", ";", "(", ")", "[", "]", "&", "|", "=", "!"

END pvt.


GRAMMAR pvt;

SKIP blankC;

Pivot = Version {Constant | Type | Component | Connector} {Instance}.

Version = "VERSION" string.
```

```
Constant = "CONSTANT" ident number [string].

Type = "TYPE" (Scalartype | Recordtype | Arraytype).

Scalartype = (ident | stdtype) ident [string].

Recordtype = ("struct" | "union") [ident] [string]
   "{" {([("struct" | "union")] ident | stdtype)
   ident {"," ident} ";"} "}" ident.

Arraytype = (ident | stdtype) ident
   "[" (number | ident) {"," (number | ident)} "]" [string].

Init = "INIT" procedure {procedure}.

Component = "COMPONENT" ident [string] {Port} Computation.

Port = "PORT" ("IN" | "OUT") ident [string] {Message}.

Message = "MESSAGE" ident ((ident ident "(" ident {"," ident} ")")
   | void) [string].

Computation = "COMPUTATION" ident [string].

Connector = "CONNECTOR" ident [string] {Role} Glue.

Role = "ROLE" ident string.

Glue = "GLUE" string.

Instance = "INSTANCE" ident ":" ident.

Attachment = "ATTACHMENT" ident "." ident "AS" ident "." ident.

END pvt.
```

B.2 Input and Output

```
SCANNER acc;

CHARACTERS
eofC = {0X}.
blankC = {020X} | {09X} | {0DX}.
any = {0X..0FFX} - eofC.
letter = {"A".."Z","a".."z"}.
digit = {"0".."9"}.
binary = {"0","1"}.
hex = digit + {"a".."f","A".."F"}.

TOKENS
eof = eofC.
void = "VOID".
ident = (letter | "_") {letter | digit | "_"}.
cardinal = digit {digit}.
number = ["-"] digit {digit} ["." {digit}]) |
   ("0X" | "0x") hex {hex} |
   ("0B" | "0b") binary {binary}.
string = "" {any} "".
wildcard = ".b" | ".s" | ".c" | ".n" | ".hd" | ".sd" | ".l" | ".t" | ".m".
procedure = ident "(" [(ident | wildcard) {"," (ident | wildcard)}] ")".

COMMENT FROM "/*" TO "*/" UNNESTED
```

```
LITERALS
"VERSION", "SAMPLING", "INCLUDE", "DEFINE", "INIT", "CARRIER", "MODULE",
"HI", "MED", "LOW", "OUTPUT", "PATTERN",
",", ";", "(", ")", "[", "]", "&", "|", "=", "!"

END acc.


GRAMMAR acc;

SKIP blankC;

IO_Config = Version Sampling {Include | Define | Init} {Carrier}.

Version = "VERSION" string.

Sampling = "SAMPLING" cardinal cardinal cardinal.

Include = "INCLUDE" string [string].

Define = "DEFINE" ident number [string].

Init = "INIT" procedure {procedure}.

Carrier = "CARRIER" ident number [string] {Module}.

Module = "MODULE" ident cardinal [string]
   {(Input {Pattern}) | (Output {Pattern)}.

Input = ("HI" | "MED" | "LOW") ident cardinal number
   ((procedure "," cardinal "," cardinal) | void)
   ((procedure "," cardinal) | void)
   (procedure | void)
   (procedure | void).

Output = "OUTPUT" ident cardinal number
    (procedure | void)
    ((procedure "," cardinal "," cardinal) | void)
    ((procedure "," cardinal) | void).

Pattern = "PATTERN" ident number cardinal {cardinal}
    ident number {ident number}.

END acc.
```

B.3 Event

```
SCANNER evt;

CHARACTERS
eofC = {0X}.
blankC = {020X} | {09X} | {0DX}.
any = {0X..0FFX} - eofC.
letter = {"A".."Z","a".."z"}.
digit = {"0".."9"}.
binary = {"0","1"}.
hex = digit + {"a".."f","A".."F"}.

TOKENS
eof = eofC.
void = "VOID".
```

```
ident = (letter | "_") {letter | digit | "_"}.
cardinal = digit {digit}.
number = ["-"] digit {digit} ["." {digit}] |
   ("0X" | "0x") hex {hex} |
   ("0B" | "0b") binary {binary}.
string = "" {any} "".

COMMENT FROM "/*" TO "*/" UNNESTED

LITERALS
"VERSION", "NODE", "TTICK", "DISPLAY", "SLAVE", "RSLAVE",
"BEVENT", "CEVENT", ",", ";", "(", ")", "[", "]", "&", "|", "=", "!"

END evt.


GRAMMAR evt;

SKIP blankC;

Evt_Config = Version Node Timer [Display] [Slave] {BEvent | CEvent}.

Version = "VERSION" string.

Node = "NODE" ident.

Timer = "TTICK" cardinal.

Display "DISPLAY".

Slave = "SLAVE" | "RSLAVE".

BEvent = "BEVENT" ident ident (ident ident {ident ident}).

CEvent = "CEVENT" ident ident
   (ident number ident number {ident number ident number}).

END evt.
```

C Glossary

Accuracy. The systematic (repeatable) *error* in a measurement (→ *precision*).

Accuracy interval. The maximum permitted time interval between the *point of observation* of a *controlled process* and the point of use of the corresponding *measured data.*

Action. Any activity aimed at bringing about an effect. Specifically, an action is the output of a *CIP transition* that eventually affects the *controlled processs* in the *environment.*

Active task. Refers to the *task* that is currently running. *Multi-tasking operating systems* need to know the identity of the currently running *task* in order to save its *context* in the right place when it is suspended.

Actuator. A *transducer* that accepts data and *trigger* information from an *interface node* and realizes an intended effect in the *controlled process.*

Address bus. A set of electrical lines connected to the processor and all of the peripherals with which it communicates. The address bus is used by the processor to select a specific memory location or register within a particular peripheral. If the address bus contains n electrical lines, the processor can uniquely address up to 2^n such locations (→ *data bus*).

Alarm shower. A correlated set of alarms that is caused by a single *primary event.*

Aliasing. The process where a sinusoid is perceived at a frequency different from its original frequency as a result of *sampling* or another nonlinear action (→ *anti-alias filter*).

Anti-alias filter. Low-pass filter placed before an analog-to-digital converter. The filter removes frequencies above $1/2$ the *sampling* rate that would alias during conversion (→ *aliasing*).

Aperiodic task. A *task* where neither the *task request times* are known nor the minimum time interval between successive requests for execution (→ *periodic task, sporadic task*).

API. Application program interface. The data and control *interface* between an application program and the *operating system.*

Application software. All the software modules specific to a particular embedded project.

Application specific fault tolerance. Fault tolerance mechanisms that are introduced within the application code (→ *systematic fault tolerance*).

A priori knowledge. Knowledge about the future *behavior* of a *system* that is available ahead of time.

Architecture. The description of a *system's* constituents and their interrelations. Either the abstract description of functional components and their interconnection (operational architecture), or the *hardware* components used to build a *system,* and their interconnections (implementation architecture).

ASIC. Application specific integrated circuit. A special-purpose circuit that is used to implement certain parts of a *system's* functionality directly, without any underlying general-purpose computation engine.

Assembler. A software development tool that translates human-readable *assembly language* programs into machine language instructions that the processor can understand and execute.

Assembly language. A low-level human-readable form of a processor's instruction set.

Assertion. A predicate or a set of conditions that must be satisfied at a particular point during program execution.

Assumption coverage. The probability that assumptions made in the model building process hold in reality. The assumption coverage limits the probability that conclusions derived from a model will be valid in the real world.

ATM. Asynchronous transfer mode. An asynchronous *communication* technology for *communication* over broadband *networks* where the information is organized into constant length cells (48 data bytes, 5 header bytes).

Atomic data structure. A data structure that has to be interpreted as a whole.

Atomic operation. An operation that has the all-or-nothing property. It either completes and delivers the intended result or else does not have any effect.

Availability. A measure of the correct service delivery regarding the alternation of correct and incorrect service, measured by the fraction of time that the *system* is ready to provide the service.

Backbone network. A non real-time *communication network* for the exchange of non time-critical information between a *real-time computer system* and the data-processing *system*s of an organization.

Back-pressure flow control. In back-pressure flow control, the receiver of a sequence of *messages* exerts back pressure on the sender so that the sender will not outpace the receiver.

Bandwidth. The maximum number of bits that can be transmitted across a channel per time unit.

Behavior. A functional and temporal description of how a *system* is expected to execute.

Bit dominance. A communication protocol based on bit-wise arbitration. On bus idle, all competing nodes start to transmit their unique node identification value simultaneously, while continuously monitoring the signals present on the bus. *Contention* is resolved by the transmission medium where low signals override high signals (or vice versa). If a node detects a dominant signal opposite to its own, it drops out of the competition. Throughput and efficiency are high (because the arbitration is part of the message), the protocol is robust (because no specific transmission order is required). Fairness can be a problem under heavily loaded conditions. This protocol is used in *CAN.*

Bit-length of a channel. The number of bits that can traverse the channel within one *propagation delay.*

Blocking. When the execution of a *task* is suspended because of another *task.* Causes are manifold: it may attempt to acquire a *semaphore* that is unavailable, or when it waits for a *message* that is not in its queue, or while it waits for a pause to expire, or when, in a time-slicing *system,* its time-slice has expired, to give a few examples.

BNF. Backus-Naur form. A meta language to describe syntactic properties of computer languages [Wir77]. Repetition is denoted by {} (i.e., {a} stands for ε | a | aa | aaa | …), optionality by [] (i.e., [a] stands for ε | a). Terminal symbols are enclosed in "" (a quote mark as a literal is written twice).

Broadcast. A *communication* scheme where all *nodes* aside from the sender are receivers.

Busy waiting. → *polling.*

Byzantine error. Occurs if a set of receivers receive different (conflicting) values about a *controlled process* at some point in time. Some or all of these values are incorrect (synonym. malicious error).

C. Common programming language used in science end engineering [KeR88, Joy96].

CAN. Controller area network. A low-cost *event-triggered real-time communication network* that is based on the *bit dominance* protocol. The serial bus has multi-master capabilities and handles simultaneous requests through non-destructive bit-wise arbitration, guaranteeing low latency to the highest priority message.

Causal order. A causal order among a set of *events* is an order that reflects the cause effect relationships between the *events.*

Channel access delay. The time between the request for sending a message and the start of its transmission.

CIP. Communicating interacting processes. A method and corresponding graphic tool for modeling the functional part of an *embedded system,* based on extended *FSMs* (→ *system, cluster, process*).

Clock. A device for time measurement that contains a counter and a physical oscillation mechanism that periodically generates an *event* to increase the counter.

Cluster. A cluster embraces a number of synchronous processes as part of a *CIP system.* Clusters are asynchronous among themselves.

CNI. Communication network interface. The *interface* between the *communication controller* and the *host* within a *node* of a *distributed computer system.*

Communication. The activity of receiving and transmitting information. In a *CIP* system, communication is asynchronous across the boundaries of a *node* using *messages* (→ *interaction*).

Communication controller. One or several communication controllers control the *communication* of a *node* within a *distributed computer system.*

Compiler. A software development tool that translates high-level language programs into the machine-language instructions that a particular processor can understand and execute.

Composability. An *architecture* is composable regarding a specified property if the *system* integration will not invalidate this property, provided it has been established at the *subsystem* level.

Computational resource. A time-shared, preemptable component, or group of components, in the *hardware architecture.*

Congestion. If the demand for *network resources* exceeds the available capacity over some period of time, a *network* is said to be congested. Congestion leads to high *jitter* and packet loss, which can also be thought of as infinite delay variation.

Connection configuration. A high level description of a *real-time computer system's* connective elements, i.e., all the aspects concerned with its *embedding* and not related to its *functional model.*

Connection Oriented Protocol. A protocol with only two nodes per physical transmission medium, typically connected with serial lines. Its *behavior* is deterministic between directly connected nodes, whereas latency can be high for indirect connections.

Construction. The bottom-up activity of actually building a *system* in accordance with a *design*.

Contact bounce. The random oscillation of a mechanical contact immediately after closing.

Contention. Competition for resources. Used in *networks* to describe the situation where two or more nodes attempt to transmit a message at the same time (→ *CSMA*, *controlled access*).

Context. The current *state* of the processor's registers and flags. In a more complete sense context also includes the *task's* local variables and its subroutine nesting information.

Context switch. The process of suspending one *task* in favor of another one in a *multi-tasking system.* A context switch involves saving the *context* of the running *task* and restoring the previously-saved *context* of the other. The code that does this is necessarily processor-specific.

Continuous signal. A signal that is formed from continuous variables, such as a voltage varying with time (synonym. analog signal, → *discrete signal*).

Control. The purposeful influencing of a *controlled process's* signals in combination with information processing. In closed-loop control, an object's controlled variable is continuously monitored and its input signals adjusted, so that it corresponds to a reference variable. In contrast, open loop control has no feedback; an object's output signals are merely influenced according to the inner laws of this object.

Controlled access. A technique that avoids *contention* by deliberately limiting access to resources (→ *TDMA*).

Controlled process. The object in the *environment* that is to be controlled by the *real-time computer system,* such as an industrial plant, or a device.

Control system. Any contrivance that performs *control* functions within an *environment* with which it is usually tightly integrated. With the advance of digital computers, *embedded system* is increasingly becoming a synonym for control system.

Cooperative scheduling. Each in a group of cooperative *scheduling tasks* voluntarily *relinquishes* control so that other *tasks* can run. If control is not given up deliberately other *tasks* will be starved. Control can be *relinquished* explicitly by a *yield,* pause, or suspend, or implicitly by waiting for an *event* (→ *priority based preemptive scheduling, time-sliced scheduling*).

CORBA. Common object request broker architecture. Enables pieces of programs, called objects, to communicate with one another regardless of what programming language they were written in or what *operating system* they are running on [OMG95].

CPU. Central processing unit. The part of a processor that executes instructions.

CRC. Cyclic redundancy check field. An extra field in a *message* for the purpose of detection of value *errors.*

Critical section. A block of code that must be executed in sequence and without interruption to guarantee correct operation of the software.

Cross-compiler. A *compiler* that runs on a different platform than the one for which it produces object code. A cross-compiler runs on a host computer and produces object code for the *target.*

CSMA. Carrier sense multiple access protocol. A set of rules determining how *network* devices respond when two devices attempt to use a physical medium simultaneously. Collisions are detected and resolved with exponential back off (CD, *Ethernet*), or avoided by generating *contention* slots following a collision (CA). The protocol's key advantage is its ability to accommodate an unlimited number of nodes without requiring network initialization and configuration. However, under heavy traffic the overhead is unbounded due to repeated collisions, lowering determinacy and efficiency. CSMA is a type of *contention protocol.*

CSP. Communicating sequential processes. A formal method where systems are described as a number of components (processes) that operate independently and communicate with each other over well-defined channels. This basic (synchronous) communication model is one of the core ideas underlying the occam programming language.

Cyclical scheduling. Accomplished by simply calling one *task* after another in an infinite loop. The advantages are that it is small and easy to understand and control, that it does not require a complicated *operating system* and only one stack. The disadvantages are missing priorities, that *tasks* must retain their *states* between runs, and the possibility of excessive loop latency when there are many *tasks* (synonym. *round robin scheduling*).

Data bus. A set of electrical lines connected to the processor and all of the peripherals with which it communicates. When the processor wants to read/write the contents of a memory location or register within a particular peripheral, it sets the *address bus* pins appropriately and receives/transmits the contents on the data bus.

Data encoding technique. Defines the way in which the logical bits are translated into physical signals on the transmission medium. Usually, through a redundant format, measures are taken to ease *clock* extraction and to remove DC components.

Deadline. The point in time when a result should/must be produced (→ *soft deadline, hard deadline*).

Deadline interval. The interval between the *task request time* and the *deadline.*

Deadlock. An unwanted software situation in which an entire set of *tasks* is blocked, waiting for an *event* that only a *task* within the same set can cause. If a deadlock occurs, the only solution is to reset the *hardware.* It is the objective of sound software *design* practices to avoid deadlocks altogether.

Debugger. A software development tool used to test and debug embedded software. The debugger runs on a host computer and connects to the *target* through a serial port or *network* connection. Using a debugger, software can be *downloaded* to the *target* for immediate execution, breakpoints can be set and the contents of specific memory locations and registers examined.

Decimation. Reducing the *sampling* rate of a digitized signal. Generally involves *low-pass filtering* followed by discarding samples (→ *interpolation*).

Design. The top-down activity of laying out a *system.* It involves component and connector definitions and their arrangement within an *architecture.* The design precedes the *construction*.

Device driver. A software module that hides the details of a particular peripheral and provides a high-level programming *interface* to it.

Difference equation. Equation relating the past and present samples of the output signal with past and present samples of the input signal. In continuous systems, the equivalent of difference equations are differential equations.

Discrete signal. A signal that uses quantized variables, such as physical entities with discrete states, or any value residing in a digital computer (→ *continuous signal).*

Dispatch. The process of granting the most eligible contending entity access to a resource (processor, bus, communication path etc.) at a particular point in time. Eligibility is manifest either by the entity's position in a *schedule,* by its *priority* or *deadline.* In time-critical applications dynamic *hardware* dispatchers are used.

Distributed computer system. A *computer system* that consists of more than one computational *subsystem.*

DMA. Direct memory access. A technique for transferring data directly between two peripherals (usually memory and an I/O device) with only minimal intervention by the processor. DMA transfers are managed by a third peripheral called a DMA controller.

Download. The transfer of executable code from a *host* to a *target.* The *target* must have resident software that can read the incoming data, translate it if necessary and load and run the code.

DSP. Digital signal processor. A device that is similar to a microprocessor, except that the internal *CPU* has been optimized for use in applications involving discrete-time signal processing. In addition to standard microprocessor instructions, DSPs usually support a set of complex instructions to perform common signal-processing computations quickly.

Duration. A section of the timeline.

Dynamic priority. Priorities that can be changed at run time. (→ *fixed priority*).

Dynamic scheduler. An algorithm that decides at run time after the occurrence of a significant *event,* which *task* is to be executed next (→ *scheduling*).

EDF. Earliest deadline first algorithm. An optimal dynamic preemptive algorithm for *scheduling* a set of independent periodic *tasks.* At every time step the task with the closest deadline is scheduled (→ *preemption*).

Embedded system. A *real-time computer system* that is embedded in a well specified larger system, consisting in addition to the embedded computer system of a physical *subsystem* and, often, a man-machine *interface.*

End-to-end protocol. A *protocol* between the users residing at the end points of a *communication* channel. At the interface of a *real-time computer system* and its *controlled process,* the effect caused by an output *action* via an *actuator* must be observed by an independent *sensor.*

Environment. The environment of an *embedded system* is the collection of its *controlled processs.*

Error. An error is that part of the *state* of a system that – caused by a *fault* – deviates from the specification.

Error masking. A mechanism that prevents an *error* from causing a *failure* at a higher level by making immediate use of the provided redundancy (e.g., error correcting codes).

ESTEREL. A synchronous specification language for the class of deterministic reactive systems that wait for a set of possibly simultaneous inputs, react to the inputs by computing and producing outputs, and then wait again. It allows only deterministic behaviors to be specified. Internal communication is compiled away, and a single deterministic finite state machine generated. Thus, the parallelism in Esterel is a means of structuring, and does not incur any run-time overhead. Furthermore, the maximum amount of time taken by any reaction is accurately bounded.

Ethernet. A local-area *network* (*LAN*) that uses a bus or star *topology* and is based on the *CSMA* access method to handle simultaneous demands.

Event. A happening at an *instant.* In a functional *CIP* model a *transition* chain is triggered by an event. Any change of *state* in the *environment* is communicated to the model as an *event.*

Event-triggered system. A *real-time computer system* is event-triggered if all *communication* and processing activities are triggered by an *event* other than a *clock tick* (→ *time-triggered system*).

Exception. Run-time errors in an *real-time computer system* that escape static scrutiny must raise an exception instead of crashing the machine. Exceptions are implemented by a software *interrupt* generated by the *CPU.*

Exclusion relation. A constraint that forbids the simultaneous invocation of a pair of *tasks,* usually to prevent *resource contention* (→ *scheduling*).

Execution time. The time it takes to execute a piece of code. The worst case execution time is called *WCET.*

Explicit flow control. In explicit flow control the receiver sends an explicit acknowledgment *message* to the sender, informing the sender that the previously sent *message* has correctly arrived and that the receiver is now ready to accept the next *message* (→ *flow control, implicit flow control, PAR).*

Explicit synchronization. The dynamic *synchronization* of *tasks* by *synchronization* statements, such as 'WAIT-FOR-EVENT' (→ *implicit synchronization*).

Fail-operational system. A *real-time computer system* where a safe *state* cannot be reached immediately after the occurrence of a *failure.* An example of a fail-operational system is a flight-control system without mechanical or hydraulic back-up onboard an airplane.

Fail-safe system. A *real-time computer system* where a safe *state* can be identified and quickly reached after the occurrence of a *failure.*

Fail-silence. A *subsystem* is fail-silent if it either produces correct results or no results at all, i.e., it is quiet in case it cannot deliver the correct service.

Failure. An *event* that denotes a deviation of the actual service from the specified or intended service.

Fairness. A concept, which dictates that if a *task* runs too long, its *priority* is lowered so that other *tasks* can get their fair share of the *CPU.* This concept is repugnant to real-time principles and is used chiefly in multi-user systems with the intent of disallowing one user to monopolize the *CPU.*

Fault. A fault is the cause of an *error,* e.g., the loss of function of a *hardware* device.

Fault hypothesis. The fault hypothesis identifies the assumptions that relate to the type and frequency of *fault*s that the computer system is supposed to handle.

Fault tolerance. The ability of a *system* to survive *failures* (→ *application specific fault tolerance, systematic fault tolerance*).

Field bus. A low cost bus for the interconnection of *sensors* and *actuators* in the *controlled process* to a *node* of a *distributed computer system.*

FIT. A unit for expressing the *failure* rate. 1 FIT is 1 failure/10^{-9} hours.

Fixed priority. Set once and remaining unalterable at run time (→ *dynamic priority*).

Flow control. Flow control assures that the speed of the information flow between a sender and a receiver is such that the receiver can keep up with the sender (→ *explicit flow control, implicit flow control).*

Forbidden region. A time interval during which it is not allowed to schedule a *task* that may conflict with another critical *task* (→ *exclusion relation*).

Fragmentation. In a system that allows for memory allocation and deallocation, the available free memory can eventually become fragmented, i.e., non-contiguous, if memory is allocated from one large pool.

FSM. Finite state machine. An abstract machine with a single point of control. It consists of a set of *states,* a set of input *events,* a set of output *actions* and a state *transition* function. The function takes the current *state* and an input *event* and returns the new set of *actions* and the next *state.* The state machine can also be viewed as a function that maps an ordered sequence of input *events* into a corresponding sequence of (sets of) output *actions.* Output generation is bound to transitions in Mealy machines (pulse output), to states in Moore machines (level output). Of functionally equivalent machines, Mealy machines are smaller and more adapted to behavioral modeling. Extensions to finite state machines include *transition operations* to handle data, and switch functions to resolve non-determinism.

Functional model. A high-level description of a system's inner *states* and *behavior* that abstracts from its connection to a real *environment* (→ *CIP, connection configuration*).

Ground state. The ground *state* of a *node* of a *distributed computer system* at a given level of abstraction is a *state* where no *task* is active and where all *communication* channels are flushed, i.e., there are no *messages* in transit.

Guaranteed timeliness. A *real-time computer system* is a guaranteed timeliness system if it is possible to reason about the adequacy of the *design* without reference to probabilistic arguments, provided the assumptions about the *load hypothesis* and the *fault hypothesis* hold.

Hard deadline. A *deadline* for a result is hard if the result becomes useless, or if a catastrophe can occur in case the *deadline* is missed.

Hard real-time. Parts of a *real-time computer system* that must meet *hard deadlines* (→ *safety-critical real-time computer system, soft real-time*).

Hardware. The physical part of a computer system, i.e., processors, *ASICs,* memories, caches, peripheral *interfaces* and their interconnections.

Heterogeneous system. A computer system that contains processing elements of several kinds, e.g., different types of processors or a mixture of processors and *ASICs.*

Host. A workstation where the development environment runs, including high level tools, code generators, compilers, linkers and *debuggers.* Finished application code is *downloaded* to a *target.*

Idle task. A *task* owned by the *operating system* that is created and scheduled during system initialization. The idle task is typically a 'do nothing' tight loop. Its only purpose is to run when no other *tasks* run. The key is that the idle task has a lower *priority* than any other *task,* and even though it is scheduled to run and occupies a place in the *ready queue,* it will not run until all other *tasks* are idle. And, conversely, when any other *task* is scheduled, the idle task is preempted.

Implicit flow control. In implicit flow control, the points in time when *messages* will be sent are agreed upon by the sender and receiver based on *a priori knowledge,* i.e., at system start up. The sender commits itself to send only *messages* at the agreed points in time, and the receiver commits itself to accept all *messages* sent by the sender, as long as the sender fulfills its obligation (→ *explicit flow control, flow control).*

Implicit synchronization. The static *synchronization* of *tasks* by a priori *temporal control* of the *task* activation (→ *explicit synchronization*).

Inquiry. An synchronous asymmetric transaction where the transitional *behavior* of a *CIP process* depends on the *state* of some other *process* without an established master-slave relationship.

Instant. An instant is a cut of the timeline.

Instantiation. To start single or multiple instances of a *task.*

Interaction. The mutual influencing of processes on the same synchronous entity using *pulses* (→ *communication*).

Interbus-S. A *field bus* based on the notion of a distributed shift register. All inputs and outputs are dealt with in every cycle and the time of a single cycle is deterministic.

Instrumentation interface. The interface between the *real-time computer system* and the *controlled process.*

Intelligent sensor. An intelligent sensor consists of a *sensor* and a processing unit such that *measured data* is produced at the output *interface.* Intelligent *sensors* can be fault tolerant.

Interface. A common boundary between a system and its *environment* or between two *subsystems.*

Interface node. A *node* with an instrumentation *interface* to the *controlled process.*

Interpolation. Increasing the *sampling* rate of a digitized signal. Generally achieved by placing zeros between the original samples and using a *low-pass filter* (→ *decimation*).

Interrupt. Most *CPU's* have a pin designated as the 'interrupt' pin. When it is asserted the *CPU* halts, saves the current *context* on the stack, and jumps to the location of an *interrupt service routine.*

Interrupt latency. The time it takes for an *interrupt service routine* to be entered, once the processor interrupt pin is asserted.

Interrupt service routine. A routine that is invoked when an *interrupt* occurs.

Irrevocable action. An *action* that cannot be undone, e.g., activation of the firing mechanism of a firearm.

Jitter. The variation in an isochronous sequence, e.g., the difference between the maximum and the minimum *duration* of a function (*sampling,* processing, *communication*), or the variation of the point in time when a periodic *event* occurs.

JSD. Jackson System Development. A software development method based on a real-world oriented paradigm. It treats dynamic information problems by means of concurrent sequential processes, simulating a part of the real world and producing requested information about it. Its modeling framework relies on concurrent *processes* described by extended regular expressions (structograms).

LAN. Local area network. A computer *network* for fast data transmission, but with limits on the number of *nodes* and the extent of the *network.* Each individual computer in a LAN can share devices and data, as well as communicate with others (→ *WAN).*

Laxity. The laxity of a *task* is the difference between the *deadline interval* minus the *execution time* (the *WCET)* of the *task.*

Life sign. A periodic signal generated by a computer. The life sign is monitored by a *watchdog.*

LL. Least laxity algorithm. An optimal dynamic preemptive algorithm for *scheduling* a set of independent periodic *tasks* (→ *preemption*).

Load hypothesis. The load hypothesis specifies the *peak load* that the computer system is supposed to handle.

Logical control. Logical control is concerned with the control flow within a *task.* Logical control is determined by the given program structure and the particular input data to achieve the desired data transformation (→ *temporal control).*

Low-pass filter. A filter, either analog or digital, which passes all frequencies below a specified value and attenuates all frequencies above that value.

Maintainability. The probability *M(d)* that the system is restored within a time interval *d* after a *failure.*

Master process. A *CIP process* that influences the *behavior* of its *slave process(es)* by the *state* it is itself in (→ *mode*).

Measured data. A *raw data* element that has been preprocessed and converted to standard technical units. A *sensor* that delivers measured data is called an *intelligent sensor* (→ *raw data*).

Media access protocol. A *protocol* that defines the method used to assign the single *communication* channel to one of the *nodes* requesting the right to transmit a *message.*

Membership service. A service in a *distributed computer system* that generates consistent information about the operational *state* (operating or failed) of all *nodes* at agreed points in time (membership points). The length and the *jitter* of the interval between a membership point and the moment when the consistent membership information is available at the other *nodes* are quality of service parameters of the membership service.

Message. The object of asynchronous *communication* activities. In a *CIP system,* all input and output activities are messages being sent or received.

Microtick. A microtick of a physical *clock* is a periodic *event* generated by this *clock* (→ *tick*).

Minimum time between events. The minimal interval (mint) between two *events* of the same type.

MMI. Man-machine interface. The *interface* between the *real-time computer system* and the operator.

MTTF. Mean time to failure. A quality measure of devices and systems obtained by integrating *reliability* over time.

Mode. One of several *behavioral* variants of a *CIP slave process,* of which only one is active at a time and which is selected according to the actual *state* of a *master process.* All of its modes together define the transition structure of a process.

Model checking. Automatic verification of formally defined systems. Because of *state space explosion,* model checking is not universally applicable.

Multicast. A *communication* scheme with one sender and multiple receivers.

Multi-processing. Describes a *hardware architecture* in which several *CPU's* are used to distribute the load of a single application.

Multi-tasking. A single computer's time is shared among several *task.*

Mutual exclusion. The guarantee that a *task* will be given sole access to a *resource* without any outside interference.

NBW. Non-blocking write protocol. A *synchronization protocol* between a single writer and many readers that achieves data consistency without *blocking* the writer.

Network. A group of two or more computer systems or *nodes* linked together. Networks are classified according to their type (*LAN, WAN*), *topology* (bus, star, ring), *protocol* (*Ethernet*, *token ring*), *architecture* (peer-to-peer, client/server), media (twisted-pair wire, coaxial cables, fiber optic cables, wireless), and intent (general purpose, *real-time communication network*).

Node. A self-contained computer that performs a well-defined function within a *distributed computer system.* A node contains the system- and *application software* and a *communication controller.*

Offset. The offset between two *events* denotes the time difference between these *events.*

Operating system. Software at the heart of many computer systems that manages its resources and provides *interfaces* between *application software* and *hardware*. Operating systems are classified as multi-user (several users can run programs at the same time), *multi-processing* (a program may run on more than one *CPU*), *multi-tasking* (more than one program can run concurrently), multithreading (different parts of a single program can run concurrently), and real-time (fulfills timing requirements plus some of the above). (→ *RTOS).*

Operation. A code snippet attached to a *CIP transition* that is executed when this *transition* is triggered by an *event* or *pulse*.

Operation mode. The various implementations that can be generated from a *real-time computer system's functional model* and pertaining *connection configuration.* These comprise partial implementations for test beds, full *target* implementations and simulation implementations.

Oversampling. → *decimation.*

PAR. Positive acknowledgement and retransmission. An event-triggered *explicit flow control protocol.* Each transaction is guarded with time-outs, and a number of retransmissions are attempted before it fails.

Partial correctness. A program is partially correct if it computes the desired result, assuming it terminates.

Peak load. The maximum amount of simultaneous propagated changes in the *environment* that a system must be able to contend with in a certain period.

Periodic task. A *task* that has a constant time interval between successive *task* request times (→ *aperiodic task*, *sporadic task*).

Petri Net. A formal method and modeling tool consisting of places, *transitions* and arcs. Places can contain tokens; the marking of the modeled system is given by the number of tokens in each place. *Transitions* model activities that can occur (the *transition* fires), thus changing the marking of the system. They are only allowed to fire if there are enough tokens available in the input places. When the *transition* fires, it removes tokens from its input places and adds some at all of its output places.

PLC. Programmable logic controller. A dedicated computer optimized for industrial *control* applications that is tightly integrated with peripheral devices. It is not universally programmable, instead an integral executive collects data periodically and evaluates them according to functional diagrams (written in relay ladder logic, for example).

Point of observation. The moment when a *controlled process* is observed.

Polling. In polling, *measured data* with regular updates is queried irregularly, i.e., the memory element is in the sphere of control of the computer system (→ *sampling*). As a communication protocol, polling refers to systems where a centrally assigned master periodically polls its slaves, giving them explicit permission to transmit. This protocol is used if peer-to-peer communication and global prioritization are not required, and when the overhead caused by the polling messages are acceptable. The master node presents a single point of failure.

Precedence relation. A constraint that prescribes the timely ordering of *tasks* (→ *scheduling*).

Precision. The *error* in a measurement that is not repeatable, since it depends on random *errors* (→ *accuracy*). In timing the precision of an ensemble of *clocks* denotes the maximum *offset* of respective *ticks* of any two *clocks* of the ensemble over the period of interest. The *precision* is expressed in the number of *ticks* of the reference *clock.*

Predictability. A property is predictable to the degree that it is known in advance. It ranges from determinism (the property is known exactly in advance) to maximum entropy (nothing at all is known in advance). Predictability of timeliness is the most fundamental property of a real-time system.

Preemption. When the currently executing *task* is stopped and execution is resumed at a different location of the program. If an *interrupt* occurs the currently executing *task* is preempted under *hardware* control by the associated *interrupt service routine.* Under software control it occurs when a dormant higher *priority task* becomes ready to run, i.e., when a *message* is sent to said *task* (→ *operating system*). It is a specific property of a *task* to be preemptable.

Primary event. A primary *event* is the cause of an *alarm shower.*

Priority. The order that determines when each *task* will run (→ *scheduling*).

Priority based preemptive scheduling. A *multi-tasking scheduling* policy by which a higher *priority task* will preempt a lower *priority task* when the higher *priority task* is ready to run. (→ *cooperative scheduling, time-sliced scheduling*).

Priority ceiling protocol. An algorithm for *scheduling* a set of mutually dependent periodic *tasks* that avoids *priority inversion.* The low priority *task* holding a shared resource is raised to a previously agreed high value (typically exceeding all the other priorities in the system.

Priority inheritance protocol. An algorithm for *scheduling* a set of mutually dependent periodic *tasks* that avoids *priority inversion.* The low priority *task* holding a shared resource is raised to the priority level of the high priority *task* requesting the *resource.*

Priority inversion. A situation where a high *priority task* is directly or indirectly blocked by a low *priority task* that has exclusive access to a *resource.* The situation can be avoided by using a server *task* instead of just using *tasks* with *exclusion relations.* The *priority ceiling protocol* or *priority inheritance protocol* present solutions to this problem.

Process. A component of a *CIP system* specified as an *FSM.* Processes within the same *cluster* are synchronous (cf. sequential programs that are schedulable separately are referred to as *tasks*).

Profibus. A multi-master *field bus* based on token passing. There are three variants: Profibus-FMS is used at upper levels of the communication hierarchy. Profibus-DP is a version with optimized performance in time-critical communication between automation systems and distributed peripherals. Profibus-PA is based on Profibus-DP and incorporates intrinsically safe transmission technology.

Propagation delay. The propagation delay of a *communication* channel denotes the time interval it takes for a single bit to traverse the channel.

Protocol. A set of rules and encoding specifications that govern the *communication* among partners.

Protocol latency. The time interval between the start of a transmission in the sending *node* and the delivery of this message at the receiver. Protocol latency includes the *propagation delay.*

Pulse. A synchronous *trigger* that propagates external *events* from one *process* to another within a *CIP cluster.*

Quantization. The *error* induced by the limited number of discrete steps used in the digital representation of *continuous signals* (→ *sampling*).

Raw data. An analog or digital data element as it is delivered by an unintelligent *sensor* (→ *measured data*).

Ready queue. A list of *tasks* that are waiting to run in *priority based preemptive scheduling.* When the currently active *task relinquishes* control, either voluntarily or involuntarily, the *operating system* chooses the first *task* in the ready queue as the next *task* to run.

Real-time communication network. A real-time *communication* system within a *subsystem* that provides all services needed for the timely and dependable transmission of data between the *nodes.*

Real-time computer system. A computer system, in which the correctness of the system *behavior* depends not only on the logical results of the computations, but also on the physical time when these results are produced. A real-time computer system can consist of one or more computational *subsystems.*

Real-time responsive. A *task* switching policy that is consistent with real-time requirements. Specifically, whenever a higher *priority task* needs to run, it runs immediately. Contrast this with a policy in which the higher *priority task* is scheduled when it is determined that it must run, but the active *task* continues to run until the *dispatcher* interrupts it and only then switches to the higher *priority task.*

Reasonableness condition. The reasonableness condition of *clock synchronization* states that the granularity of the global time must be larger than the *precision* of the ensemble of *clocks.*

Reentrant code. A required quality for functions that may be interrupted and called again by an *interrupt service routine.* It must rely on stack variables only instead of using global data.

Reliability. The probability that a system will provide a specified service until time t, given that the system was operational at $t = t_0$. With an assumed constant failure rate of λ *failures/h,* reliability can be described as $R(t) = exp(-\lambda(t-t_0))$.

Relinquish. When the active *task* voluntarily gives up control of the *CPU* by notifying the *operating system* of its wish to do so.

Resource. A general term used to describe a physical device or software data structure that can only be used or accessed by one *task* at a time. Examples of physical devices include *CPU's,* physical transmission media, peripheral devices, etc. Likewise, shared data structures are software resources. Resource access problems can be solved by managing access using a *server task.*

Resource adequacy. A *real-time computer system* is resource adequate if there are enough computing *resources* available to handle the specified *peak load* and the *faults* specified in the *fault hypothesis.* Guaranteed response systems must be based on *resource adequacy* (→ *guaranteed timeliness*).

Resource controller. A computational unit that controls a *resource,* hides the concrete world *interface* of the *resource,* and presents a standard abstract message interface to the clients of the *resource.*

Rise time. The time required for a signal to rise to a specific percentage of its final equilibrium value as a result of change on the input.

RMA. Rate monotonic analysis. Real-time performance analysis of a system *design* that uses static *priority* driven *scheduling,* in particular, the rate monotonic static *priority* assignment, where *tasks* with shorter periods get the higher priorities. Rate monotonic scheduling is based on RMA.

Round robin scheduling. A *scheduling* policy in which all *tasks* are run in their turn, one after the other. When all *tasks* have run, the cycle is repeated. Various *scheduling* policies are run in round robin fashion, e.g., *cooperative scheduling, time-sliced scheduling,* and *cyclical scheduling.*

RPC. Remote procedure call. A type of *protocol* that allows a program on one computer to execute a program on a server computer. The client program sends a *message* with arguments to the server and the results are returned in another *message.*

RTOS. Real-time operating system. Software that governs processors and *resources* in a *real-time computer system.* It is a *multi-tasking* kernel with predictable worst case task switch times; it provides timer management, *communication* mechanisms and may also support *task* priorities (→ *operating system, fixed priority, dynamic priority*).

Safety. Safety is *reliability* regarding critical failure modes.

Safety-critical real-time computer system. A *real-time computer system* that must meet hard deadlines and fulfill additional safety properties, such as *partial correctness, mutual exclusion,* and absence of *deadlock.*

Sampling. In sampling, the *state* of a *controlled process* is periodically interrogated by the computer system. Sampling systems double as *low-pass filters.* Specifically, it refers to the conversion of *continuous signals* to *discrete signals* in data acquisition (→ *quantization, aliasing, polling*).

Sampling theorem. If a *continuous signal* composed of frequencies less than *f* is sampled at *2f,* all of the information contained in the *continuous signal* will be present in the sampled signal. Also called the Shannon or Nyquist sampling theorem.

Schedulability test. A schedulability test determines whether there exists a schedule such that all *tasks* of a given set will meet their *deadlines.*

Schedule. A partially ordered list specifying how contending accesses to one ore more sequentially usable resources will be granted. Eligibility parameters for creating a schedule are a *task's priority,* its *deadline,* or simply *precedence relations* among various tasks. Schedules are either computed statically (offline), or dynamically (online).

Scheduling. The technique of sharing a *resource* among various contenders according to their *temporal requirements,* at the same time considering *precedence relations* and *exclusion relations.*

SDL. Specification and description language. A specification language with both a textual and a graphic representation that is used in the telecommunications industry to specify the *behavior* of switching systems. It has a processed-based model of concurrency with asynchronous message passing between processes.

Semantic agreement. An agreement is called semantic agreement if the meanings of the different *measured data* are related to each other by a process model that is based on *a priori knowledge* about the physical characteristics of the *controlled process.*

Semaphore. A data structure that is used for inter *task communication.* Semaphores are usually provided by the *operating system.* A type of semaphore with just two *states* is called binary semaphore or mutex. It can be used to protect *critical sections* from interruption. A counting semaphore on the other hand is used to track multiple *resources* of the same type. An attempt to take a counting semaphore is blocked only if all of the available *resources* are in use.

Sensor. A device that measures physical phenomena and converts them to a data format suitable for further processing (→ *raw data, measured data*).

Server task. A *task* dedicated to the management of a specific *resource*. A server *task* accepts requests in the form of *messages* from other *tasks.* This approach does not block the calling *task* like a *semaphore* will when the *resource* is unavailable; instead the *message* is queued to the server *task,* and the requesting *task* continues to execute.

Signal conditioning. Signal conditioning refers to all processing steps that are required to generate a *measured data* element from a *raw data* element.

Slave process. A *CIP process* with several *behavioral modes,* one of which is chosen according to the *state(s)* of one or several *master processes.*

Soft deadline. A *deadline* for a result is soft if the result has utility even after the *deadline* has passed.

Soft real-time. Parts of a *real-time computer system* that are not concerned with any *hard deadline.* A *failure* to meet a *soft deadline* degrades performance without being catastrophic. *Scheduling* optimization criteria include: minimize the number of missed *deadlines,* minimize the mean (or maximum, or summed) lateness, miss *deadlines* according to *priority* (→ *hard real-time*).

Sporadic task. A *task* where the *task request times* are not known but where it is known that a minimum time interval exists between successive requests for execution (→ *periodic task, aperiodic task*).

Stability. In communication, the ability of a protocol to continue to operate effectively in the presence of network traffic variations and temporary network overloading.

State. The condition of an external or internal entity. The places in a *FSM.* Specifically, the unique status of a *process* in a *CIP system.* The combined states of all *processes* in a (synchronous) *cluster* defines the status of the *cluster.*

Statecharts. A graphic notation that generalizes finite state machines in the direction of hierarchy, concurrency, and timing. It allows child-parent and concurrent decomposition of states. Each edge in a graph may have multiple source nodes, and nodes may 'contain' other nodes. Events are represented as conditions and transitions.

State space explosion. The number of *states* in a system is exponential with respect to the number of *states* in the parallel components. This is a serious problem in analyzing systems that allow concurrent occurring of *events* (→ *event-triggered system*).

Static scheduling. A *scheduling* mechanism in which all *tasks* and *task priorities* are described and bound a priori. They cannot be changed during execution (→ *dynamic scheduler*).

Stress testing. The process by which a software system is put under heavy load and demanding conditions in an attempt to make it fail.

Subsystem. A subsystem of a *real-time computer system* is a single *node* or a set of *nodes* interconnected by a *real-time communication network.*

Suspension. The immediate cessation of a *task,* preceded by the saving of its *context.*

Synchronization. Sometimes one *task* must wait for another *task* to finish before it can proceed. Typically, producers and consumers must synchronize by sending and waiting for *messages.*

System. A combination of components working together. A computer system. Specifically, a *CIP* model, i.e., the functional description of an *embedded system,* consisting of synchronous *processes* grouped within asynchronous *clusters.*

Systematic fault tolerance. Fault tolerance mechanisms that are introduced at the *architecture* level, transparent to the application code (→ *application specific fault tolerance*).

Target. The target is the *real-time computer system* on which the software finally is to run. Executable files are *downloaded* from the *host* to the target.

Task. A task is the execution of a sequential program. Tasks are separately executable and loadable modules (→ *preemption*).

Task control block. A data structure that contains information about the *task.* It may include *task* name, start address, a pointer to the *task instance* data structure, stack pointer, stack top, stack bottom, *task* number, *message* queue, etc.

Task instance. A single generic *task* can be used to handle multiple instances of a device.

Task request time. The point in time when a *task* becomes ready for execution.

TDMA. Time division multiple access protocol. A multiplexing protocol that allows a number of users to access a single channel without interference by allocating unique time slots to each user. Each participant is assigned a specific time slot for transmission (→ *round robin scheduling*).

Temporal accuracy. *Measured data* is temporally accurate if the time interval between the moment 'now' and the point in time when the current value of the *measured data* was the value of the corresponding *controlled process* is smaller than an application specific bound.

Temporal control. Temporal control is concerned with the determination of the points in time when a *task* must be activated or when a *task* must be blocked because some conditions outside the *task* are not satisfied at a particular moment (→ *logical control).*

Temporal order. The temporal order of a set of *events* is the order of *events* as they occurred on the time line.

Temporal requirement. The points in time of its earliest start and latest end as required by a *task.*

Thrashing. The phenomenon that a *system's* throughput decreases abruptly with increasing load is called thrashing.

Tick. A tick (synonym. macrotick) of a synchronized *clock* is a specially selected *microtick* of this *clock.* The *offset* between any two respective ticks of an ensemble of synchronized *clocks* must always be less than the *precision* of the ensemble (→ *reasonableness condition*).

Time-sliced scheduling. Each *task* is allotted a certain number of time units during which it has exclusive control of the processor. After the time-slice has expired the *task* is preempted and the next *task* at the same *priority,* for which time-slicing is enabled, runs. If the concept of *fairness* is not implemented, a group of time-sliced high *priority tasks* will starve other lower *priority tasks.* Because *fairness* is repugnant to real-time principles and compromises the concept of *priority,* time-slicing is not a recommended real-time implementation scheme.

Time-triggered system. A *real-time computer system* is time-triggered if all *communication* and processing activities are initiated at predetermined points in time at an *a priori* designated *tick* of a *clock* (→ *event-triggered system*).

Timing failure. A timing failure occurs when a value is presented at the *system interface* outside the specified interval of real-time. Timing failures can only exist if the *system* specification contains information about the expected temporal *behavior* of the *system.*

Token bus. A *network topology* similar to a *token ring* that operates on a virtual ring. The main difference is that messages are *broadcast* simultaneously to all *nodes* instead of passing them along a physical ring. The increased minimum time for a token to traverse the logical ring makes prioritization of messages impractical. Cable breaks or failed nodes do not disable the entire network, but adding or deleting nodes requires a lengthy reconfiguration process.

Token ring. A ring-like *network topology* using point-to-point links. The right to transmit is contained in a token that is passed among the communicating partners. The protocol is deterministic with a moderate overhead, and throughput is good under heavy traffic conditions. The token's priority field permits global prioritization. Detecting duplicate or lost tokens is a problem, and possible broken connections and dead nodes may call for bypass hardware and dual rings.

Topology. The geometric arrangement of devices. In an implementation *architecture,* topology refers to the constituents of a *node,* i.e., processors, buses, peripheral devices etc. In a *network* it denotes the arrangement of the participating *nodes.* Common *network* topologies are bus, star, and ring.

Transducer. A device converting energy from one physical domain into another. The device can either be a *sensor* or an *actuator.*

Transient error. An *error* that exists only for a short period of time after which it disappears.

Transient fault. A *fault* that exists only for a short period of time after which it disappears.

Transition. The change of state in a state-based formal model such as a *Petri Net* or *FSM.* Specifically, the change from one *state* of a *CIP process* to another. Transitions are triggered by *events* or *pulses;* they can contain *operations* and output *actions* and/or *pulses.*

Trigger. A trigger is an *event* that causes the start of some *action.*

TTO. Time-triggered observation. An observation is time-triggered if the *point of observation* is triggered by a *tick* of the global time.

UML. Unified modeling language. A non-formal language for specifying, constructing, visualizing, and documenting software systems.

Value failure. A value failure occurs if an incorrect value is presented at the *system interface.*

VDM. Vienna development method. A collection of techniques for the formal specification and development of computing *systems.* It consists of a specification language, refinement rules, which allow one to establish links between abstract requirements specifications and detailed *design* specifications down to the level of code, and a proof theory in which rigorous arguments can be conducted about the properties of specified *systems* and the correctness of *design* decisions.

WAN. A *system* of interconnected *LANs.*

Watchdog. An independent external device that monitors the operation of a computer. The computer must send a periodic signal to the watchdog. If this *life sign* fails to arrive at the watchdog within the specified time interval, the watchdog assumes that the computer has failed and takes some action (e.g., the watchdog forces the *controlled process* into the safe *state*).

WCAO. worst case administrative overhead. The worst case *execution time* (*WCET*) of the administrative services provided by an *operating system.*

WCCOM. worst case *communication* delay. The maximum *duration* it may take to complete a *communication* transaction under the stated *load hypothesis* and *fault hypothesis.*

WCET. Worst case execution time. The maximum *duration* it may take to complete an *action* under the stated *load hypothesis* and *fault hypothesis,* quantified over all possible input data.

Yield. The process by which a *task* voluntarily *relinquishes* control so that other *tasks* can run.

Z. A formal specification notation based on set theory and first order predicate logic. It consists of a type system and a deductive system that supports reasoning about specifications.

This glossary helps to clarify often-used terms; it draws upon [Axe97, Bar99, Jen99, Smi97, and Tic96] and numerous contributions from Usenet [`news:comp.realtime`]. What had started as minor retouches to eliminate contradictions in the above exemplars, finally lead to an attempt of reconcilement. The effect is that the definitions in this glossary now differ from the ones in the sources…

References

[All97] R. J. Allen, *A Formal Approach to Software Architecture,* DPhil Thesis Carnegie Mellon University, Pittsburgh 1997.

[AlG97] R. J. Allen, D. Garlan, *A Formal Basis for Architectural Connection,* ACM Transactions on Software Engineering and Methodology 6(3):213-249, July 1997.

[ABB93] J. van Amerongen, A. W. P. Bakkers, J. F. Broenink, *The Twente Approach to System Level Embedded Controller Design,* Proc. of the 5th Transputer/Occam International Conference, Osaka 1993.

[AnG99] S. Anderson, C. Gurr, *Understanding Software Architecture: A Semantic and Cognitive Approach,* First Working IFIP Conference on Software Architecture WICSA1, San Antonio 1999.

[ARS91] K. Arvind, K. Ramamritham, J. A. Stankovic, *A Local Area Network Architecture for Communication in Distributed Real-Time Systems,* Real-Time Systems 3(2):115-147, May 1991.

[Axe97] J. Axelsson, *Analysis and Synthesis of Heterogeneous Real-Time Systems,* Diss. No. 502, University of Linköping 1997.

[Bac99] G. Bachmann Electronic GmbH, *Controller System M1, Operating Manual,* Tosters 1999.

[Bak97] A. W. P. Bakkers, *CSP and Programming,* Lecture, Las Vegas 1997.

[Bar99] M. Barr, *Embedded Systems Glossary,* online at `http://www.netrino.com/Publications/Glossary/`

[vdB94] M. von der Beeck, *A Comparison of Statechart Variants,* Formal Techniques in Real Time and Fault Tolerant Systems FTRTFT'94, Lecture Notes in Computer Science Vol. 863, 1994.

[BeG92] G. Berry, G. Gontier, *The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation,* Science of Computer Programming, 19(2):87-152, November 1992.

[Bir97] A. Birolini, *Quality and Reliability of Technical Systems. Theory – Practice –Management,* Springer 1997.

[Boe88] B. W. Boehm, *A Spiral Model of Software Development and Enhancement,* IEEE Computer Magazine 21(1988)5:61-72, 1988.

[BoH95] J. P. Bowen, M. G. Hinchey, *Seven More Myths on Formal Methods,* IEEE Transactions on Software Engineering 12(3):34-41, July 1995.

[Bri97] O. Bridal, *Issues in the Design and Analysis of Dependable Distributed Real-Time Systems,* Technical Report No. 297, Chalmers University of Technology, Göteborg 1996.

[BrW96] A. W. Brown, K. C. Wallnau, *Engineering of Component-Based Systems,* Selected Papers from the Software Engineering Institute, Carnegie Mellon University, Pittsburgh 1996.

[BEW94] Bundesamt für Energiewirtschaft und Schweiz. Nationalkomitee des Welt-Energie-Rates, *Schweizerische Gesamtenergiestatistik 1994,* Bulletin SEV/VSE, 16/95, August 1995.

[Bur92] A. F. Burke, *Hybrid/Electric Vehicle Design Options and Evaluations,* SAE SP-915 No. 920447, 1992.

[Cam89] J. R. Cameron (Ed.), *JSP and JSD: The Jackson Approach to Software Development,* IEEE Computer Society Press, 1989.

[Cat69] K. W. Cattermole, *Principles of Pulse Code Modulation,* Iliffe, London 1969.

[CIP00] CIP System AG, *CIP Tool User Manual,* Solothurn 1996/2000.

[CHT94] M. Colnaric, W. A. Halang, R. M. Tol, *A Hardware Supported Operating System Kernel for Embedded Hard Real Time Applications,* Microprocessors and Microsystems, 18(10):579-591, 1994.

[Dan97] A. Danuser, *Execution Analysis of Real-Time Software Running on Heterogeneous Distributed Computer Systems,* Diss. ETH Nr. 10900, TIK-Schriftenreihe No. 10, Zürich 1997.

[Dea92] E. B. Dean, *Quality Function Deployment for Large Systems,* International Engineering Management Conference, Eatontown 1992.

[DHE93] Ph. Dietrich, H. U. Hörler, M. K. Eberle, *The ETH Hybrid III Car - a Concept to Minimise Consumption and to Reduce Emissions,* ISATA'93, Aachen 1993.

[Die96] Ph. Dietrich, *Hybrid III, ein anspruchsvoller Forschungsantrieb; Erwartungen, Ergebnisse, Ausblick,* Erweiterte Konferenz über hybride Automobilantriebe, LVV/ETH, Zürich 1996.

[DiV96] Ph. Dietrich, A. Vezzini, *The Hybrid III: Concept for a Low Emission and High Efficiency Passenger Car,* Electric & Hybrid Vehicle Technology'95, UK & International Press 1996.

[Dij76] E. W. Dijkstra, *A Discipline of Programming,* Prentice Hall 1976.

[EIA98] Electronic Industries Association, CDIF Technical Committee, *CDIF CASE Data Interchange Format,* Arlington 1998, online at `http://www.eigroup.org/cdif/`

[ECS99] The Embedded C++ Technical Committee, *The Embedded C++ Specification,* 1999, online at `http://www.caravan.net/ec2plus/language.html`

[End96] M. W. Ender, *Der Taktbetrieb als teillastverbessernde Massnahme bei Ottomotoren,* Diss. ETH Nr. 11835, Zürich 1996.

[Fae93] O. Faergemand (Ed.), *Using Objects,* 6th SDL Forum SDL'93, North-Holland 1993.

[FeL98] P. Feiler, J. Li, *Consistency in Dynamic Reconfiguration,* 4th International Conference on Configurable Distributed Systems CDS'98, Annapolis 1998.

[FMN93] H. Fierz, H. Müller, S. Netos, *Communicating Interacting Processes, A Formal Method for the Development of Reactive Systems,* SAFECOMP'93, Poznan 1993.

[Fie94] H. Fierz, *SCSM – Synchronous Composition of Sequential Machines,* TIK Report No. 14, ETH Zürich 1994.

[Fie99] H. Fierz, *The CIP Method: Component and Model-based Construction of Embedded Systems,* European Software Engineering Conference ESEC'99, Toulouse 1999.

[Fie00a] H. Fierz, *Formal Semantics of CIP Models,* to appear.

[Fie00b] H. Fierz, *Problem Frames in Embedded System Development,* to appear.

[FJW97] V. Friesen, S. Jähnichen, M.Weber, *Specification of Software Controlling a Discrete-Continuous Environment,* The 19th International Conference on Software Engineering ICSE'97, Boston 1997.

[Gab96] R. P. Gabriel, *Patterns of Software: Tales from the Software Community,* Oxford University Press 1996.

[GHJ95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley 1995.

[GaS94] D. Garlan, M. Shaw, *An Introduction to Software Architecture,* Carnegie Mellon University Technical Report CMU-CS-94-166, January 1994.

[GKP00] D. Garlan, A. Kompanek, P. Pinto, *Reconciling the Needs of Architectural Description with Object-Modeling Notations,* unpublished.

[Göh98] P. Göhner, *Komponentenbasierte Entwicklung von Automatisierungssystemen,* GMA-Kongress 98, VDI Verlag, Düsseldorf 1998.

[GSH97] A. Gokhale, D. C. Schmidt, T. Harrison, G. Parulkar, *Towards Real-time CORBA,* IEEE Communications Magazine, 14(2), February 1997.

[Hal90] A. Hall, *Seven Myths of Formal Methods,* IEEE Transactions on Software Engineering 7(5):11-19, September 1990.

[Ham97] D. K. Hammer, *Process-Oriented Development of Embedded Systems: Modeling Behavior and Dependability,* Object-Oriented Real-Time Dependable Systems WORDS'97, Newport Beach 1997.

[Har87] D. Harel, *Statecharts: A Visual Formalism for Complex Systems,* Science of Computer Programming, 8(3):231-274, 1987.

[HaG96] D. Harel, E. Gery, *Executable Object-Modeling with Statecharts,* The 18th International Conference on Software Engineering ICSE'18, Berlin 1996.

[HPC93] D. J. Hatley, I. A. Pirbhai, D. Craignan, S. Gerhart, T. Ralson, *Strategies for Real-Time System Specification, An International Survey of Industrial Applications of Formal Methods,* Dorset House Publishing, Computer Systems Laboratory 1987/93.

[HeM96] C. Heitmeyer, D. Mandrioli (Eds.), *Formal Methods for Real-Time Computing,* John Wiley & Sons 1996.

[HPP81] K. Heninger Britton, R. A. Parker, D. L. Parnas, *A Procedure for Designing Abstract Interfaces for Device Interface Modules,* Software Engineering, 5:195-204, 1981.

[HBV97] G. Hilderink, J. Broenink, W. Vervoort, A. Bakkers, *Communicating Java Threads,* University of Twente 1997.

[Hoa85] C. A. R. Hoare, *Communicating Sequential Processes,* Prentice Hall International 1985.

[Hoa96] C. A. R. Hoare, *How Did Software Get So Reliable Without Proof?* Proc. of FME'96, Industrial Benefit and Advances in Formal Methods, Lecture Notes in Computer Science Vol. 1051, 1996.

[Hub97] D. Huber, *Tokyo Motor Show: innovative Japaner,* Neue Zürcher Zeitung Nr. 256, 4.11.1997.

[IAR99] IAR Systems, *IAR visualSTATE Concept Guide,* 1999.

[IEC93] International Electrotechnical Commission, *International Standard IEC 1131. Programmable Controllers,* Geneva 1993.

[Inm89] Inmos Limited, *The Transputer Data Book,* 1989.

[Inm91] Inmos Limited, *The Transputer Development and i*q *Systems Data Book,* 1991.

[JaD96] D. Jackson, C. A. Damon, *Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector,* IEEE Transactions on Software Engineering, 22(7):484-495, July 1996.

[JaR99] D. Jackson, M. Rinard, *Software Analysis: A Roadmap,* in A. Finkelstein (Ed.), The Future of Software Engineering, ACM Press, December 1999.

[JaC00] D. Jackson, J. Chapin, *Redesigning Air Traffic Control: An Exercise in Software Design,* IEEE Software, May/June 2000.

[JaZ93] M. A. Jackson, P. Zave, *Domain Descriptions,* IEEE International Symposium on Requirements Engineering, pp. 56-64, IEEE CS Press 1993.

[Jac94] M. A. Jackson, *Software Development Method,* in A. W. Roscoe (Ed.), Essays in Honour of C. A. R. Hoare, Prentice Hall International 1994.

[Jac95] M. A. Jackson, *Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudice,* Addison-Wesley and ACM Press 1995.

[Jac99a] M. A. Jackson, *Problem Analysis Using Small Problem Frames,* Workshop on Formal Aspects of Computer Science WOFACS'98, Special Issue of the South African Computer Journal, 22:47-60, March 1999.

[Jac99b] M. A. Jackson, *Problem Analysis and Structure,* unpublished, September 1999.

[Jen99] E. D. Jensen, *Real-Time Manifesto,* online at `http://www.real-time.org/`

[JoW98] B. F. Jones, J. Wegener, *Measurement of Extreme Execution Times for Software,* in Real-Time Systems, Colloquium organised by Professional Group A1 (Software Engineering) 98/306, York 1998.

[JoR97] M. B. Jones, G. Reeves, *What Really Happened on Mars?,* online at `http://www.research.microsoft.com/research/os/mbj/Mars_Pathfinder/`

[Jos96] M. Joseph (Ed.), *Real-time Systems; Specification, Verification and Analysis,* Prentice Hall International 1996.

[Joy96] I. Joyner, *A Critique of C++ and Programming Language Trends of the 1990s,* 3rd Edition 1996, online at `http://www.progsoc.uts.edu.au/~geldridg/cpp/cppcv3.html`

[KaM99] M. Katara, T. Mikkonen, *Design Approach for Real-Time Reactive Systems,* International Workshop on Real-Time Constraints RTC'99, Taipei 1999.

[KeR88] B. W. Kernighan, D. M. Ritchie, *The C Programming Language,* ANSI C Second Edition, Prentice Hall International 1988.

[Kop92] H. Kopetz, *Six Difficult Problems in the Design of Responsive Systems,* in 2nd International Workshop on Responsive Computer Systems, Saitama 1992.

[Kop97] H. Kopetz, *Real-Time Systems, Design Principles for Distributed Embedded Applications,* Kluwer Academic Press 1997.

[Kün87] A. Kündig, *A Note on the Meaning of 'Embedded Systems',* in Embedded Systems; New Approaches to their Formal Description and Design, Lecture Notes in Computer Science Vol. 284, 1987.

[Kün95] L. Küng, *Entwicklung einer Drehfeldmaschine mit optimalem Wirkungsgrad für ein Hybridfahrzeug,* Diss. ETH Nr. 11373, Zürich 1996.

[KVR92] L. Küng, A. Vezzini, K. Reichert, *Hybrid III: Concepts for the Electric System of a Hybrid Passenger Car,* 11th International Electric Vehicle Symposium, Florence 1992.

[KYL98] T. W. Kuo, W. R. Yang, K. J. Lin, *EGPS: A Class of Real-Time Scheduling Algorithms Based on Processor Sharing,* 10th Euromicro Workshop on Real Time Systems, Berlin 1998.

[Lab95] J. J. Labrosse, *Embedded System Building Blocks,* R&D Publications 1995.

[LaH94] K. Lano, H. Haughton (Eds.), *Object-Oriented Specification Case Studies,* Prentice Hall 1994.

[LFB96] P. G. Larsen, J. Fitzgerald, T. Brookes, *Applying Formal Specification in Industry,* IEEE Transactions on Software Engineering 13(3), May 1996.

[LCF97] B. Le Charlier, P. Flener, *Specifications Are Necessarily Informal or: Some More Myths of Formal Methods,* Journal of Systems and Software, Special Issue on Formal Methods Technology Transfer 40(3):275-296, March 1998.

[Lio96] J. L. Lions (Chair), *Ariane 5 Flight 501 Failure,* Report by the Inquiry Board, Paris 1996, online at `http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html`

[LiL73] C. L. Liu, J. W. Layland, *Scheduling Algorithms for Multi-Programming in a Hard Real-Time Environment,* Journal of the Association for Computing Machinery 20(1):46-61, January 1973.

[Luc96] D. C. Luckham, *Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events,* DIMACS Partial Order Methods Workshop IV, Princeton University 1996.

[Mar96] J. M. R. Martin, *The Design and Construction of Deadlock-Free Concurrent Systems,* DPhil Thesis University of Buckingham 1996.

[MAT99] The Mathworks, *MATLAB, The Language of Technical Computing,* 1988/99.

[MeT97] N. Medvidovic, R. N. Taylor, *A Framework for Classifying and Comparing Architecture Description Languages,* in M. Jazayeri, H. Schauer (Eds.), Software Engineering Notes 22(6), 1997.

[Mik99] T. Mikkonen, *The Two Dimensions of an Architecture,* 1st Working IFIP Conference on Software Architecture WICSA1, San Antonio 1999.

[Mil89] R. Milner, *Communication and Concurrency,* Prentice Hall International 1989.

[MIS98] MISRA, The Motor Industry Software Reliability Association, *Guidelines for the Use of the C Language in Vehicle Based Software,* The Motor Industry Research Association, Nuneaton 1998.

[Miz88] S. Mizuno (Ed.), *Management for Quality Improvement: The 7 New QC Tools,* Productivity Press Inc., Cambridge 1988.

[Mui11] J. Muir, *My First Summer in the Sierra,* The Riverside Press Cambridge 1911.

[Nau92] P. Naur, *Computing: A Human Activity,* ACM Press/Addison-Wesley 1992.

[NiB97] M. Nicholson, A. Burns, *Structuring Architectural Topologies for Real-Time Safety-Critical Systems,* Computer Science Report YCS 284, University of York 1997.

[OMG95] The Object Management Group, *The Common Object Request Broker: Architecture and Specification,* Framingham, 1995.

[OMG99] The Object Management Group, *OMG Unified Modeling Language Specification,* Framingham, 1999.

[Pai96] S. Paynter, *Real-time Mode-Machines,* in J. Jonsson, J. Parrow (Eds.), Formal Techniques for Real-Time and Fault Tolerance (FTRTFT), Lecture Notes in Computer Science Vol. 1135, 1996.

[PaD93] E. Panzieri, R. Davoli, *Real-Time Systems: A Tutorial,* in L. Donatiello, R. Nelson (Eds.), Performance Evaluation of Computer and Communication Systems, Lecture Notes in Computer Science Vol. 729, 1993.

[Par75] D. L. Parnas, D. P. Siewiorek, *Use of the Concept of Transparency in the Design of Hierarchically Structured Systems,* Communications of the ACM 18(7):401-407, July 1975.

[Par97] D. L. Parnas, *Software Engineering: an Unconsummated Marriage,* in M. Jazayeri, H. Schauer (Eds.), Software Engineering Notes 22(6), 1997.

[Pie09] H. Pieper, *Mixed Drive for Automobiles,* US Patent 1909.

[Pla99] P. J. Plauger, *Standard C/C++: Frequently Answered Questions,* C/C++ Users Journal, November 1999.

[Pow91] D. Powell (Ed.), *A Generic Architecture for Dependable Distributed Computing,* Research Reports ESPRIT, Project 818/2252 (Delat-4) Vol. 1, Springer 1991.

[Pur94] J. M. Purtilo, *The POLYLITH Software Bus,* ACM Transactions on Programming Languages and Systems 16(1):151-174, January 1994.

[RTJ99] The Real Time for Java Expert Group, *Real Time Specification for Java™,* Draft 1999, online at `http://www.rtj.org/rtj.pdf`

[RVK92] K. Reichert, A. Vezzini, L. Küng, *Elektroversorgungs- und Antriebssystem für das Hybrid III Fahrzeug,* 4. Kolloquium Antriebstechnik R. Bosch GmbH 1992.

[Rei97] D. J. Reifer, *Letter to the Editor,* Software Engineering Notes 22(1), 1997.

[Risks] The Risks Digest, *Forum on Risks to the Public in Computers and Related Systems,* online at `news:comp.risks`

[Sai96] H. Saiedian (Ed.), *An Invitation to Formal Methods,* IEEE Transactions on Computers 29(4):16-30, April 1996.

[Sam97] J. Sametinger, *Software Engineering with Reusable Components,* Springer 1997.

[Sch97] D. T. Schweizer, *Verifikation von Implementationen abstrakter Datentypen für sicherheitskritische Systeme am Beispiel von Oberon,* Diss. ETH Nr. 12056, TIK Schriftenreihe Nr. 19, Zürich 1997.

[SGW94] B. Selic, G. Gullekson, P. T. Ward, *Real-Time Object-Oriented Modeling,* John Wiley & Sons 1994.

[Sel99] B. Selic, *Turning Clockwise: Using UML in the Real-Time Domain,* Communications of the ACM 42(10):46-54, October 1999.

[SRL90] L. Sha, R. Rajkumar, J. P. Lehoczky, *Priority Inheritance Protocols: An Approach to Real-Time Synchronization,* IEEE Transactions on Computers, vol. 39, September 1990.

[SGC96] E. Shafai, S. Ginsburg, U. Christen, H. P. Geering, *Regelungstechnische Aspekte des Hybrid III-Antriebsstrangs,* Erweiterte Konferenz über hybride Automobilantriebe, LVV/ETH, Zürich 1996.

[Sha96] M. Shaw, *Procedure Calls are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status,* in D. A. Lamb (Ed.), Studies of Software Design ICSE'93, Lecture Notes in Computer Science Vol. 1078, 1996.

[Smi97] S. W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing,* California Technical Publishing, San Diego 1997.

[SpG84] A. Spector, D. Gifford, *The Space Shuttle Primary Computer System,* Communications of the ACM 27(9):874-900, September 1984.

[Sto96] N. Storey, *Safety-Critical Computer Systems,* Addison Wesley Longman 1996.

[StB94] A. D. Stoyenko, T. P. Baker, *Real-Time Schedulability-Analyzable Mechanisms in Ada9X,* in C. M. Krishna, Y. H. Lee (Eds.), IEEE Special Issue on Real-Time Systems 82(1):95-107, January 1994.

[StG95] P. Stravers, P. Groeneveld, *Design Automation for Embedded Systems,* Oldenbourg 1995.

[SwG95] B. Swaminathan, K. Goldman, *Dynamic Reconfiguration with I/O Abstraction,* 7th IEEE Symposium on Parallel and Distributed Processing SPDP'95, San Antonio, October 1995.

[ThC97] S. Thibault, C. Consel, *A Framework for Application Generator Design,* Symposium on Software Reusability SSR'97, in M. Harandi (Ed.), Software Engineering Notes 22(3), 1997.

[Tho94] U. Thoeni, *Programming Real-Time Multicomputers for Signal Processing,* Prentice Hall International 1994.

[Tic96] Tics Realtime, *Tics Realtime Definitions,* San Jose 1996, online at `http://www.concentric.net/~Tics/ticsdef.htm`

[Tru92] HO. Trutmann, *Control System for Research Car Hybrid III,* WoTUG-15, Aberdeen 1992.

[Tru96] HO. Trutmann, *Verteilt und doch zusammen – zum Steuersystem von Hybrid III,* Erweiterte Konferenz über hybride Automobilantriebe, LVV/ETH, Zürich 1996.

[Tru97a] HO. Trutmann, *Komplexe Technik schont die Umwelt,* Bulletin SEV/VSE 9/97, Mai 1997.

[Tru97b] HO. Trutmann, *Flexible Small Scale Parallel System in the Development of a Hybrid Vehicle,* PDPTA'97, Las Vegas 1997.

[Tru98] HO. Trutmann, *Generation of Embedded Control Systems,* 23rd IFAC/IFIP Workshop on Real Time Programming WRTP'98, Shantou 1998.

[Tru99a] HO. Trutmann, *Well-Behaved Applications Allow for More Efficient Scheduling,* 24th IFAC/IFIP Workshop on Real Time Programming WRTP'99, Schloss Dagstuhl 1999.

[Tru99b] HO. Trutmann, *On Component Reuse in Embedded Systems,* IEEE International Symposium on Industrial Electronics ISIE'99, Bled 1999.

[Tru00] HO. Trutmann, *Dynamic Processes in a Static Environment,* 25th IFAC Workshop on Real Time Programming WRTP'2000, Palma de Mallorca 2000.

[Unb87] H. Unbehauen, *Regelungstechnik,* Vieweg & Sohn, Braunschweig 1987.

[UpK94] B. P. Upender, P. J. Koopman, *Communication Protocols for Embedded Systems,* Embedded Systems Programming, 7(11), pp. 46-58, November 1994.

[VKR94] A. Vezzini, L. Küng, K. Reichert, *Performance Optimization for the Electrical Drive System of the Hybrid III Passenger Car,* 12th International Electric Vehicle Symposium, Anaheim 1994.

[Vez97] A. Vezzini, *Optimierung des elektrischen Antriebssystems für ein Hybrid-Fahrzeug,* Diss. ETH Nr. 11784, Zürich 1997.

[VIT87] VITA, *The VME-Bus Specification,* 1987.

[VIT91] VITA, *Mezzanine Bus Mini-Specification,* 1991.

[Wai94] W. Wait-Gibbs, *Software's Chronic Crisis,* Scientific American, pp. 72-81, September 1994.

[WaM85] P. T. Ward, S. J. Mellor, *Structured Development for Real-Time Systems,* Prentice Hall International 1985.

[Web97] M. Weber, *Systematic Design of Embedded Control Systems, Composing Models of System Structure and Behaviour,* GMD-Berichte, Oldenbourg 1997.

[Weg87] P. Wegner, *Varieties of Reusability,* in P. Freeman (Ed.), Tutorial: Software Reusability, IEEE Computer Society Press 1987.

[Wil85] R. J. Wilson, *Introduction to Graph Theory (3rd Edition),* Longman Scientific & Technical 1985.

[Wir77] N. Wirth, *What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions,* Communications of the ACM 20(11):822-823, November 1977.

[Wit96] Ch. Wittmer, *Entwurf und Optimierung der Fahrstrategien für ein Personenwagen-Antriebskonzept mit Schwungradkomponente und stufenlosem Zweibereichsgetriebe,* Diss. ETH Nr. 11672, Zürich 1996.

[XuP93] J. Xu, D. L. Parnas, *On Satisfying Timing Constraints in Hard-Real-Time Systems,* IEEE Transactions on Software Engineering 19(1):70-84, 1993.

[Zav84] P. Zave, *The Operational Approach versus the Conventional Approach to Software Development,* Communications of the ACM 27(2):104-118, February 1984.

[ZaJ97] P. Zave, M. Jackson, *Four Dark Corners of Requirements Engineering,* ACM Transactions on Software Engineering and Methodology 7(1):1-30, January 1997.

List of Figures

1: Comprehensibility vs. formality ... 5
2: Separation of concerns ... 7
3: Emphasis on implementation ... 8
4: Emphasis on functionality ... 9
5: Embedded system ... 15
6: Process step response ... 20
7: Filter magnitude and phase vs. frequency response ... 21
8: Jitter effects ... 22
9: Variations in processor resource allocation ... 23
10: Input signal range and resolution ... 24
11: Embedded system communication architecture ... 27
12: Gap in the development process ... 31
13: Component architecture mapped onto different implementation architectures ... 33
14: Modes of operation ... 38
15: Implementation variants ... 41
16: Basic activities in control applications ... 49
17: Aspects of decomposition ... 50
18: Simple control problem ... 52
19: Embedded system problem frame ... 53
20: Problem frames used in the decomposition ... 53
21: Requirement deduction problem frame ... 54
22: Functional problem frame ... 54
23: Model congruity problem frame ... 55
24: Connection problem frame ... 56
25: Instrumentation problem frame ... 56
26: Connection transformations ... 56
27: Domain classification ... 57
28: Spheres of activity ... 58
29: Robot: operational architecture ... 63
30: Robot: two-processor implementation ... 64
31: Robot: fully distributed implementation ... 65
32: Component architecture notation ... 67
33: Functional component ... 69
34: Event extractor component ... 69
35: Connective components ... 70
36: Connectors ... 71
37: Robot: component architecture ... 73
38: Development ripples ... 74
39: State model of finite state machine ... 76
40: State diagram of finite state machine ... 76
41: Example state diagram: communication protocol ... 77
42: Functional reactive block ... 81
43: Process structure ... 82
44: Motor controller: channels and messages ... 83
45: Motor controller: communication net ... 84
46: Motor controller: interaction net ... 84
47: Motor controller: pulse translation ... 85
48: Master-slave hierarchy ... 86
49: Motor controller: master-slave hierarchy ... 87
50: Motor controller: Boss (as master) ... 87

51: Motor controller: Controller slave modes ... 88
52: Motor controller: Controller combined mode ... 89
53: Motor controller: inquiry ... 89
54: Implementation of a CIP unit ... 90
55: Robot: internal communication ... 92
56: CIP high-level process interface ... 93
57: Motor controller: cascades ... 95
58: Motor controller: interaction trees ... 96
59: Interfacing the functional block ... 98
60: Device access sequence ... 99
61: Optimizing device accesses ... 101
62: Encoding of discrete values ... 103
63: Input data flow ... 104
64: Unbalanced capacity utilization caused by signal periodicity ... 105
65: Input access schedule for three frequency classes ... 105
66: Processor utilization 42% (15 input signals, class HI: 2, MED: 9, LOW: 4) ... 107
67: Processor utilization 99% (75 input signals, class HI: 8, MED: 39, LOW: 28) ... 107
68: Input data distribution strategies ... 109
69: Input data distribution ... 110
70: Input data copying ... 111
71: Events in a continuous signal ... 112
72: Output data flow ... 113
73: Output access queues and schedule ... 114
74: Communication data flow ... 114
75: Execution sequence ... 116
76: Motor controller: processor time demands ... 119
77: Event extractor data structures ... 120
78: Global and local schedules ... 123
79: Processing task schedule ... 124
80: Implementation variants ... 125
81: Single and dual processor implementations ... 126
82: Simulation stages ... 126
83: Interface implementations for different modes of operation ... 127
84: Robot: interaction net (Motion) ... 132
85: Robot: cascade subsets (Motion) ... 132
86: Modes of operation and processor utilization ... 133
87: Processor utilization at mode change ... 134
88: Memory allocation at mode change ... 134
89: Hybrid concepts ... 136
90: Hybrid III drive system ... 139
91: Hybrid III drive train ... 140
92: Hybrid III system structure ... 143
93: Operational structure of the control system ... 145
94: Partial system ... 146
95: Implementation structure of the control system ... 147
96: IMS T805 transputer block diagram ... 149
97: Data stream networks ... 150
98: Point-to-point connection between transputers ... 150
99: Message levels ... 151
100: Link broadcast mechanism ... 152
101: Input and output cycles ... 152
102: Synchronous input and asynchronous output ... 153
103: Data input path variants ... 153
104: Cabling and galvanic isolation ... 155
105: Hybrid III software development strands ... 156
106: Development setup ... 157
107: Partial implementation ... 158
108: Hybrid III project structure (excerpt) ... 160

109: Complete implementation 161
110: Principle of rotor speed measurement 163
111: Distributor net implementation variants 164
112: Task structure of cluster O 166
113: Task structure of cluster IO 167
114: Task structure of clusters C, E, M, H, and S 168
115: Task structure of a plain Slave 169
116: Task structure of an RSlave 169
117: Configurations of master and slave nodes 170
118: Dependencies 172
119: Input phases 187
120: Output phases 188
121: Code generator program structure (acc) 194
122: Example signal configuration 195
123: Internal signal representation 196

List of Tables

1: Relative number of Bytes transmitted 110
2: Robot modes of operation 131
3: Modes of operation and memory requirements 133
4: Input configuration items 187
5: Output configuration items 188
6: Delay timer values 191
7: ST (symbol table) data fields 194
8: ST (symbol table) data field usage 195
9: AP (access procedure) data fields 197
10: APL (access procedure list) list element 197
11: APL (access procedure list) lists 197
12: IP (initialization procedure) list element 198
13: Sampling frequencies 198

Index

A

abstraction
- deficient 10, 97
- in development 9, 37
- from implementation 66
- level of 11–12, 18–19, 35–36, 49–50, 58–59, 62, 79–80, 92–93, 95–96, 115, 171, 174–175

acc. *See* generator, grammar, scanner, syntax

access
- distribution 199–200
- input 100
- interleaved 100
- low-level 98
- module 194
- multiple 99
- multi-stage 100
- output 100, 113
- procedure 197

accuracy
- interval 207
- measurement 207
- sensor 24
- signal 24
- temporal 218

action 55, 93
- CIP 94, 207
- irrevocable 213
- message 54–55, 57–59, 113
- phenomenon 53–56, 207

actuator 36, 219
- device 207

agreement, semantic 217

alarm, shower 207

aliasing, problem 21, 207

API. *See* interface

application
- industrial 174
- software 207

architecture 32–34, 49–50, 207
- component 32–34, 50–51, 62, 66, 73, 80
- configuration 178
- description language 61, 66–68, 178
- implementation 33–34, 50–51, 62, 64–65
- operational 33–34, 50–51, 62
- style 66–68

ASIC 207

assertion 207

assumption
- for analysis 45

assumption, cont.
- coverage 58, 208

ATM 208

attachment, declaration 182

availability
- assessment 25
- of energy converter 142
- resource 24–25
- of sensor/actuator 52
- of a system 208
- system 28

B

Backus-Naur Form 203–206, 208

bandwidth
- channel 208
- waste of 99

behavior 208
- abstraction of 96
- change 86
- continuous 81
- correct 15
- deterministic 12, 27, 118
- dynamic 121
- essential 92
- functional 49, 54, 159
- non-deterministic 44
- predictable 18, 29
- reactive 18, 30, 36, 53
- required 52–55

binary countdown. *See* bit dominance protocol

block, reactive 81, 90

BNF. *See* Backus-Naur Form

bottom-up 31–32, 66

bounce, contact 209

buffer 165–166

bus
- address 207
- data 210
- field- 27, 46, 212
- mezzanine 154
- VME 154, 160

busy waiting. *See* polling

C

CAN. *See* network

carrier
- board 186
- production 186

cascade
CIP 95
execution 81
motor controller 95
CDIF 7
channel
access delay 208
bit-length of a 208
capacity 102
CIP 83–84
characteristics, application 173
CIP
machine 90
method 78, 80, 208
shell 90
unit 68, 80, 90, 125, 132, 164–165, 181
CIP Tool 80
class, frequency 106
clock, device 208
cluster
CIP 160, 208
activation 85
CNI. *See* interface
code
functional 91, 117
structure 119
as switch 117
table 118
generated 88
generation 35, 42, 99, 105, 177
generator 174
legacy 36, 41, 63, 79, 121
object 35, 50, 59
reentrant 216
static 130
communication 49, 209
asynchronous 26
bandwidth 27
broadcast 208
buffer 98
channel, motor controller 83
CIP 83
controller 209
interface 114
latency 27–28
message 98, 114
multicast 214
net, motor controller 84
network 214
protocol 215
semaphore 217
synchronous 26
traffic 28
compiler 209
cross- 209
complexity 49, 62, 158
component
architectural element 61
component, cont.
connective 69
declaration 180
event extractor 69
functional 68, 75, 80
Hybrid III 141
reuse 32, 175
composability 209
composition 31–32, 41, 78
concern, separation of 36
concerns, separation of 6, 11, 36, 50, 158
condition
CIP 89
reasonableness 216
configuration
connection 158
definition 129
dynamic 129
flexibility 47
strategy 130
congestion, of a network 209
connection 37, 159
chores 98
configuration 209
specification 98
connector
architectural element 61
declaration 181
high-level 71–72
low-level 71
constant
declaration 179
production 185
construction 32
of a system 209
contention 209
context 209
switch 209
control 36, 209
continuous 20
algorithm 20
logical 213
temporal 218
controller, programmable logic 8, 215
conversion 120
A/D 99
cooperation
asynchronous 78
synchronous 78
CORBA 26, 209
correctness
functional 16
partial 214
of program 34
temporal 16
cost, per node 47
CPU 209
CRC 209

CSMA. *See* protocol
CSP. *See* formal method
CVT. *See* transmission

D

data
- acquisition 103
- conversion 98, 103
- dynamic 130
- encoding 210
- measured 213
- module 194
- output 98
- range 24
- raw 98, 216
- representation, internal 101
- structure
  - atomic 208
  - dynamic 117
- transport 104, 108
- verification 104
- width 99

deadline 22, 45, 210
- hard 20, 212
- interval 210
- soft 23, 217

deadlock 210
debugger 17, 210
declaration
- attachment 182
- component 180
- computation 181
- connector 181
- constant 179
- glue 182
- instance 182
- message 181
- port 180
- role 181
- type 179
- version 179

decomposition 31–32, 49, 145
delay 188
- accumulation of 106
- channel access 208
- conversion 100
- propagation 215
- variation. *See* jitter

description
- algorithmic 76, 79
- appropriate 173
- behavioral 60, 76
- boxes with lines 50, 61–63
- formal 98
- informal 64
- non-formal 96

design, of a system 210
development
- concept 6
- cross- 17, 30
- emphasis 7
- gap in the 30
- independent 51, 63, 157
- loop 8–9
- method
  - CIP. *See* CIP
  - JSD 78, 213
  - ROOM 78
  - SDL 19, 78, 217
  - Statecharts 19, 78, 217
  - *See also* formal method
- model 37
- phase 49
- problem-oriented 58, 66, 75, 171
- strands 156

device
- access 99, 102
  - time 100
- input 57
- with input and output 101
- peripheral 97, 100
- transducer 219
- virtual 97

dispatching, sequential 91
display, production 192
DMA 149, 210
domain
- classification 57
- controlled 52

download 210
driver, device 210
DSP 211
duration 211

E

EDF. *See* earliest deadline first scheduling
emphasis
- on functionality 8
- on implementation 7

encoding, of discrete values 102
energy
- consumption 135
- density 137
- efficiency 137
- recovery 139
- storage 137–138

engine
- internal combustion 137
  - control 163

environment 16–17, 39, 49
- static 174
- of a system 211

equation, difference 210
error 211
- Byzantine 208

error, cont.
masking 211
transient 219
ESTEREL. *See* formal method
Ethernet. *See* network
event 53–58, 93, 211
CIP 94, 211
configuration 189
discrete 19
extraction 57, 98, 111–112, 120–121
algorithm 112
hysteresis 111
message 54–55, 57–59, 111
phenomenon 211
primary 215
process 93
production 192–193
sporadic 19, 22
temporal 93
evt. *See* generator, grammar, scanner, syntax
exception 211
exclusion, mutual 214
execution
path 118
worst case *See* worst case
periodic 19–20, 117
property 68, 72, 115, 173
sequence 100, 116–117
time 39, 106–107, 116, 122–125, 187–188
measurement 25, 106
worst case *See* worst case
executive, run-time 123
exhaust gas
cleaning 139, 142
emissions 138
extension
of finite state machine 77–79, 82, 88, 90
of formal model 79
motor controller 90

F

failure 17, 211
malign 213
rate 29, 212
timing 219
value 219
fairness 212
fault 29
device 29, 212
hypothesis 212
tolerance 18, 29–30, 212
application specific 207
systematic 218
transient 219
filter
anti-aliasing 21, 207
low-pass 103, 124, 213
setting 99
FIT. *See* failure rate
flow control
back pressure 208
explicit 211
implicit 212
transmission 212
flywheel 141
FSM. *See* formal method
functionality 37, 159

G

generator
acc 159, 183, 194
dsp 159
evt 159, 189
symbol table 194
glue, declaration 182
grammar
acc 205
evt 206
pvt 203

H

hardware 212
capacity 43
Hybrid III, broadcast 152
peripheral 98–99
resource 216
topology 64
HI input. *See* input production
hierarchy 78, 145
motor controller 87
host 212
hybrid
concept 136
parallel 137, 139
serial 136
vehicle 135
Hybrid III
cluster IO 162, 166
cluster O 165
clusters 146
clusters CEMHS 167
component group 154
configuration 158
connection 159
control system 145
functionality 159
modeling 158
power supply 155
processing node 157
project
description 138
structure 160
satellite 154
signal conditioning 162
simulation 157

Hybrid III, cont.
structure
implementation 147
operational 145
task
abyss 165
adaptor 165
broadcast receiver 165, 169
broadcast sender 166
CIP 165, 170
clock 166
communicator 151, 165, 170
event extractor 167
input 152
input converter 166
monitor 165
output 152
output converter 166
receiver 169
sender 169
strategy master 148
submaster 148
system master 148
tick generator 165
working cluster 167

I
ICE. *See* internal combustion engine
implementation 41, 49
assembly 173
assembly of 171
constraint 42
deterministic 81
distributed 125
efficiency 110
lean 11–12
simulation 126
single-processor 125
finding suitable 127
variant 99, 124–125, 129
include, production 185
industry 173
initialization 186
init, production 186
procedure 198
input 36, 49, 52–58, 99, 101
access
interrupt 100
static schedule 100
as task 100
configuration 183
data, distribution 108
device 57
production 187
routing 154
schedule 105–106, 120
as task 122
inquiry
CIP 83, 213
motor controller 89
instance, declaration 182
instant 213
instantiation 213
interaction
asynchronous 79, 83
CIP 84, 213
cyclic 85
higher-order 78, 86–87
internal 95
net, motor controller 84
pulse 85
restriction 95
synchronous 79, 84
tree 96
motor controller 96
Interbus-S. *See* network
interface 213
abstract 97
API 207
CNI 208
communication 83
high-level 93
instrumentation 97, 213
interaction 83
low-level 32
MMI 214
M-module 170
node 213
virtual 81, 93–94
VME-bus 166
interrupt
hardware 100, 213
latency 213
procedure 41
service routine 213
IO_Init, function 186

J
jitter
communication 27, 213
input 20–22, 97, 106
output 22
processing 22, 29, 213
JSD. *See* development method

K
knowledge, a priori 17, 44, 46, 207

L
LAN. *See* network
life sign 213
link
broadcast 151
failure 151, 154, 165
protocol 150

LL. *See* least laxity scheduling
load
distribution 105
hypothesis 25, 213
peak 25, 214
processor 105, 107
loop, unbounded 117
LOW input. *See* input production

M

machine
CIP 90
connection 58
control 52
finite state. *See* formal method
functional 57–58
Mealy 76, 79, 212
Moore 212
reactive 91
maintainability 213
management, thermal 141
mask. *See* pattern production
master-slave
connector 87
graph 87
relationship 87
measurement
accuracy 101
performance 40
precision 215
quantization 216
representation 101
rotor speed 162
mechanism, run-time 131
MED input. *See* input production
membership, service 214
memory
allocation 117
fragmentation 212
message
action 54–55, 57–59
attribute 93
communication 214
declaration 181
delay 28, 46
event 54–55, 57–59
size 28
method
development
CIP. *See* CIP
JSD 78, 213
ROOM 78
SDL 19, 78, 217
Statecharts 19, 78, 217
formal
CSP 149, 210
ESTEREL 78, 211
FSM 19, 76, 212
method, cont.
formal
Petri Net 19, 215
VDM 19, 219
Z 19, 219
M-ICX. *See* internal combustion engine control
MMI. *See* interface
mode
change 86, 88
CIP 86, 214
of operation 174
of operation. *See* operation mode
setting 87
model
abstract 52
behavioral, context 93
checking 214
CIP 91
context 94
essential 75
formal 19, 75
functional 50, 93–94, 158, 212
context 112
context of 93
initialization 95
life cycle 37
states of a 54
module
generator 193
M- 154, 161
mezzanine 154, 186
production 186
monitor, task 165
motor, electric 137
M-RPM. *See* rotor speed measurement
MTTF 65, 214

N

NBW. *See* protocol
network
backbone 27, 208
CAN 46, 208
collector 150, 164
distributor 150, 164
Ethernet 211
FAST-bus 46
instrumentation 27, 46
Interbus-S 46, 213
LAN 213
Profibus 46, 215
real-time communication 27, 216
token bus 219
token ring 219
topology 46, 219
WAN 27, 219
node
interface 213
processing 16, 46, 64–65

node, cont.
  production 191

O

object, controlled 16
observation
  point of 215
  time-triggered 219
offset 214
operation
  atomic 208
  CIP 90, 214
  duty cycle 138–141
  initial 39
  mode 37–38, 157, 214
    simulation 126
order
  causal 208
  temporal 70, 72, 75, 83, 218
orthogonality 36–37
outptu, schedule 121
output 36, 49, 52–58, 99, 101, 112, 121
  access 113
  configuration 183
  production 188
  queue 113
  routing 154
  schedule 100, 113
  as task 122
  task 114
overload, protection 44

P

PAR. *See* protocol
paradigm, programming 36
parallelism, conceptual 91, 118, 174
parser, module 193
pattern
  abstract 96
  production 189
performance
  analysis 39–40, 42, 118
  average 17
  degradation 118
  increase 100
  measurement 118
  run-time 133
  system 18, 44
  verification 39
  worst case. *See* worst case
Petri Net. *See* formal method
phenomenon
  controllable 52–53
  essential 55
  shared 53–55, 57
  of the world 52, 81
pivot 51, 66, 73
PLC. *See* programmable logic controller
port
  declaration 180
  property 75
predictability 30, 41, 215
preprocessor directive, include 185
priority
  dynamic 211
  fixed 45, 212
  inversion 44, 215
  message 47
  task 46
problem
  area 12, 175
  connection 54–55, 57
  decomposition 49
  frame 52–57
  functional 54, 75
  separation 171
procedure call, remote 216
process
  activation, sequence 85, 95
  array 83
  behavior 39
  CIP 82, 159, 215
  controlled 209
  external 81, 94
  function 94
  interface 94
  master 213
  slave 217
processing
  requirement, upper bound on 118
  time 118
processor
  general-purpose 171
  load 99
  multi- 16, 71, 80, 214
  node 214
  single 16, 67, 115, 128, 164, 169, 189
  utilization 106–107
production
  carrier 186
  constant 185
  display 192
  event 192–193
  generator 179, 184, 190
  include 185
  init 186
  input 187
  module 186
  node 191
  output 188
  pattern 189
  slave 192
  timer 191
  version 185, 191
productivity 39
Profibus. *See* network

profile, automatic 117
programming
  language 34
    assembly 34–35, 173, 207
    C 34–35, 88, 106, 117, 208
    C++ 34–35
    syntax 8
    restricted use of 34–35, 42, 75
  paradigm 34
progress, independent 39
property, deterministic 121
protocol
  bit dominance 208
  broadcast 151
  communication 215
  connection oriented 209
  contention 209
  controlled access 209
  CSMA 210
  efficiency 28, 46
  end-to-end 211
  flexibility 28
  latency 28, 46, 216
  media access 28, 46, 213
  NBW 214
  PAR 214
  point-to-point 151
  polling 215
  priority ceiling 215
  TDMA 218
pulse
  CIP 216
  multicast 85
  translation, motor controller 84
  transmission 85
pvt. *See* grammar, syntax

Q

QoS. *See* quality of service
quantization 20–21

R

range
  cruising 137
  of a value 24
real-time
  hard 212
  image, phase sensitive 72
  responsive 216
  soft 217
reconfiguration
  dynamic 128
  static approach 128–129
recursion 117
region, forbidden 212
relation
  exclusion 101, 211
  precedence 215
relationship, behavioral 54
reliability 18, 28–29, 65, 216
relinquish 216
repeatability 25
report, CIP 96
requirement
  accuracy 24
  communication 26
  connection 55–56
  customer 17–19, 51, 53
  deduction 55
  functional 18, 53, 55
  implementation 25
  interface 59
  minimum 11
  real-time 23, 30
  reliability 28, 30
  temporal 19, 38–39, 44–45, 218
  timing 16
research, real-time system 173
resolution 24
resource
  adequacy 24, 65, 216
  computational 209
  controller 216
  hardware 216
  management 17, 25, 43
  need 25
  software 216
  common use of 101
  utilization 12, 16–17, 40–41, 45, 174
response, predictable 121
reuse
  component 175
  software 175
risk, of testing 39
RMA. *See* scheduling
robot
  architecture
    component 73
    implementation 64
    operational 63
  communication, channel 91
  operation, mode of 131
robustness
  communication 28, 46
  operational 47
role, declaration 181
RPC. *See* remote procedure call
RTOS. *See* operating system

S

safety 216
sampling 21, 217
  decimation 210
  interpolation 213
  over- 22, 214
  polling 215

sampling, cont.
rate 20–21, 104, 106, 185, 198
theorem 21, 217
satellite 154, 162
scanner
acc 204
evt 205
module 193
pvt 203
S-card 162
schedule 217
global 122
input 105, 120
non-deterministic 44
output 100, 113, 121
processing 123–124
run-time 118
static 100
scheduler, module 193
scheduling 217
cooperative 209
cyclical 210
dispatch 210
dynamic 43–44, 122, 129, 211
earliest deadline first 211
laxity 213
least laxity 213
preemptive 44, 122, 211, 213, 215
principles 43
rate monotonic 45, 216
round robin 45, 123, 216
static 43, 122, 218
time-sliced 45, 218
SDL. *See* development method
section, critical 209
selection, of signals 199
semantics
acc 183–189
CIP 80
component 61
composition 82
evt 190–193
formal representation 79–80
programming language 42
pvt 178–182
sensor 36, 217, 219
intelligent 213
server, task 217
service
quality of 27, 47
remote 148
shell, CIP 90–91
signal
conditioning 217
continuous 101, 209
conversion 114
discrete 102, 210
encoding 102
signal, cont.
pattern 111
simulation 38–39, 126–127, 157, 174
implementation 99, 126
slave
motor controller 88
plain slave 168
production 192
ring slave 168
solution, complete 173
specification 174
connective 50
language 75, 178, 183, 190
stability 217
state 217
CIP 217
continuous 94
discrete 93
ground 212
inspection 85, 89
instantaneous 79
space, explosion 3, 76–77, 218
transition 219
Statecharts. *See* development method
structure
behavioral 86
implementation 147
operational 145
synchronization
explicit 211
implicit 213
syntax
acc 204
evt 205
pvt 203
system
CIP 218
composition 31–32, 41
control 15, 209
decomposition 31–32, 145, 173
distributed 16, 210
drive 137
embedded 15, 211
event-triggered 18–19, 25, 211
fail-operational 211
fail-safe 211
fail-silence 211
hard real-time 16, 212
heterogeneous 212
hierarchy 145
initialization 95
operating 43, 214
RTOS 17, 216
partitioning 91
performance 44
reactive 16
real-time 16, 216
safety-critical 28–30, 217

system, cont.
  single-processor 16
  soft real-time 16, 23, 217
  sub- 218
  time-triggered 18–19, 25, 218

T
table, off-line 130
target 218
  final 39
task 218
  active 207
  aperiodic 207
  blocking 208
  CIP 165
  control block 41, 218
  idle 212
  instance 218
  multi- 123, 214
  periodic 214
  preemption 44, 215
  priority 46, 215
  ready queue 216
  request time 218
  sporadic 44–45, 217
  suspension 218
  synchronization 41, 218
TDMA. *See* protocol
test
  performance 39, 41
  schedulability 217
  stress 218
  on target 10
thrashing 27, 218
tick 218
  micro- 214
time
  execution 211
  to failure. *See* MTTF
  minimum between events 214
  rise 216
timeliness, guaranteed 25, 212
timer, production 191
timing, assertion 118
token bus. *See* network
token ring. *See* network
tool 36–37, 159
  domain 37
  integration 36
top-down 10, 31–32
topology
  bus 46
  hardware 33, 64, 219
  network 46, 219
  ring 46, 151
torque 138
TRAM. *See* transputer module
transaction, sporadic 47
transition
  CIP 82, 84–85, 219
  structure 86
transmission, CVT 138, 141
transputer 148–149
  module 149, 160
trigger 219
TTO. *See* time-triggered observation
type, declaration 179

U
UML 7, 96, 219
unit, CIP 90–92
unit. *See* CIP unit
use, intermittent. *See* duty cycle operation

V
variation, requirement 132
VDM. *See* formal method
vehicle
  hybrid 135
  interface 142
verification 39
version
  declaration 179
  production 185, 191

W
WAN. *See* network
watchdog 219
WC… *See* worst case
weight 138
worst case
  administrative overhead 219
  behavior 28, 46
  communication delay 219
  execution
    path 12, 39–40, 42, 91, 118–119
    time 124, 219
  performance 17
  response time 47

Y
yield 123, 219

Z
Z. *See* formal method
zero-emission 139