

TECHNISCHE UNIVERSITÄT MÜNCHEN
INSTITUT FÜR INFORMATIK

Doctoral Thesis

Timing Constraints in Distributed Development of
Automotive Real-time Systems

Oliver Scheickl

Technische Universität München
Institut für Informatik

**Timing Constraints in Distributed
Development of Automotive Real-time Systems**

Oliver Scheickl

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Alois Knoll

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Dr. h.c. Manfred Broy
2. Univ.-Prof. Dr. Rolf Ernst,
Technische Universität Braunschweig

Die Dissertation wurde am 3. März 2011 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 16. August 2011 angenommen.

Acknowledgements

This thesis was created during my work as doctoral candidate at BMW Car IT GmbH in Munich. From 2007 to 2010 I worked there on several internal and external projects related to the modeling and analysis of embedded automotive real-time systems. I actively participated in the timing subgroup of AUTOSAR, which developed and released a timing model for the automotive industry's embedded system specification standard.

First and foremost I wish to thank Prof. Manfred Broy for supervising my thesis and for all the fruitful discussions during my time working on it. Further I thank Prof. Rolf Ernst for being the second referee of the thesis and for his helpful feedback.

I am thankful to all of my colleagues at BMW Car IT for an incredibly inspiring working environment and for their kind help and support at all times. I extend my thanks especially to Prof. Harald Heinecke and Michael Rudorfer for guiding my work and for providing me with such a great opportunity in balancing a combination of scientific and industrial work. For their continuous support, countless discussions and finally for reviewing and commenting on my thesis, I am deeply grateful to my colleagues Christoph Ainhauser, Thomas Benedek, Dr. Sebastian Benz, Paul Hoser and especially Dr. Reinhard Stolle, who inspired and motivated me to write a doctoral thesis.

Finally, I thank my parents for their invaluable support throughout my studies and the work on my thesis. All of this has only been possible because of you.

Abstract

The amount of functions that are realized by software is increasing in modern automobiles. Most innovations in the automotive industry are driven by such functions today. Many of these functions, especially safety-relevant functions, must fulfill strict timing constraints. This thesis introduces a new development approach for automotive real-time systems.

Traditionally, the overall vehicle electrical system is designed and integrated by the car manufacturer (OEM). Suppliers actually develop different subsystems in a so-called distributed development process. The OEM specifies the desired functionality, and suppliers develop their subsystem according to its specification. End-to-end car functions are often realized by software components that are distributed over several electronic control units (ECU), which exchange data via communication busses. ECUs and software components are typical subsystems. The control and data paths of functions thus often cross several subsystems, which are typically provided by different suppliers.

The response times of functions, which must fulfill given timing constraints, include execution and transmission times along their control and data paths. In such a distributed development process of distributed automotive real-time systems, OEMs today face a challenging system integration task. First, they must ensure that the combined timing behavior – i.e. execution and transmission times – of all supplied subsystems fulfills all function timing constraints of the system. Second, if a timing constraint is not fulfilled, the OEMs need to know which subsystem causes the problem and how the problem can be solved.

This thesis proposes a solution to that system integration challenge. In our approach, the specifications for the suppliers include requirements for the desired subsystem timing behavior. However, the subsystem timing requirements are not independent from each other. Rather they are derived from the function timing constraints. The timing behavior of a supplied subsystem is reported back to the OEM in a way that abstracts from the underlying implementation details by providing data path-related timing behavior guarantees. By comparing the timing requirements with the reported guarantees of all subsystems, timing problems can be localized and an according reaction in terms of an intelligent modification of the timing requirements can be triggered. In an iterative process the approach tries to find a suitable timing specification for all subsystems, until all function timing constraints are fulfilled.

The process is based on TIMEX, a new timing model for the specification of both function timing constraints and derived subsystem timing requirements. Further, the TIMEX development methodology describes and formalizes an algorithm to derive and iteratively maintain subsystem timing requirements. The benefit of the methodology is that the timing behavior of subsystems can be analyzed independently from each other. Timing problems that cause unfulfilled function timing constraints can be identified in the model. They are then repaired by a structured, systematic redistribution of time budgets between subsystems.

Zusammenfassung

Die Anzahl Software-basierter Funktionen in Fahrzeugen nimmt stetig zu. Viele der Funktionen müssen strikte zeitliche Anforderungen erfüllen. Die Software-Komponenten solcher Funktionen sind zunehmend über kommunizierende Steuergeräte verteilt. Wegen der für die Automobilindustrie typischen verteilten Entwicklung, in der mehrere Zulieferer verschiedene Steuergeräte oder Software-Komponenten liefern, müssen zeitliche Anforderungen während der Entwicklung zwischen dem Automobilhersteller und den Zulieferern koordiniert werden. Diese Arbeit stellt einen neuen Ansatz vor, um zeitliche Anforderungen von Funktionen auf Subsysteme abzubilden und während der Entwicklung Zeitbudgets für Subsysteme zu koordinieren. Basis des Ansatzes ist das in der Arbeit eingeführte Modell für zeitliche Anforderungen TIMEX. Die Regeln zur Abbildung von Funktions- auf Subsystemanforderungen werden mittels Prädikatenlogik formalisiert und deren Anwendung an einer typischen Automobilfunktion demonstriert.

Contents

1	Introduction	1
1.1	Background	1
1.2	Problem Statement	3
1.3	Solution Outline and Thesis Contributions	4
1.4	Structure of this Thesis	5
2	Fundamentals	7
2.1	Real-time Systems	7
2.1.1	Introduction and Definitions	7
2.1.2	Scheduling Concepts and Strategies	11
2.1.3	Timing Analysis	13
2.1.4	Time-triggered Systems	16
2.1.5	Event-triggered Systems	20
2.2	Automotive Embedded Systems	22
2.2.1	Distributed Embedded Real-time System	22
2.2.2	Automotive Software Engineering	24
2.2.3	Operating Systems and Communication Busses	25
2.3	Model-based Development	27
2.3.1	Overview of System Modeling	27
2.3.2	AUTOSAR	28
2.3.3	Timing-augmented System Model	29
2.4	Constraint Logic Programming	30

3	Basic Concepts Used in this Thesis	33
3.1	Distributed Development of Automotive Systems	33
3.1.1	Process Roles in Distributed Development	33
3.1.2	Collaboration Workflows	35
3.1.3	Role Responsibilities Regarding Timing	36
3.2	Observable Events and Event Chains	36
3.2.1	Events in Real-time System Literature	37
3.2.2	Events in Real-time System Development	37
3.2.3	Definition of Observable Events	37
3.2.4	Definition of Event Chains	39
3.3	System Architecture Model Used in this Thesis	39
3.3.1	System Definition	39
3.3.2	System Architecture Description Overview	40
3.3.3	Functional Architecture Model	44
3.3.4	Logical Software Architecture Model	45
3.3.5	Technical Architecture Model	48
3.3.6	Example System Model	55
3.4	Observable Events of the System Model	58
3.4.1	Observable Events of the Functional Architecture	59
3.4.2	Observable Events of the Logical Architecture	60
3.4.3	Observable Events of the Technical Architecture	61
3.4.4	Example for all Observable Events of a System Model	63
3.5	Function-triggered Timing Constraints	65
3.5.1	Timing Constraints in Real-time Systems Literature	65
3.5.2	Definition of Function-triggered Timing Constraints	69
3.5.3	Function-triggered Timing Constraint Types	70
3.6	Requirements and Guarantees	71
3.7	Summary and Comparison of Timing Constraint Types	73
3.7.1	Summary of Related Work on Timing Constraints	73
3.7.2	Comparison of Constraint Types	74

4	The Global Scheduling Problem	75
4.1	Usage of our System Model	75
4.1.1	Assumptions	75
4.1.2	Timing Properties	77
4.1.3	System Model Example	78
4.2	Timing Constraints	78
4.3	Global Scheduling Problem Description	79
4.3.1	Definition of Global Scheduling	79
4.3.2	Global Scheduling Steps	81
4.4	Task Mapping and Frame Mapping Approaches	83
4.5	Scheduling Approaches for Event-triggered Systems	84
4.5.1	Rate Monotonic Scheduling	84
4.5.2	Response Time Calculation	85
4.5.3	Deadline Monotonic Scheduling	86
4.5.4	Compositional Scheduling Analysis	87
4.5.5	Generating Intermediate Task Deadlines	88
4.6	Scheduling Approaches for Time-triggered Systems	89
4.6.1	Frame Mapping and Scheduling	89
4.6.2	Frame Scheduling	90
4.6.3	Instruction Precedence Scheduling	90
4.6.4	System Scheduling of Tasks and Messages	91
4.6.5	System Scheduling with Frame Mapping	92
4.7	Limitations of Related Work	92
4.7.1	Derivation of Subsystem Constraints	92
4.7.2	Visibility of Timing Properties	93
4.7.3	Consequences of Changes	94
4.7.4	Conclusion of the Related Work Analysis	95
5	TIMEX - A new Timing Model for Distributed Development	97
5.1	Need for a new Timing Model	97

5.2	TIMEX - Timing Extension	100
5.2.1	Timing Extension Definition	100
5.2.2	TIMEX	101
5.3	TIMEX Model Elements Formalization	103
5.3.1	TIMEX Example	103
5.3.2	Hand Over Points	104
5.3.3	Function Timing	106
5.3.4	System Timing	109
5.3.5	TIMEX Example Formalization	112
5.4	Consistency of a TIMEX Model	114
5.4.1	Reasonable Hops	114
5.4.2	Loose Hops	115
5.4.3	Outer Hops	115
5.4.4	Segment Types	116
5.4.5	Segment Sequences	119
5.4.6	Chain Segmentation	121
5.4.7	Chain Constraints	123
5.5	Development Methodology based on TIMEX	123
5.5.1	Define Function Timing	124
5.5.2	Develop Deployment Model	126
5.5.3	Define System Timing	126
5.5.4	Initialize Requirement Types	126
5.5.5	Generate Requirement Values	127
5.5.6	Generate Communication Model	128
5.5.7	Iterative Steps	128
6	Generating Subsystem Requirements in TIMEX with CLP	131
6.1	Requirements and Guarantees in a TIMEX Model	131
6.1.1	Possible Types of Requirements and Guarantees	131
6.1.2	Assigning the Types to Segments and Hops	132

- 6.1.3 Fulfillment of Requirements with Guarantees..... 134
- 6.2 Initializing Requirement Types 136
 - 6.2.1 Initialization Basics..... 136
 - 6.2.2 Initialization for Event-triggered Systems 138
 - 6.2.3 Initialization for Time-triggered Systems 140
 - 6.2.4 Example 143
- 6.3 Generating Requirement Values - General Approach 145
 - 6.3.1 Problem Analysis..... 145
 - 6.3.2 General Algorithm Description 146
 - 6.3.3 Set up Output Model 149
 - 6.3.4 System Timing Invariants 150
- 6.4 Generating Requirement Values for Event-Triggered Systems.. 151
 - 6.4.1 Timing Constraint Fulfillment..... 151
 - 6.4.2 Consider Guarantees - The Shifting Approach..... 152
 - 6.4.3 Search 160
- 6.5 Generating Requirement Values for Time-Triggered Systems .. 160
 - 6.5.1 General Rules for Requirement Values in Time-triggered Systems 160
 - 6.5.2 Timing Constraint Fulfillment..... 162
 - 6.5.3 Consider Guarantees - The Windowing Approach..... 164
 - 6.5.4 Search 175
- 6.6 Generating a Communication Model 175
- 7 Evaluation 177**
 - 7.1 Standardization of the TIMEX Model..... 177
 - 7.2 Application of the TIMEX Methodology 180
 - 7.2.1 Discussion of the Shifting Approach 180
 - 7.2.2 Discussion of the Windowing Approach 184
 - 7.3 Benefits of the TIMEX Methodology 188
 - 7.3.1 Automatically Derive Subsystem Timing Requirements. 188
 - 7.3.2 Systematically React to Local Timing Problems 189

8 Conclusion	191
8.1 Summary	191
8.2 Outlook	192
A Tool Prototype	195
A.1 Tool Overview	195
A.2 Textual TIMEX Model Editor	197
A.3 Timex Visualization	199
A.4 Requirement and Guarantee Value Table Editor	201
References	203

List of Figures

1.1	The increasing complexity of vehicle electrical systems (based on Negele [67]).	2
1.2	Structure of this thesis.	6
2.1	Hard and soft timing constraints and the relation between their time bounds and the system quality (based on Mächtel [63, page 30]).	9
2.2	The basic task state model according to OSEK [70].	10
2.3	Representation of a search tree for three boolean variables A, B, and C.	31
3.1	Roles, collaboration workflows and development contexts in distributed development of automotive real-time systems.	35
3.2	A system, its interface and the environment.	40
3.3	Three abstraction layers of the system architecture model according to Feilkas et al. [30].	42
3.4	Overview of the system architecture model used in this thesis.	43
3.5	The functional architecture of the system model.	44
3.6	The logical software architecture of the system model.	46
3.7	The technical architecture of the system model consists of four sub-models.	49
3.8	The hardware topology model of the system model.	50
3.9	The deployment model of the system model.	51
3.10	The communication model of the system model.	52
3.11	The execution model of the system model.	54

3.12 Visualization of the formally specified system model example. . . 56

3.13 The temporal order of all observable events of the system model example. 63

3.14 Constraint types and their relation according to Saksena [79]. . . 68

3.15 Comparison of timing constraints in this thesis and in related work. 74

4.1 Timing-augmented system model as input and output of global scheduling. 80

4.2 Dependency of the five global scheduling steps. 82

5.1 Observable events as our concept to connect a system model with a timing extension like TIMEX. 102

5.2 Timing extension of the example system model using TIMEX on both function level and implementation level. 104

5.3 Methodology for distributed development of automotive real-time systems based on TIMEX and our system model. . . . 125

6.1 Flowchart representing the algorithm to iteratively generate and modify subsystem requirements according to function-triggered timing constraints. 147

6.2 The input and output of the processing step *consider guarantees*. 148

6.3 Example for horizontal shifting. 154

6.4 Example for vertical shifting on the same resource for an iteration i and the subsequent iteration $i+1$ 155

6.5 Example for diagonal shifting across different resources for an iteration i and the subsequent iteration $i+1$ 156

6.6 Window representation of the TIMEX example. 165

6.7 Unwanted situations of windows: a) gap b) overlap. 166

6.8 Four possible situations how the old required window can relate to its guaranteed window. 167

6.9 Four possible situations how the guaranteed window can relate to its new required window. 168

6.10 All window requirements must be in a way that a period switch does not occur during a window. 169

7.1 AUTOSAR timing model: events and event chains [4]. 178

7.2 AUTOSAR timing model: timing requirements and timing guarantees [4]. 178

7.3 Example with one solution by horizontal shifting. 181

7.4 Example with seven solutions by diagonal shifting. 182

7.5 Example with two solutions by the windowing approach. 185

A.1 Three different model types of the overall TIMEX tooling. 196

A.2 Example function timing in the textual TIMEX editor. 197

A.3 Example system timing in the textual TIMEX editor. 198

A.4 Visualization of the example TIMEX model for an event-triggered system. 199

A.5 Visualization of the example TIMEX model for a time-triggered system. 200

A.6 Structural visualization of the example TIMEX model for a time-triggered system. 200

A.7 Table editor view of the requirements and guarantees of the event-triggered example. 201

A.8 Table editor view of the requirements and guarantees of the time-triggered example. 202

List of Tables

3.1	Categorization of timing constraints in related work.	73
4.1	Frame Packing according to Navet et al. [78, 61].	84
4.2	Rate Monotonic Scheduling by Liu and Layland [59].	85
4.3	Response time calculation based on Rate Monotonic Scheduling, Joseph and Pandya [51].	86
4.4	Deadline Monotonic Scheduling, Audsley et al. [1, 3, 2].	86
4.5	Compositional Scheduling Analysis according to Richter et al. [74, 49].	87
4.6	Generating intermediate task deadlines according to Gerber et al. [37].	88
4.7	Frame scheduling according to Grenier et al. [38].	89
4.8	Message scheduling according to Nossal and Galla [68].	90
4.9	Instruction precedence scheduling according to Chung and Dietz [21].	91
4.10	System scheduling according to Schild and Würtz [84].	91
4.11	System scheduling according to Ding et al. [24].	92
4.12	System scheduling approach addressed by our work.	95
5.1	Subsystems developed by the different roles in distributed development of automotive systems.	98
6.1	Possible types of the first and last segment of a function event chain in the two possible collaboration scenarios.	142
6.2	Complete example for horizontal shifting for an iteration i and the subsequent iteration $i+1$	154

6.3	Complete example for vertical shifting on the same resource for an iteration i and the subsequent iteration $i+1$	156
6.4	Complete example for vertical shifting on different resources for an iteration i and the subsequent iteration $i+1$	157
7.1	The input values of the windowing approach example.	185
7.2	Solution 1 of the windowing approach example.	187
7.3	Solution 2 of the windowing approach example.	187

Introduction

1.1 Background

Today's automobiles include an increasing number of functions that are realized by electronics and especially by software [8, 14]. These functions are typically provided by interactive distributed real-time systems. The development of these vehicle electrical systems is a complex task mainly due to the following five reasons:

1. Software-based car functions are often distributed across the system and can involve several electronic control units (ECUs), sensors, actuators and communication busses for their execution.
2. Each ECU can be involved in the realization of many different functions. This leads to a mutual influence of the functions on each ECU.
3. Subsystems are often developed by different teams and suppliers. The car manufacturer (OEM, *Original Equipment Manufacturer*) must integrate the subsystems to a fully functioning system.
4. The ECUs realize an increasing number of functions. This leads to a higher degree of integration on each ECU.
5. The distributed functions in an automobile's real-time system often have to fulfill stringent timing constraints to function properly.

The main drivers for the increasingly challenging development of automotive systems are depicted in Figure 1.1. In the beginning of software and electronics in cars each function was realized by software that was exclusively executed on one dedicated ECU. This era is shown as *phase I* in Figure 1.1. An example for such an early software-based function is the electronic fuel injection. Later, airbags and automatic door openers followed. In the early 1990s, denoted by *phase II*, more functions and ECUs were introduced in the system. The characteristic of this phase is that for the first time several functions were brought onto one ECU and the functions started to become distributed. This increasing *function connectedness* was driven by the introduction of the CAN bus in cars in the early 1990s (Davis et al. [23]).

Several years ago, the current *phase III* of automotive system development started, which is characterized by the fact that the number of ECUs is not growing further but even tends to decrease. However, new innovative software-based functions still are continuously introduced. This causes a broadening integration gap on each ECU. From the viewpoint of real-time system development, the increasing complexity results in more and more functions that a) have to fulfill timing constraints, b) are developed by different suppliers and c) highly influence each other's timing behavior in the system.

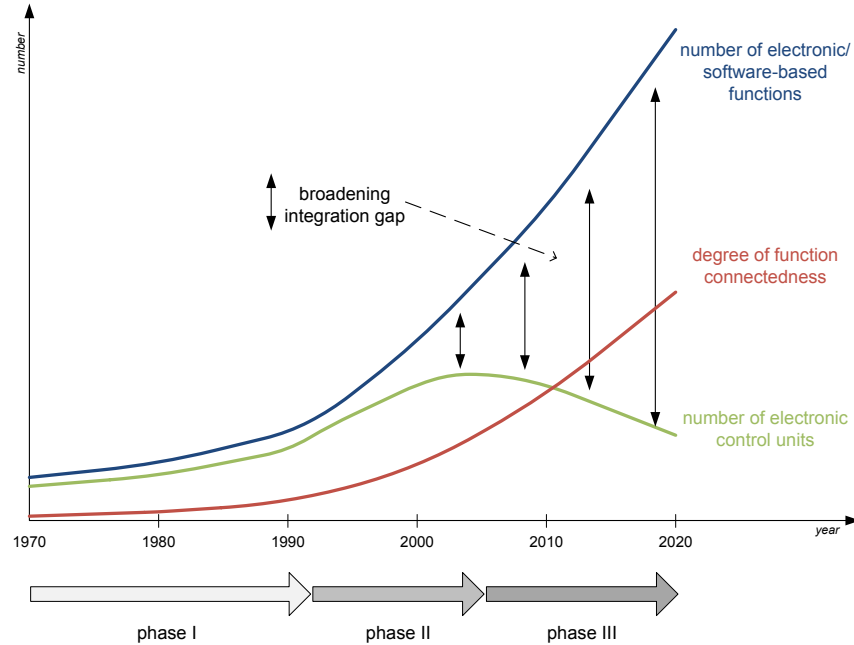


Fig. 1.1. The increasing complexity of vehicle electrical systems (based on Negele [67]).

A continuation of software-based functional innovations, markedly in the regions of safety, comfort and chassis functions, as well as further growth of system design and integration complexity are expected for future automobile generations [14, 47, 13, 87]. The industry as well as the research community is searching for methods and new approaches to cope with the increasing complexity of automotive system design.

A new branch of system design and software engineering has emerged in computer science in the last decade, which is focusing on the challenges of the automotive industry. This new research field called Automotive Software Engineering – see for example Broy et al. [8, 14, 13] or Schäuffele et al. [85] – both brings well-understood methods of traditional software engineering to the automotive domain and develops new solutions especially for the needs that arose with modern automotive system design. Other research groups like Ernst et al. [74, 43, 71] have developed methods for the correct design and analysis of vehicle electrical real-time systems. However, some challenges still remain to be solved as we will explain in the next section.

In the automotive industry a new spirit of collaboration has emerged in recent years. To be prepared for the increasing complexity of automotive embedded systems, major car manufacturers and suppliers founded the AUTOSAR development partnership in the year 2003. Today, many car manufacturers, suppliers, software and hardware companies as well as service providers participate in this partnership [33]. The main goal of the initiative is to define a common development methodology and a standardized software architecture for ECUs with well-defined component interfaces [31]. AUTOSAR allows for a standardized formal description of a system model that consists of application software components, communication, basic software and the mapping of software to ECUs. This information can be exchanged between car manufacturers and suppliers if necessary, i.e. between different development teams.

We identified the lack of a timing model in AUTOSAR in [76]. Later, we presented a proposal for such a timing-augmented system model in [82] and ideas for its application in [83], [72] and [81]. Finally, since its version release 4.0 (Dec. 2009), the desired timing behavior can be described with AUTOSAR and exchanged between development teams [4]. The structural system, ECU and component models can be extended to carry a variety of timing constraints. The remaining prime challenge however is, how timing constraints for teams, or subsystems, can be managed during system design. This is the main problem focus of our work, which we state more precisely in the following section.

1.2 Problem Statement

As outlined in the previous section, a vehicle electrical system is a so-called real-time system, because many of the functions that are realized with the system have to fulfill timing constraints. As these timing constraints characterize a function and are independent of any function realization using hardware and software, we call them function-triggered timing constraints. The realization of one function is often carried out by several application software components on several ECUs that share data over a common communication bus. We call all these components subsystems. Thus, the realization of one function often involves several subsystems in modern automobiles. The functions are therefore often called distributed functions, because they are "distributed" over several subsystems across the system.

Traditionally, and further driven by the AUTOSAR development partnership, the automotive industry is working in a distributed development process. That is, the subsystems are typically developed by suppliers according to the car manufacturer's specification of the subsystem's desired functionality and timing behavior. The car manufacturer has the role of a system designer and system integrator in that distributed development process. The suppliers usually only exchange information with the car manufacturer and not among each other, although they collaboratively develop the same automotive system. Furthermore, the car manufacturer, who has a view on the entire system, has the knowledge of all function-triggered timing constraints and must ensure their fulfillment after all subsystems have been integrated into the overall system. The suppliers only develop and bear for their subsystem and its timing requirements.

As functions often are distributed over several subsystems, different suppliers indirectly collaborate on the development of one function. Function-triggered timing constraints thus can lead to mutual timing dependencies of different suppliers in such a distributed development process.

There are many examples for such timing dependencies between subsystems in both the typical time-triggered and event-triggered networks of automobiles. In a synchronous time-triggered FlexRay [32] network the ECU and bus schedules are tightly coupled if they use the same synchronized time base. A change of the bus schedule influences the data availability for software that is executed on an ECU, which is connected to that network. In an asynchronous event-triggered CAN [50] network all sending ECUs can influence the sending behavior of each other dynamically, because messages with a higher priority delay messages with a lower priority sent by another subsystem. Another example for timing dependencies in both types of networks is the synchronization of the timing behavior of one or more functions. The synchronization of events on different ECUs implies the synchronization of the schedules of different ECUs developed by different suppliers.

Every subsystem developer is responsible for the timing behavior of his particular subsystem. However, as the above explanation clarifies, the subsystem implementations also influence the timing behavior of other subsystems. Thereby they influence the timing behavior of the entire system. Thus, every single subsystem takes part in fulfilling the overall function-triggered timing constraints of the system. As the suppliers of the subsystems do not collaborate with each other, but only with the system designer, the system designer must control the timing behavior of the subsystems by coordinated subsystem timing requirements. These subsystem timing requirements must be derived from the given function-triggered timing constraints of the system.

To tackle this problem, we define a special timing model called TIMEX and an according methodology for distributed development of automotive real-time systems based on function-triggered timing constraints. Furthermore we develop algorithms to iteratively generate timing requirements for the different development teams according to the function-triggered timing constraints.

1.3 Solution Outline and Thesis Contributions

To solve the above-mentioned system integration problem for large automotive systems that are developed by several different teams we first propose a new timing model called TIMEX. TIMEX basically serves for two purposes. First, it is used to capture function-triggered timing constraints in a formal way by a so-called *function timing* model. Function timing models are an implementation-independent formalization of function-triggered timing constraints. When it comes to a system implementation, TIMEX provides a formalism to abstract from low-level timing properties of subsystems to a more abstract level to specify subsystem timing requirements. This is called the *system timing* model. Each timing requirement is accompanied by an according timing guarantee, which is provided by the subsystem developer to the system designer (OEM) as a response to the timing requirement.

The main goal of the TIMEX model is that the fulfillment of the system-wide function-triggered timing constraints shall be ensured on subsystem level. That means the subsystem timing requirements must be fulfilled by their according timing guarantees. Therefore it is important that the timing requirements have been derived from the function-triggered timing constraints such that the later validation can be performed on subsystem level. To enable this, we formalize timing requirement derivation and modification rules by predicate logic, based on constraint logic programming (CLP) techniques.

The timing requirement derivation rules and the rules for iterative timing budget redistribution for unfulfilled requirements differ for the two main system types of today’s automobiles, which are event-triggered and time-triggered systems. Therefore we developed two different formalization approaches, which we also formalized using predicate logic. For event-triggered systems, we developed the *shifting approach*. It basically works by redistributing time budgets iteratively in a TIMEX system timing model by shifting unused time budget either along a data path from one subsystem to another (we call it horizontal shifting), or by shifting unused time budgets on one resource, i. e. an ECU or a communication bus (we call it vertical or diagonal shifting). For time-triggered systems, we developed the *windowing approach*. It is used to define time budgets as time windows according to a TIMEX function timing and move and resize these windows iteratively until all function-triggered timing constraints are fulfilled, according to the fulfillment of subsystem timing requirements by their guarantees. Shifting and window moving are precisely defined reactions to local timing problems that occur when timing requirements of subsystems are not fulfilled by their timing guarantees.

The TIMEX model and its according methodology proposed in this thesis are beneficial for automotive system development due to the following reasons. The model is suited as exchange format between OEMs and suppliers in distributed development. Its abstraction technique enables the sharing of timing constraint and behavior information, without revealing implementation details, which are considered as intellectual property of the suppliers. The reasoning about the fulfillment of function-triggered timing constraints can be performed on this abstraction level by explicitly considering subsystem borders. The iterative subsystem timing requirement – or time budget – derivation, which we formalized by the shifting and windowing approach, allows for a systematic solving of local timing problems of subsystems.

1.4 Structure of this Thesis

Figure 1.2 shows the organization of this thesis as a flow chart. The main topics and contributions are displayed as rectangles that are grouped by the thesis chapters in which they are discussed or presented. Arrows indicate dependencies between these topics.

In Chapter 2 we describe the fundamentals for this thesis that are necessary to understand our work, such as an introduction to the automotive domain and real-time systems. We introduce our main thesis concepts and definitions (e. g. function-triggered timing constraints, our system model) in Chapter 3.

In Chapter 4 we describe related work in the area of embedded real-time systems and analyze its limitations. Our TIMEX model and methodology, which we use to overcome these limitations, are introduced in Chapter 5. We provide an informal description as well as a predicate logic formalization for both the shifting and the windowing approach in Chapter 6. An evaluation of the TIMEX model and methodology is provided in Chapter 7. Finally, we conclude this thesis in Chapter 8.

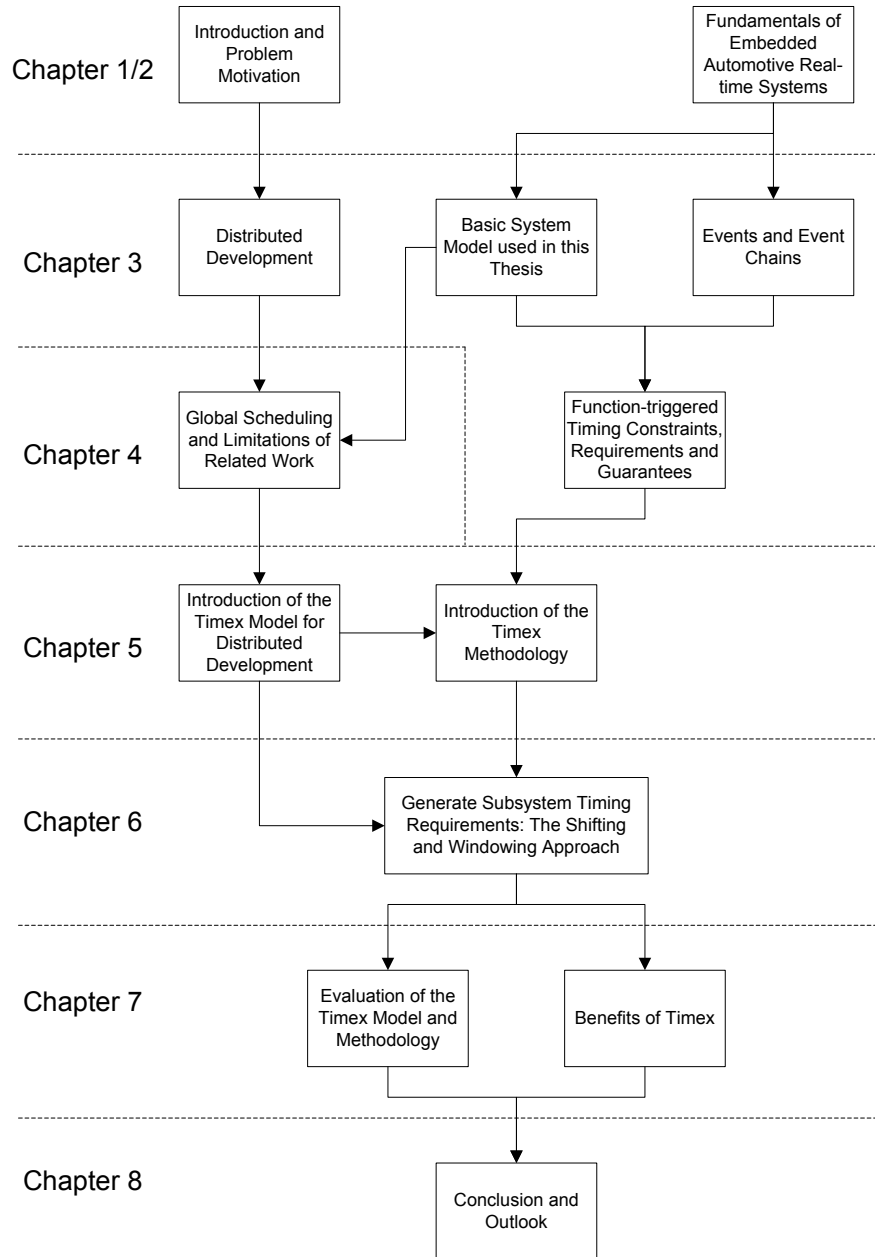


Fig. 1.2. Structure of this thesis.

Fundamentals

This chapter provides the fundamentals of real-time systems theory and practice. Further we introduce the reader to the automotive systems domain and some techniques that we use in our work, such as model-based development and constraint logic programming.

2.1 Real-time Systems

In this section we provide an overview of real-time system theory. We make some basic assumptions and definitions for our thesis and explain the general concepts of such systems.

Throughout this section we make some references to other publications to confirm our statements. However, the reader finds surveys of related work also in other chapters. We give an overview of related work regarding types of timing constraints in Section 3.5.1. Our survey of scheduling and timing analysis approaches can be found in Chapter 4, where we explain the so-called *global scheduling problem*.

2.1.1 Introduction and Definitions

Real-time Systems

The correctness of usual computer systems basically is defined by the correctness of the calculations, or operations that these systems perform. In real-time systems additionally the timeliness of such operations is important for the overall correctness.

Definition 2.1. *A system is called a real-time system if the overall correctness of an operation provided by the system depends not only upon the logical correctness of its operations by providing the correct output values, but also it depends upon the time in which the output is provided.*

Real-time systems can be found in many different areas of our everyday life. The focus of this thesis however is on real-time systems in automobiles. The concepts though are the same for other real-time system domains as well. Examples for other domains are avionics, industrial automation, medical engineering, consumer electronics, and many more. The magnitude of time that is relevant in a real-time system, i. e. the time duration after which some action or reaction is expected, varies and depends on the domain. In vehicle electrical systems the magnitude typically is milliseconds or even microseconds. Other domains could also have seconds, minutes or even days as their temporal magnitude.

Timing Constraints

According to Definition 2.1 real-time systems have to provide correct output values for given input values within certain time bounds. This time bounds can be manifold. We call such time bounds timing constraints.

Definition 2.2. *A timing constraint is a time bound that has to be fulfilled by a real-time system to function correctly.*

To evaluate the correctness of a real-time system, a clear specification of its timing constraints is mandatory.

Many different semantics and levels of abstraction to express timing constraints exist in practice. We provide a comprehensive survey of timing constraints in real-time systems literature in Section 3.5.1.

In literature timing constraints are often divided into two classes, called *hard* and *soft* timing constraints. Again, many definitions of hard and soft timing constraints exist in literature. However, they basically have the same meaning. Liu [60] defines hard and soft timing constraints as follows.

Definition 2.3. *A timing constraint is called hard if it must always be fulfilled by the real-time system. A timing constraint is called soft if it must not necessarily be fulfilled by the real-time system, but its fulfillment increases the system's quality.*

The relation of hard and soft timing constraints and the quality of the system is depicted in Figure 2.1. A hard timing constraint must necessarily fulfill its time bounds within a certain tolerance, indicated by the *min* and *max* values on the time line. When a hard timing constraint is not fulfilled the system's quality drops to null. A soft timing constraint can have a specific influence on the quality, depending on the fulfillment of the time bound.

Hard timing constraints can typically be found in safety-relevant systems, where non-fulfillment can cause severe damage to the system itself, its users or its environment. In a car for example the chassis control can be classified as such kind of system. Soft timing constraints can be found in systems, where non-fulfillment "only" leads to a reduced quality, but no severe consequences. Often, body electronics have soft timing constraints in a car.

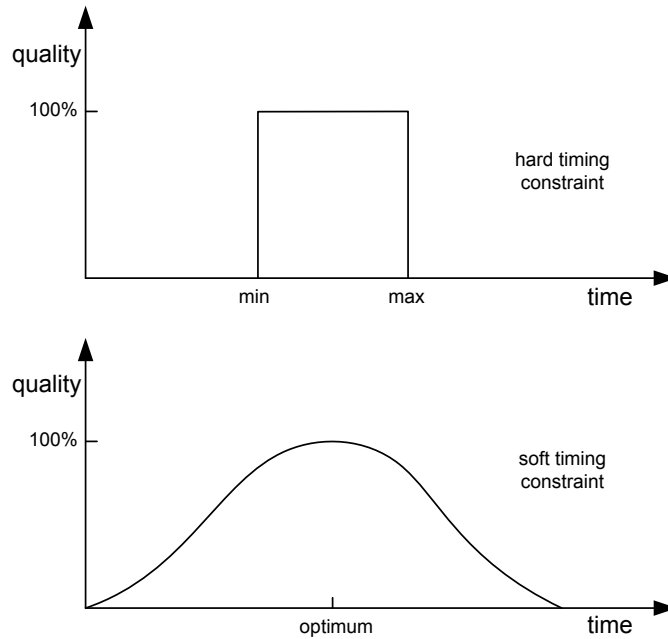


Fig. 2.1. Hard and soft timing constraints and the relation between their time bounds and the system quality (based on Mächtel [63, page 30]).

Assumption 2.1 *In this work we assume that all timing constraints are hard timing constraints. The real-time system, in our case the vehicle electrical system, must always fulfill all timing constraints.*

Tasks and Frames

So far we defined real-time systems rather abstract as systems, which perform operations under certain timing constraints. In the automotive context (as well as other contexts) the system can be divided in two types of subsystems, namely processors and busses. The subsystems share similar concepts but have a different terminology. Basically, in both kinds of subsystems there is a shared resource. The shared resource either is a processor or a communication bus. In both cases special *entities* concurrently compete for the resource.

On a processor these entities are typically called tasks. The processor executes tasks. Thereby they probably have to fulfill timing constraints. A single processor can always only execute just one task at the same time. The automotive real-time operating system standard OSEK [70] defines a standard state model for tasks, which is depicted in Figure 2.2. We will refer to task states in this thesis according to OSEK's standard state model. A task is *suspended* when its execution has finished and no subsequent execution is currently required. When a task is activated it changes its state to *ready*. When it is started, i. e. during its actual execution on the processor, its state is *running*. While a task is preempted as described in Section 2.1.2 its state changes from *running* back to *ready* in the meantime. After termination the task state again is *suspended*.

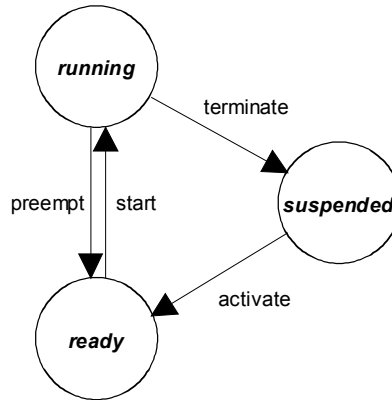


Fig. 2.2. The basic task state model according to OSEK [70].

On a communication bus the entities that concurrently compete for the resource are typically called frames. Frames are transmitted over the communication bus and have one sender and one or more receivers. Again, the transmission probably has to fulfill timing constraints. A communication bus can only transmit one frame at the same time. Basically, the task state model of Figure 2.2 also fits for frames, except that frames cannot be interrupted, or preempted, once they are currently transmitted, i. e. *running* according to OSEK’s terminology for task states.

Scheduling

The concept of task scheduling and frame scheduling in general is not only applied in real-time systems. Rather, scheduling is performed in every computer system, in which several processes compete with each other for a common resource. For example, every personal computer performs scheduling to handle several concurrent applications. In real-time systems though scheduling exceedingly attracts the researcher’s attention, because it highly influences the fulfillment of the system’s timing constraints.

Definition 2.4. *Scheduling is performed by an operating system to sequence the execution of several concurrently activated tasks that are ready to be executed. The processor can execute only one of these tasks. In the context of communication, scheduling is the sequencing of several concurrently active frame transmissions, where just one frame can actually be transmitted at once.*

In a real-time operating system, scheduling is actually carried out by a scheduler, which is part of the operating system. In the case of a real-time communication network, the scheduler is part of the communication controller. The dispatcher is also a part of an operating system. The dispatcher starts and stops tasks that have been chosen by the scheduler to be started or preempted.

2.1.2 Scheduling Concepts and Strategies

In general there exist several concepts and strategies that a scheduler can be based on in its decision algorithm. Some of the described concepts and strategies can be applied to processor scheduling as well as bus scheduling. To emphasize this, we explain each strategy for both resource types, if possible. The list we provide here is not complete but contains the concepts that are important for this thesis.

Time-triggering and Event-triggering

For embedded systems (see Definition 2.2.1), which are discussed in this thesis, there exist two main scheduling strategies. The first one is time-triggering, which we explain in Section 2.1.4. The second and very different strategy is event-triggering, which we explain in Section 2.1.5. Event-triggering and time-triggering are concepts that affect the whole real-time system development, analysis and runtime behavior. All these aspects are investigated in the following sections. For this thesis both strategies are important. Today's automotive functions still rely on both strategies. Some functions fit for event-triggering, others for time-triggering. This leads to heterogeneous systems in cars.

All techniques we investigate for both of these strategies assume that all scheduling entities like tasks and frames are known in advance and static. This assumption is appropriate for vehicle electrical systems as well as other embedded system domains. Other strategies of course also can deal with arbitrary task sets. These so-called online scheduling strategies are typical for PC systems. In Section 2.1.2 we explain an example of such an online scheduling strategy.

So-called hybrid systems combine time-triggered and event-triggered scheduling concepts. For practical applications this is a very convenient way to benefit from the advantages of both system types. Hybrid systems are not considered in this thesis. Our work concentrates on either purely time-triggered or purely event-triggered systems.

Preemption

Scheduling strategies can be divided into preemptive and non-preemptive strategies. In a preemptive scheduling strategy a running task can be interrupted during its execution. This happens, if another task wants to run and thereby would occupy the same resource at the same time and that task is – for whatever reason, mostly due to a higher priority – selected to run by the scheduler. In the non-preemptive case a task finishes its execution once it is running. It can usually not be interrupted by another task.

The reason why the above explanation explicitly focuses on tasks is that the preemption concept only fits for scheduling of tasks on processors. In a communication network usually a frame transmission cannot be interrupted once it was started. That is, frame scheduling by default is non-preemptive.

First in first out

A very simple scheduling strategy for processes is scheduling by the order of the activation of the tasks. In this case the scheduler maintains a simple queue of currently activated tasks that are in the ready state (see Figure 2.2). A newly activated task is added to the end of that queue. When the currently running task finished its execution and terminates, the task on top of the queue is selected to run. This strategy is often referred to as *first in first out* strategy, or short FIFO.

In a communication network the above-mentioned queue usually can only be maintained locally on one connected communication controller. The queue then contains all frames to be sent by this single local controller. A FIFO strategy is only applied to a local frame set by one controller.

Priorities

Priority-based scheduling is a widely used scheduling strategy for both processors and communication networks that follow the event-triggered paradigm. In this strategy, the scheduled entities are characterized by a *priority* attribute. Based on the priorities of all currently activated scheduled entities, the scheduler selects the one with the highest priority and allows it to access the resource. Eventually, an entity with lower priority is displaced, or preempted as described above. If more than one entities with the same priority are active, additionally a FIFO strategy (or any other, for example also random selection) can be applied by the scheduler.

An additional differentiation can be made for priority-based scheduling strategies. The priorities can either be assigned before or during system runtime. In the former case the strategy is referred to as *fixed priority scheduling*. Priority assignment then is engineering work and part of the system development phase. Researchers have studied many priority assignment strategies. We look into some important strategies of that kind in Section 4.5. The latter case is called *dynamic priority assignment* and is performed by the scheduler itself. The scheduler therefore needs some kind of priority assignment algorithm that is executed at runtime. Those are also called *online scheduling algorithms*. This causes additional runtime scheduling overhead compared to fixed priority scheduling.

An example for such an online scheduling algorithm is the *Earliest Deadline First* scheduling (short EDF, see for example [20, 18, 25, 96]). The scheduler in this case always selects the task with the closest completion deadline for execution. The selection is performed every time a task is activated or finishes its execution. As online scheduling is not common in the automotive domain, such strategies are not further mentioned or investigated in this thesis. At this point EDF is solely mentioned as one online scheduling example.

Time Slots

All scheduling concepts and strategies mentioned above are only needed if several scheduling entities want to claim the shared resource at the same time. Time slot scheduling however is a strategy that is used to avoid concurrent execution of tasks or concurrent transmission of communication frames. In this case, all scheduling entities are actually scheduled before runtime, i. e. they are scheduled *statically* instead of *dynamically*. Therefore, so-called time slots are defined for their execution or transmission, which must not be exceeded. In a pure time slot scheduling, the scheduler almost becomes superfluous during runtime. Time slot scheduling requires an absolute time base on the resource to activate and terminate tasks or frames at the right time. The remaining job of the actual scheduler is to monitor that time slots are not exceeded. The task activation or frame transmission in time slot scheduling is done by the dispatcher according to a so-called *schedule table*.

Definition 2.5. *In a time-triggered system a schedule table is used to precisely describe at which point in time according to the absolute time base which action is performed. Actions to be performed are the activation and termination of tasks and the transmission of frames on a bus.*

In practice often additionally priority based scheduling must be performed for entities, which have no assigned time slot. This concept is used for example in hybrid systems, which we already mentioned in Section 2.1.2.

2.1.3 Timing Analysis

Timing Behavior

A real-time system has a specific dynamic behavior that depends on its implementation. The dynamic behavior can be observed at system runtime or predicted using an appropriate model of the system (see Section 2.3).

The entire system implementation in our understanding consists of the following parts:

- the system's concrete software implementation, basically consisting of application software
- the hardware topology that executes the software, consisting of ECUs and busses
- a mapping of the software to the hardware (which processor executes which part of the software organized as tasks)
- and especially the system's timing properties, i. e. all configuration parameters like task schedules and bus schedules

Timing properties are all configuration parameters that influence the scheduling effects of a real-time system. The available timing properties depend on the real-time system type and its scheduling strategy, i. e. event-triggering or time-triggering. During runtime, these timing properties influence the system's so-called *timing behavior*. The timing behavior can be observed at runtime. However, if the real-time system is represented by an appropriate model, its timing behavior can even be determined before runtime based on that model. Timing properties are then captured and collected in a model, usually along with the other system implementation parts listed above. We define such a model as a *timing-augmented system model* in Section 2.3.3.

Besides the functional behavior, especially a system's timing behavior is part of the above-mentioned dynamic behavior. The timing behavior is the most important part in the context of our work.

As mentioned before, a system model can be used to analyze the timing behavior before runtime, i. e. without an actual system implementation. This is done using so-called timing analysis methods.

Definition 2.6. *Timing analysis is the method of determining the expected runtime timing behavior of a real-time system. Timing analysis can be performed using an appropriate model of the system.*

Especially in the context of event-triggered real-time systems timing analysis is often also called *schedulability analysis*.

Note that timing analysis is typically carried out using an appropriate model of the real-time system. The model is analyzed statically, i. e. without actually running the system. The principles of model-based development are discussed in Section 2.3. In this thesis we assume model-based timing analysis methods to be used to determine the timing behavior. As we show in Chapter 4, many such model-based analysis approaches exist in literature.

Other methods to gather and analyze a system's timing behavior are:

- simulation, i. e. actually executing the model of a system and observing the behavior
- runtime testing and monitoring, i. e. executing the real system itself (not a model) and observing the behavior

Both simulation and runtime testing are not considered in this thesis.

According to Definition 2.1, real-time systems usually have to meet timing constraints additionally to other functional requirements. The correctness of the functional behavior is verified using well-established methods known from software engineering, such as software tests. However, for real-time systems especially the fulfillment of its timing constraints is important and must be verified. Timing analysis is a means to timing constraint verification. Usually timing analysis is performed to verify the fulfillment of its timing constraints. Besides that, timing analysis can also be used to determine intermediate timing properties.

An example for such an intermediate timing property is the so-called *worst case response time* (WCRT) of a task or frame (see Section 4.5.2). Another example is the latency – or WCRT – of a data path through the system, like the path from a sensor that produces some data to an actuator.

Timing Analysis Areas

In this section we further refine the term timing analysis to several subareas. According to Montag et al. [65] timing analysis consists of the following subareas:

- Worst case execution time analysis
- ECU schedulability analysis
- System schedulability analysis

The worst case execution time (short WCET) of a task is the time it takes on a specific processor to execute the task without interruption. The determination of a task’s WCET can be performed using different methods of *worst case execution time analysis*. In contrast to simple runtime measuring methods, static WCET analysis promises safe upper bounds. These methods analyze the WCET based on two main factors, namely the task’s internal control flow and the hardware it is executed on. A lot of work exists in this area. For example, Montag et al. [65] present a tool for static analysis based on hardware models of the processors. Tavares et al. [91] present a method to express a WCET independently from a certain hardware and to map it onto a concrete hardware with a concrete WCET later. The WCET is the most basic timing property of a task and the basis of all other timing analysis methods.

Assumption 2.2 *In this work we assume that all worst case execution times are given input and do not change. Worst case execution times are a timing property of tasks or of runnables (runnables are software containers that are mapped to tasks, see Section 3.3).*

The other two areas of timing analysis mentioned above correspond to the classification made by Richter [73]. In his work, these two analysis areas are called component scheduling analysis and system scheduling analysis. They basically have the same meaning as in the definition of Montag et al. [65].

Component analysis focuses on one individually scheduled component, which is either a processor or a communication bus. Montag et al. neglect timing analysis for a communication bus as single component. In the work of Montag et al. , communication-related timing analysis is considered as part of the system timing analysis. According to Richter and other researchers however, component analysis concepts often match for both types of components. The models in use are very similar and just depend on the specific scheduling strategy as discussed in Section 2.1.2.

One basic property of a set of periodic tasks or frames is the *resource utilization*. Liu and Layland [59] define resource utilization as follows. Given a set of n strict periodic tasks, let C_i be the constant WCET of task i and P_i the period of task i . Accordingly, if applied for a set of frames, C_i is the transmission time for frame i and P_i is its period. The resource utilization is defined by Equation (2.1).

$$U = \sum_{i=1}^n C_i/P_i \quad (2.1)$$

The utilization of a component must be lower or equal than 1. Otherwise no schedule can exist that guarantees a successful execution of the task set, because the processor is overloaded, regardless of the execution order of the tasks.

System timing analysis is used to analyze the timing behavior of a complete system that consists of single components, such as processors and busses. System timing analysis introduces some additional challenges. Systems often perform distributed functionality. This leads to complex interferences and dependencies of the timing behavior of the single components. Different approaches to system timing analysis exist. We will look into some approaches in Chapter 4.

2.1.4 Time-triggered Systems

One of the two main paradigms for real-time systems is time-triggering. Comprehensive studies of such systems have been provided by Kopetz et al. [54, 28]. The concepts can be applied to both processors and communication busses. In a pure time-triggered system all actions are triggered solely by the progression of time. Therefore the complete execution and transmission schedules are defined before runtime to perform a time slot scheduling as explained in Section 2.1.2. The resulting schedule table is cyclically processed and used to trigger transmission or execution based on an absolute time line. Triggering cannot be influenced at runtime by the system itself or by any dynamic application behavior.

In a pure time-triggered operating system each task has a pre-defined absolute time window and a period. It is executed strictly cyclic with that period in that time window. The task is guaranteed not to be interrupted during its execution, at least within its assigned time window.

In a time-triggered communication schedule, each frame on the bus also has a fixed period and a transmission time and can only be transmitted by the sending component at these predefined time instances cyclically. All receiving components share the same schedule table and can catch the transmitted data at the correct time.

Scheduling of Time-triggered Systems

As already indicated in Section 2.1.2, scheduling of a time-triggered system as defined in Definition 2.4 is rather trivial. In the case of time-triggering there is no runtime scheduler responsible to make the right decisions. Instead it just has to follow a pre-defined schedule table (see Definition 2.5) that indicates when which task must be activated or terminated. In fact, the work to establish and maintain such a schedule table has to be done before runtime by engineering work. Because of that, the term *scheduling* in the context of time-triggered systems often actually is used to describe the process of creating such a schedule table. This understanding of scheduling differs from the one of Definition 2.4.

Creating a schedule table can be performed manually by an engineer or by using algorithms that compute schedule tables. The process of setting up a schedule table is often called scheduling as described above. Furthermore, *schedule generation* denotes time-triggered scheduling if it is realized by algorithms.

Definition 2.7. *The creation of a schedule table for a time-triggered system using algorithms instead of manual engineering work is called schedule generation.*

A schedule table is often also just called schedule. Such a schedule is a set of tasks in case of a processor schedule, or a set of frames in case of a bus schedule. In both cases each of these entities virtually has an individual *window* for its execution. Each window has the following timing properties that completely describe a time-triggered schedule:

- Each window has a certain **window size**. In case of a processor schedule the window size should be at least the task's WCET to prevent the task from exceeding its execution window even in the worst case. In case of a bus schedule the window size is the maximum frame length. In typical time-triggered communication networks the maximum frame length is a constant. This means all transmission windows have the same size.
- The **period** defines the exact cycle time of every window. A task is executed cyclically with that period. A frame is transmitted cyclically with that period.
- Additionally, each window has an **offset**. This timing property determines the absolute position in time of the execution window. To apply offsets, the already mentioned absolute time base is mandatory for time-triggered systems.

Note that we make the following Assumption 2.3 for the relation of period and offset values.

Assumption 2.3 *We assume that in a time-triggered system the offset of a task, frame, event, or any other timing entity cannot be greater than the entity's period. The offset refers to a time instance within the entity's period.*

A schedule as defined above only lists the timing properties for each window. At runtime a time-triggered system repeats that schedule again and again. Therefore a repetition period must be defined per system. We call this period the overall *system cycle*.

Definition 2.8. *The system cycle is the overall repetition period of a time-triggered system after which the entire schedule is repeated.*

In fact, setting up such a static schedule table before runtime for a time-triggered system – i. e. defining the above-mentioned three timing properties for each window – is not trivial.

There are usually many boundary conditions that must be met by a valid schedule. In the following we list some of the most important conditions for time-triggered schedules:

- Each reserved schedule window appears periodically in any case, because of the system cycle. So, each task must be executed periodically and each frame must be transmitted periodically. The period of a task or frame must be an integer divisor of the system cycle. For example, if the system cycle is 300 milliseconds, a correct task or frame period is 100 ms, which conforms to a divisor of 3.
- Often time-triggered systems have a *base cycle*, which narrows the possible periods for tasks and frames by defining the base cycle as their minimum period. The periods must be an integer multiple of the base cycle.
- Only one task can be executed by a processor and only one frame can be transmitted by the bus at once. That means the schedule must ensure that there are no overlaps of tasks on a processor and no concurrent frame transmissions on the bus. This means that the offsets of all tasks or frames must be chosen correctly.
- The no-overlap-condition also imposes additional limitations on the periods used in a schedule. To have a valid schedule it is not only sufficient that the periods are integer divisors and multiples as already described. Additionally all periods of a schedule must be in a way, that they do not inevitably cause overlaps regardless of the according offsets. For example consider a system cycle of 35 and a base cycle of 1. The periods 5 and 7 that might be used for two tasks are both integer multiple of 1 and integer divisor of 35. Though there exists no offset combination without a clash of two particular instances of two tasks. The periods of all tasks or frames of a schedule must be what we call *harmonic* (see Definition 2.9).
- Static schedules are not really flexible with respect to changes of any schedule properties. That especially means that all task WCETs and frame sizes must already be known and must not vary. Variation can cause the schedule to become invalid, because for example a WCET increase can lead to an overlap with another task. This would imply a redesign of the schedule. The same applies for changed to offsets and periods, as such changes automatically also affect other tasks or frames.

- The processor and bus utilization according to Equation (2.1) must be lower or equal than 1 for every set of periodic tasks and frames on a processor or bus. The higher the utilization is, the more difficult it is to find a valid schedule for the task or frame set.
- Depending on the application, the system and thus the schedules must ensure the fulfillment of timing constraints additionally to all basic conditions mentioned so far. This is the most challenging condition that must be met by a schedule table. How this can be achieved is part of our work and intensively discussed later.

Definition 2.9. *A task or frame set has harmonic periods, if all periods are an integer multiple of the smallest period.*

Despite, or even because of its determinism, time-triggered scheduling has some disadvantages. First, static schedules can cause a waste of processor or bus resource capacities. This can happen due to an oversize execution window for tasks or frames already at design time. The cause for this can be an initially overestimated WCET of tasks, for example. The waste can also happen when a task or frame does not need the complete assigned time window in a specific execution instance at runtime. The wasted resource time cannot be used dynamically, at least in a pure time-triggered system. The second major drawback of time-triggered scheduling is its inflexibility with respect to changes as already indicated in the list above. If a schedule table gets corrupted because it contains overlaps of execution or transmission windows due to a change of a WCET or an offset, the schedule table often requires a complete redesign.

A task's WCET and period usually are given constants. As described above a change of these values during system development can happen in practice. However, when a schedule table is developed, the WCET and period of each task must be known or at least assumed and are input for the schedule generation. The output of schedule generation for a time-triggered system is the offset of each task (or frame), such that a) all boundary conditions and b) additional timing constraints are met.

Setting up a static schedule table that fulfills all conditions and especially all additional timing constraints is known to belong to the class of so-called NP-complete problems, see Garey et al. [35]. Those problems can be solved efficiently only by using heuristics. Nossal and Galla [68] for example presented a genetic algorithm approaches to find a time-triggered bus schedule based on the time-triggered protocol TTP according to Kopetz et al. [55].

Analysis of Time-triggered Systems

The pre-defined schedules of a time-triggered system imply a completely deterministic timing behavior. All scheduling actions and therefore the system's runtime behavior are known in advance, which makes the system predictable. Because of their determinism such systems fit very well for safety-relevant functions in a car, such as chassis functions like steering or breaking.

Timing analysis of time-triggered systems can be done comparatively easy using static analysis. All processor and bus schedules are given and represent the input to timing analysis. Time-triggered systems are characterized by the fact, that the single component schedules are tightly coupled. Because of that, all these component schedules of potentially networked processors and busses together are also called *global system schedule* especially for time-triggered systems. Two cases can be distinguished for such time-triggered global system schedules:

1. The components, i. e. processors and busses, share a common time base and thus have synchronized schedules.
2. Each component maintains its own time base and thus has an unsynchronized schedule.

Timing analysis is used to verify end-to-end timing constraints. Analysis details differ in each of these cases as follows. In case 1, the knowledge of a common and always synchronized time base can be exploited to construct coupled schedules. For example, the producer of data can be scheduled in a way that it finishes its computation right before the transmission of its output to the consumer is scheduled. The transmission in turn is scheduled right before the consumer starts its computation. It is obvious that all task computations and frame transmissions can be scheduled such that the resulting data paths through the system fulfill timing constraints by default. Timing analysis solely must check such path timings statically. We investigated static analysis of time-triggered distributed systems with coupled schedules in our prior work [80] and [82].

The ease of timing analysis of time-triggered systems makes clear, why they fit so well for safety-relevant systems. Timing analysis enables a safe and easy way to obtain a proof of timing constraint fulfillment. This is mandatory for safety-relevant systems, because non-fulfillment cannot be tolerated at all. However there is the drawback of complex schedule table setup and the system's inflexibility with respect to changes of schedule basics like WCETs, periods and offsets.

2.1.5 Event-triggered Systems

Despite their determinism, which is great for safety-relevant applications, pure time-triggered systems do not fit for all functions and subsystems of a car due to the mentioned disadvantages. Ringler [75] and Kopetz [53, 54] see time-triggered system as ideal for all automotive functions. However, in practice still also event-triggered systems are used due to the existence of functions with a clear "event-like character", for example in the area of body electronics of a car. Furthermore, the use of time-triggered systems tends to result in higher costs per unit. One reason for this is that resources are potentially not fully utilized at runtime because of the inflexible static schedule, which often leaves space between tasks or frames (see Section 2.1.4). This leads to a potentially over-sized and more expensive hardware. The high quantity of units in the automotive domain however demands low unit costs, which

cannot be achieved by using only time-triggered systems. The existence of event-triggered systems is therefore expected for future cars as well.

In pure event-triggered systems all actions are triggered by events. In contrast to time-triggered systems, a static schedule is not part of the implementation. Scheduling is done dynamically at runtime by the scheduler. Thereby some of the different scheduling strategies that are discussed in Section 2.1.2 can be applied. The task scheduling concept in event-triggered automotive systems typically has fixed priorities and allows preemption. Therefore it is often called FPPS (Fixed-Priority Preemptive Scheduling).

On an event-triggered communication bus the frames compete for the common bus resource. Multiple-access and arbitration methods are used to solve conflict situations. On an event-triggered operating system, concurrent tasks compete for the common processor resource. We introduce the term scheduling entity for tasks and frames together in Section 2.1.1.

Scheduling of Event-triggered Systems

Similar to time-triggered systems, the term scheduling here does not refer to the action performed by the actual scheduler at runtime. Rather it means the configuration of a real-time system. In the context of FPPS it basically means the assignment of priorities to the scheduling entities. Depending on the task or frame model, the entities might also need a period to be assigned.

Definition 2.10. *The assignment of priorities for tasks or frames of a system is called scheduling in the context of event-triggering.*

The priority assignment can be performed following certain rules. Researchers have developed several such assignment strategies in the last decades. The first such strategy was Rate Monotonic Scheduling (RMS) by Liu and Layland [59], which we describe in Section 4.5.1. According to RMS, priorities are assigned according to the arrival rate of the tasks. Another approach is Deadline Monotonic Scheduling (DMS) by Audsley et al. [1, 3, 2], where priorities are assigned according to the criticality of the task's deadlines. We describe DMS in Section 4.5.3.

Solely assigning priorities and other timing properties does not ensure the fulfillment of timing constraints, especially in the context of an entire system. Therefore timing analysis is used. We outline timing analysis for event-triggered systems in the following section.

Analysis of Event-triggered Systems

For time-triggered systems we already explained that the main effort in development must be spent on finding a valid static schedule. The analysis of the resulting system behavior is rather simple, as no complex runtime effects must be taken into account. At runtime the schedule is processed exactly as statically defined.

The analysis of event-triggered systems however takes a lot more effort compared to time-triggered systems. Given an event-triggered schedule, i. e. a set of tasks or frames with their according timing properties like WCET and priority, analysis methods are used to investigate runtime effects and to verify the fulfillment of timing constraints. A given schedule is often called *schedulable*, if it can be executed and fulfills all timing constraints. Such analysis methods are therefore also called *schedulability analysis* in event-triggered systems. Schedulability analysis can be applied to a component (processor or task) or a whole system, according to the categorization made in Section 2.1.3. We investigate some of the most prominent analysis methods for event-triggered systems separately in Section 4.5.

All analysis methods assume a certain task or frame model. This model especially must include information about the actual triggering of the tasks and frames. A periodic model, for example, assumes periodic triggering of the according entities. The triggering information is essential for the analysis process, because otherwise no statements can be made, how often a certain entity is interrupted by other higher priority entities. If the system does not provide any information about the triggering then assumptions must be made. In an automotive system, for example, a source of unpredictable triggering behavior is a button that can be pressed by the user or any other external interrupt. The schedulability analysis methods however abstract these practical limitations by assuming a certain triggering.

2.2 Automotive Embedded Systems

2.2.1 Distributed Embedded Real-time System

The functions of a modern car are realized using software and hardware that make up the so-called vehicle electrical system. Often the functions have to fulfill timing constraints. Thus, the resulting system can be characterized as a distributed embedded real-time system. In contrast to usual computer systems, like PCs and other business computers, an embedded system is designed to perform just one or a few dedicated functions. Its hardware and software are embedded as one complete device, which often does not even have a user interface or it has a user interface with only a very dedicated purpose (for example some special buttons).

The system is called distributed, because it consists of several autonomous computing units that are coupled to communicate with each other to provide a certain distributed functionality, i. e. functionality that is enabled by the interaction of these computing units [90]. The computing units are called Electronic Control Unit, or short ECU. The distributed character of modern vehicle electrical systems enables innovative new functions. However this also implies several additional challenges for the development of such systems. Some challenges are mentioned in Section 2.2.2. This thesis in particular covers one of these challenges, which is the distributed development of such real-time systems.

Embedded systems generally are so-called reactive or interactive systems. Harel and Pnueli [44] define reactive systems as computer systems that continuously react to their environment. Thereby they are triggered by their environment. In contrast, interactive systems also continuously interact with their environment, but the speed of interaction, i. e. the interaction rate, is defined by the system and not dictated by the environment. Both system types however share one important fact: they both interact with their environment. Therefore such systems have sensors and actuators. Both of these component types represent the system's border to its environment.

Halbwachs [42] names some very important main features of reactive systems, however these features indeed fit for interactive systems as well:

- Reactive systems involve concurrency. First, there is a concurrency between the system and its environment. Second, the systems often consist of a set of parallel cooperating components.
- Reactive systems have to fulfill strict timing constraints, typically both on their input rate and on their input/output response time.
- Reactive systems are deterministic, because their outputs depend on the given input values and the input values' occurrence times.
- Reactive systems are made by software and by hardware.

All hardware and software components perform different tasks. We distinguish the following three main types of such components, which are integrated to the system as part of an ECU:

- *Sensors* measure a physical quantity to collect some information from their environment and convert it into an electrical signal. The signal can be read by an electronic system and thus becomes available inside the system. In a car, sensors typically are used to measure temperature, location, or speed, or to recognize the activation of a button.
- *Actuators* are devices for moving or controlling a mechanism. Actuators convert energy into mechanical work. Using actuators, a vehicle electrical system can influence its environment. In a car, actuators typically move an electric motor or activate light.
- *Controllers* perform the actual control algorithms of functions. They usually take sensor data and calculate according output values. Typically controllers are responsible for the control of actuators, depending on certain input data.
- *Bus networks* are used to realize communication between different electronic control units that are connected to the same communication bus.
- *Gateways* are used to interconnect different sub-networks. The networks may have different communication protocols. Gateways must therefore translate the protocols.

All busses, sensors, actuators and ECUs together are the overall vehicle electrical system. This system, or rather the action and interaction of all its components, provides all distributed functions of the car.

The overall vehicle electrical system often is divided into several so-called *clusters*. A cluster basically is one bus and the ECUs connected to it. Gateways are used to connect clusters. Gateways forward frames from one cluster network to another. Often clusters are dedicated to a specific *system domain*. Domains group automotive functions by their scope. Historically, the following five domains exist in cars (based on Simonot-Lion and Trinquet [87]).

- The **Chassis Domain** contains functions related to the wheels and dampers of the car. These functions control the wheels' position and movement, e. g. by steering, braking and controlling the dampers.
- The **Power Train** domain contains functions related to the longitudinal propulsion. It basically includes the engine and transmission control.
- The **Body Domain** includes functions and subsystems, which do not belong to the vehicle dynamics, but that support the driver, e. g. airbag, wiper, lighting or the door opener.
- The **HMI Domain** includes all systems and functions, which enable information exchange between the car and the car driver. Therefore, displays, buttons and switches are used.
- The **Telematic Domain** covers components, which allow information exchange between the car and the outside world.

In the following section we will have a closer look on the software development for such automotive real-time systems.

2.2.2 Automotive Software Engineering

Until the 1970s cars were primarily driven by mechanics. During the last 30 years more and more software and electronics were used in cars. Until today the amount of software has been growing exponentially, and this trend is expected to continue for the next 20 years at least [8]. Decreasing hardware costs and especially the demand for new innovative functions with every new car generation are the main drivers of this trend. In fact, software is considered as the most essential driver for innovations in modern cars. Software is used to both realize new functions and replace traditional realizations of functions or function parts that used to be based on mechanics or electronic circuits. Furthermore, software-based functions can easily be combined and correlated to again create new functional innovations. Premium cars have up to 80 controllers that are connected by up to five different bus systems [9]. Many examples for such innovations exist in modern cars, like park assistants or active front steering.

The importance of software for the automotive industry increased significantly over the past decade. As a result, a dedicated computer science discipline

called *Automotive Software Engineering* has emerged (see for example [85] or [14]). Researchers are eager to adopt existing development methods to the car industry and even develop new individual solutions due to its specific engineering requirements. However, still many challenges remain for software development in automotive domain. The most important challenges are summarized by Broy [8] in a comprehensive survey. According to this survey, research challenges from a computer scientist’s viewpoint are found in the development of the system’s architecture, reduction of the overall complexity, improvement of development processes and tools, standardization, and seamless model-driven development.

This thesis covers some of these typical automotive-specific development challenges. The automotive industry is characterized by distributed development [8]. Car manufacturers and different types of suppliers cooperate and collaboratively develop the entire system. The approach presented in this thesis focuses on this fact with respect to the development of real-time systems. The methods presented are entirely based on *model-driven concepts* and also use *standards* that are already well established, like for example AUTOSAR [17, 33].

2.2.3 Operating Systems and Communication Busses

In this section we shortly introduce typical real-time systems and communication busses used in cars today. Our methods are built up on the basic principles of these systems. A general overview of typical automotive systems, especially the communication techniques that are used today, was presented by Navet et al. [66].

OSEK and AUTOSAR OS

In the early phase of vehicle electrical systems several car manufacturers and suppliers developed a standard for real-time embedded operating systems in cars. The standard, called OSEK [70], is very well established today and different compliant products exist. With the raise of time-triggered systems in cars, also a time-triggered OSEK specification was released [69]. The basic task model of an event-triggered OSEK is shown in Figure 2.2.

The industry standard AUTOSAR (see Section 2.3.2) also includes an operating system standard specification, called AUTOSAR OS [6]. The standard basically matches the OSEK specification and also supports time- and event-triggering (see Section 2.1.2). The event-triggered system basically is a typical fixed priority preemptive scheduling (FPPS) system. A time-triggered AUTOSAR OS monitors the execution times of tasks to ensure that they do not exceed their limits. The task models of real-time systems literature can widely be mapped to AUTOSAR OS.

Assumption 2.4 *In our work, we assume an AUTOSAR OS compliant operating system, either purely time-triggered or purely event-triggered.*

Controller Area Network

The CAN bus (Controller Area Network [50]) was developed in the 1980s by Bosch and came into cars in the early 1990s. At this time the amount of interconnected controllers that share data in cars has reached a level that raised the demand for a common communication system between them. CAN uses a CSMA/CR arbitration. Each frame has a unique integer identifier, exact one sender and an arbitrary number of receivers. Identifiers are used both for frame addressing, i. e. configuring sender and receiver relations, and frame prioritization.

According to the scheduling strategies mentioned in Section 2.1.2, CAN follows a fixed priority non-preemptive scheduling. Each connected node follows the arbitration process of CAN before sending a frame. The process is designed to negotiate the node that is allowed to transmit, because its current frame to send has the highest priority. Of course, this scheduling strategy can lead to several typical effects regarding the dynamic timing behavior at system runtime. Most notably message transmission is not deterministic. The sending delay caused by the scheduling strategy depends on the sending behavior of other connected controllers at runtime. This non-determinism basically has two consequences. CAN is typically used for event-triggered applications, like the body domain in a car, which contains many user functions with lower timing criticality. Or, if used in safety-relevant domains, assumptions about the sending behavior are necessary that enable upper bounds of such scheduling delays. Today a lot of work already exists where researchers analyzed the timing behavior of CAN networks, an overview is given by Davis et al. [23]. We will review some of these approaches in Section 4.5.

Assumption 2.5 *In our work, we assume a CAN network for event-triggered clusters in cars.*

FlexRay

FlexRay is a modern bus system for in-vehicle communication [32]. Several car manufacturers and semiconductor companies initiated its development in 2000. The main differences to CAN networks are a higher bandwidth and especially the support for a time-triggered protocol. FlexRay implements the time slot scheduling strategy mentioned in Section 2.1.2. The cyclically applied schedule table for frame transmissions is called a *FlexRay cycle*. The cycle is divided in a static segment that is used for strict time-triggered frame transmission, and a dynamic segment for event-triggered frame transmission. Both segments are used to find a trade-off between an event-triggered network's flexibility and a time-triggered network's determinism. Static and dynamic segment sizes can be freely configured. Even complete time- or event-triggering is possible.

ECUs that are connected to a FlexRay network can synchronize their internal clocks with the FlexRay clock. This leads to a common, global time base and enables FlexRay's tight coupling of ECU and communication schedules.

Assumption 2.6 *In our work, we assume a FlexRay network for time-triggered clusters in cars. Furthermore, we assume a completely time-triggered configuration with no dynamic segment.*

We make Assumption 2.6 for simplicity reasons and thus assume to have the whole bus time available for statically scheduled frames. Considering also the dynamic segment for our approach solely would mean not to have the whole bus time available, because the other part is reserved for the dynamic segment, according to the FlexRay protocol specification [32]. Thus the assumption is no limitation for our approach.

2.3 Model-based Development

2.3.1 Overview of System Modeling

Model-based development is a modern approach for efficient software and systems engineering. The basic idea is to use so-called models as main development artifacts instead of the actual code of the software. This idea can be applied in all phases of the development process. In an ideal model-based development the code is finally generated automatically using appropriate code generators that take models as their input. Models are artifacts on a higher level of abstraction compared to code. Models can be designed to fit for the needs of different engineering viewpoints individually, such as the software architecture, the system dynamics, system component structure, or the system's timing.

In general, model-based development is a promising approach to increase both the quality of software systems and the development productivity by a) raising the level of abstraction during development, b) enabling easy integration of automation and code generation, and c) coupling system design and system implementation on a formal basis.

The work in our thesis also assumes a model-based development approach. That is, the automotive system under development is assumed to be available as an appropriate model instead of its actual hardware and software, i. e. code, realization. Component-based system models are widely used in software and systems engineering. Broy et al. [11, 15, 13] developed a component model, which we will investigate in Section 3.3, where we also introduce our system model used in this thesis. In the automotive industry, AUTOSAR has been developed as a standardized component model, which we briefly outline in Section 2.3.2. Our system model is a subset of AUTOSAR that contains all model elements that are necessary for our approach.

AUTOSAR is a component model developed especially for the automotive industry. In contrast to other system models, however, AUTOSAR does not contain a model of the system behavior. It covers the description of the system's static software architecture by means of communicating components. The dynamics of the internal behavior of these components is not part of the model.

Other models that do include behavior exist and are common also in the automotive industry, especially for the development of control functions. Examples for such models are ASCET [7] or Simulink [62, 52]. As described above, these models are used to generate the actual target code for the control function based on mathematical models.

2.3.2 AUTOSAR

Methodology and Concepts

Major car makers and tier-1 suppliers founded the AUTOSAR development partnership [46, 31, 33] in the year 2003. Today many automotive OEMs, suppliers as well as software and hardware developers and service providers participate in a global partnership. The main goal of the initiative is to define a methodology that supports a distributed development process and to create a standard for the software architecture of automobile ECUs and entire vehicle electrical systems. The standardized architecture includes several structural concepts on different layers, such as:

- application software components,
- basic software, including communication, sensor/actuator access, operating system services etc.,
- a runtime environment that connects application and basic software,
- standardized interfaces between these components.

By using a standardized formal specification model for the structure (not the behavior) of automotive software that all development participants have committed to, AUTOSAR brings several benefits to the industry with respect to the increasing systems development complexity. Some of the proposed benefits of the standard are:

- smooth integration of third party software
- smooth integration of supplier subsystems
- easier reuse of software and hardware components
- seamless application of diverse development tools based on a common system model

As stated above, in this work we present an own dedicated system model in Section 3.3. It can be seen as a simple subset mainly of the AUTOSAR application software component layer and communication stack.

Timing Extensions

When AUTOSAR started to be applied by the industry in its first releases, the specification of the system’s timing constraints was not yet covered. However, to fulfill some of the main requirements of AUTOSAR [5], such as the *Protection of Timing Requirements*, timing needs to be addressed as well. Therefore, we proposed a proper extension of the standard in [82], based on the concepts of so-called *observable events* and *event chains*. We explain these concepts in Section 3.2. Since Release 4.0 AUTOSAR supports the specification of timing constraints, which is realized as so-called *Timing Extension* (see Definition 5.1) for a given AUTOSAR system model [4].

The timing model of AUTOSAR is very generic. It enables the specification of timing constraints for the five main so-called *views* of AUTOSAR:

- Software Component Timing: timing constraints for one software component type
- VFB (Virtual Function Bus) Timing: timing constraints for a network of software components without mapping to ECUs
- System Timing: timing constraints for a concrete system
- ECU Timing: timing constraints for one ECU in a system
- Basic Software Timing: timing constraints for a basic software module

For each of these five views a certain set of different timing constraint types can be used to describe the desired timing behavior of the targeted model elements. The whole timing model is very generic and leaves a lot of modeling freedom, especially with respect to the use and refinement of event chains. In this thesis we present a special timing methodology for distributed development that makes use of the generic timing model of AUTOSAR and we present a more concrete timing model, which we call TIMEX. Both TIMEX and the methodology are explained in Chapter 5.

2.3.3 Timing-augmented System Model

System models are always created for a dedicated purpose. That means they abstract system details that are not in the scope of the system model user and highlight details that are of special interest. Throughout this thesis we often use the term *timing-augmented system model*, which we introduced in [82].

Definition 2.11. *A timing-augmented system model is a model of a system that additionally includes all information that is necessary to specify, analyze, and verify the system’s correct timing behavior. This additional information is called the system’s timing information.*

We divide timing information into the two groups *timing properties* and *timing constraints*. Timing properties define the timing behavior of the system at runtime. Timing constraints constrain this behavior to certain bounds.

2.4 Constraint Logic Programming

In Chapter 6 we explain our approach to generate and iteratively modify timing requirement values for subsystems of an automotive network based on system-wide timing constraints (we call them *function-triggered timing constraints*, Section 3.5). The approach is formalized by predicate logic and implemented by constraint logic programming techniques. Therefore we briefly outline some basics of constraint logic programming in this section.

Constraint programming is a programming paradigm. Relations between variables are expressed by the use of so-called constraints over these variables. That is, the value of one variable is bound to the values of other variables. In contrast to the imperative programming paradigm of standard programming languages, where a sequence of steps describes how the way to a solution looks like, constraint programming describes how a solution looks like. Constraint programming is a form of declarative programming.

Different kinds of constraints can be used to describe solutions. Examples for such kinds are mathematical equations (like $X = Y + 3$) or predicate logic (like $A \Rightarrow B$ and C). The constraint programming paradigm and logic programming share several conceptual similarities and features like logical variables and backtracking. Therefore constraint programming was first embedded into Prolog, which is a logic programming language. Constraints are expressed as logical formulas in Prolog as host language, for which reason it is called *Constraint Logic Programming* in this context, or short CLP.

Constraint variables can be of several typical domains, such as boolean (only true and false values), integer (natural number values), linear (linear functions are described and analyzed) or finite (finite sets of values) domain.

Today there are several implementations of Prolog-based constraint logic languages. One very popular such language is ECLiPSe [19], which we use in our work. We build up a system of variables with an integer domain to formulate our timing problem and use a constraint solver provided by ECLiPSe to search for viable solutions.

In general, a CLP program contains the following three steps (see for example [27, 19]):

1. The variables and their domains are set up. This spans the entire space of possible solutions. Every combination of possible instantiations of variables with values of their domains is one possible solution.
2. A set of constraints over these variables is formulated. The formulation expresses the concrete problem model and thus narrows the space of possible solutions to the ones that fit the solution description, modeled by the constraints.
3. The search for one or more solutions is performed. A constraint solver is used for this purpose, which implements a specific search technique.

A constraint solver uses propagation to reduce the search space during search. Propagation means that additional knowledge of the domains of all variables is gained recursively by considering the constraints. The knowledge is used to narrow the domains by removing values that cannot be part of a solution. By backtracking, the constraint solver tries to iteratively find sub-solutions, and if one sub-solution cannot be part of the whole solution, the sub-solution and all depending sub-solutions are removed by "backtracking" to a sub-solution that did not contain invalid values.

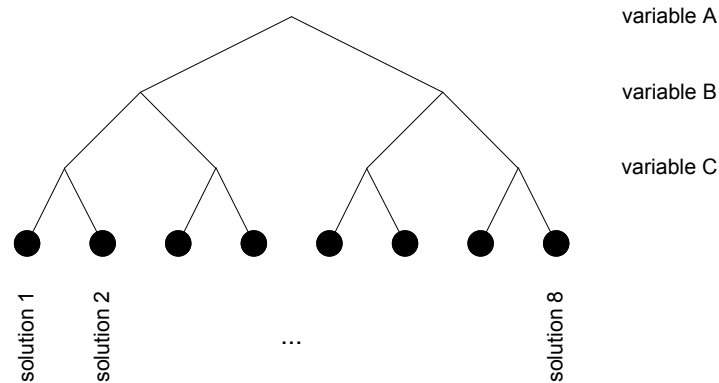


Fig. 2.3. Representation of a search tree for three boolean variables A, B, and C.

Different search strategies can be applied by constraint solvers. A common strategy is tree-based search. In this case the solution space is represented as a so-called *search tree* [19]. Backtracking in this case represents a depth-first search in this search tree. Figure 2.3 shows a search tree of a problem that consists of three boolean variables. Each level of the tree represents the possible instantiations of one variable. We use tree-based search over integer variables in our constraint solver, what we explain in Chapter 7.

Basic Concepts Used in this Thesis

In this chapter we introduce all basic concepts that are used throughout this thesis. These basic concepts are:

- main roles in an automotive system design process
- collaboration workflows in distributed development
- the idea of observable events and event chains
- the system model that is used in this thesis
- the idea of function-triggered timing constraints
- our concept of timing requirements and timing guarantees

If appropriate, we give an overview of related work for each of the concepts, and show in what way our ideas and models are different.

3.1 Distributed Development of Automotive Systems

3.1.1 Process Roles in Distributed Development

The automotive industry frequently follows a distributed development process. Broy et al. [13] call the automotive industry a complex ecosystem of highly interdependent organizations which collaborate in all phases of the vehicle life cycle. This work focuses on the development phase of a vehicle. Many different teams develop the overall vehicle electrical system. The teams have different roles in the process and therefore bear responsibility for different parts of the system and different phases of the project. We identified three main roles that are important for our proposed new design methodology, which we describe in Section 5.5. In fact, this classification is a simplification compared to actual development processes in the automotive domain. Depending on the respective definition, there can also be more or different role definitions and many people actually may represent each role. Our classification however is appropriate in the context of this thesis.

System Designer

The system designer designs, specifies and finally integrates the vehicle electrical system. The system designer is typically an OEM or first-tier supplier. The specification of the entire system comprises the parts that are itemized in the following list. Note that before such a specification can be developed, the functionality of the system must be defined. The list contains the specification of the implementation of the system by means of ECUs, networks and software.

- the system's network consisting of sensors, controllers, actuators, and communication busses
- the software architecture consisting of communicating software components as part of the implementation of all desired car functions
- the mapping of software components to ECUs
- the bus configuration, i. e. communication matrix and bus scheduling

The system designer is responsible for the fulfillment of the system's functional and non-functional requirements. In the context of this thesis we are especially interested in the system's correct timing behavior, which has to be ensured by the system designer.

ECU Developer

Each ECU is actually developed by an ECU developer. This role typically is a first-tier supplier. An ECU developer integrates the entire software of an ECU according to the system designer's specification. According to AUTOSAR the entire software consists of application software components and basic software, as described in Section 2.3.2. We focus on the application software part and neglect basic software other than communication specific basic software. The ECU developer supplies the ECU to the system designer, who integrates it into the rest of the system.

Software Component Supplier

As one of AUTOSAR's proposed benefits, software components may be delivered by third party software component suppliers. This is a rather new role in the automotive industry. Such software components often realize functionality that is not necessarily bound to specific hardware. This enables a free mapping of the component to an ECU which in turn means, it can potentially be developed by a third party company. An (application) software component developed by this role is integrated into the ECU of an ECU developer.

3.1.2 Collaboration Workflows

It is the system designer’s task to integrate all delivered ECUs to a functioning system. It is the ECU developer’s task to integrate all software components on an ECU according to the ECU specification. To avoid inconsistencies and misunderstandings in the collaboration between the three roles, a common specification format is required. Therefore AUTOSAR offers its standardized software architecture and its standardized exchange format [33]. Since release 4.0 also timing constraints can directly be added to the specification [4].

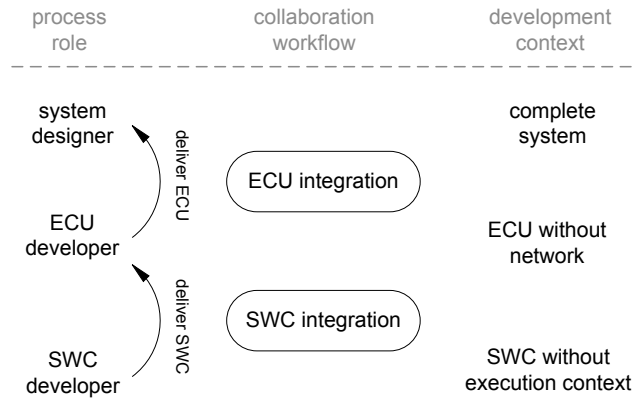


Fig. 3.1. Roles, collaboration workflows and development contexts in distributed development of automotive real-time systems.

Based on the three roles defined above we can identify two typical collaboration workflows in the automotive domain. The three roles and two collaboration workflows are depicted in Figure 3.1.

ECU Integration

ECU integration is the integration of one ECU into a vehicle electrical system. System designer and ECU developer have to collaborate in this use case.

Software Component Integration

Software component integration is rather new for the automotive industry and mainly enabled by the AUTOSAR approach. The standardized software component interfaces enables the integration of third-party software components into an ECU . Therefore, system and ECU developers have to collaborate with the software component supplier.

3.1.3 Role Responsibilities Regarding Timing

In Section 2.1 we described how various implementation properties influence the timing behavior of a real-time system. This behavior must always lead to the fulfillment of a system’s timing constraints. Each timing property is controlled by one of the defined roles, which we say is *responsible* for that property. In the following we describe the responsibility assignment for roles and timing properties.

The system designer is responsible for the whole system. This role must also be aware of the system’s function-triggered timing constraints (see Section 3.5). In our approach these constraints are considered as the basis of system development from a real-time viewpoint. The system designer chooses the network of communicating software components, the hardware topology (ECUs and communication busses) and the mapping of software components to ECUs. The mapping is one of the most important design decisions with influence on the timing behavior. The system designer is also responsible for the bus design. Bus design covers the mapping of signals to bus frames, often called frame packing, and the frame scheduling (Section 2.1.1).

The ECU developer configures the basic software and application software components on an ECU according to the specification. In our work we focus on application software. Timing constraints for basic software modules are not taken into account. Software components, or more precisely the runnable entities in software components, have to be mapped to operating system tasks with an appropriate task schedule. The schedule as implementation property of course highly influences the timing behavior.

The software component supplier cannot directly influence the runtime timing behavior of his software component as it is integrated by the ECU developer and executed at runtime. Typical timing effects at runtime depend on the integration context on the target ECU. However, the software component supplier can provide component-based timing properties like execution times, required execution periods or execution order constraints to the ECU developer.

3.2 Observable Events and Event Chains

Our timing specification approach is based on the concept of so-called observable events and event chains, sometimes also called timing chains. In this thesis we use the term *event chain*. We already mentioned observable events and event chains in previous publications [82, 83]. The industry standard AUTOSAR adopted the concepts in its current Release 4.0 [4]. In this section we provide a clear definition of observable events and event chains.

Events have been studied and used all across timing analysis literature and development since its very beginning. However, the common understanding of the term *event* differs. In this section we explain the difference of the semantics of events in a) standard real-time system theory in literature, b) practical real-time system development and implementation, and c) our definition of observable events.

3.2.1 Events in Real-time System Literature

In real-time system theory, event models are a widely-used concept to describe the occurrence behavior of events. One single *event occurrence* is often also just called event. The concrete occurrence behavior leads to a so-called *event stream* (see for example Gresser [39]). The available event models use different parameters to describe event streams.

A simple example is a strict periodic event model that can be described by using one parameter called the event period. A possible extension to this periodic event model can be achieved by adding an additional parameter jitter that allows a standard deviation from the strict period. Such periodic event models are the basis for many theories, for example Rate Monotonic Scheduling by Liu and Layland [59] and Deadline Monotonic Scheduling by Audsley et al. [3].

Other typical event models are sporadic events and burst events. The model of a sporadic event as described by Sprunt et al. [88] only contains a so-called minimum inter-arrival time. More complex streams descriptions of sporadic events can be modeled using the event stream concept of Gresser [39]. Gresser defines an event stream as a set of event tuples. An event tuple contains two parameters. First the distance of an event occurrence from time point zero, second the interval of the subsequent occurrences. Thiele et al. [92] propose a concept called real-time calculus. It is based on arrival curves of events. For a given time interval, such arrival curves model upper and lower bounds for the number of event occurrences. Finally, Richter et al. [74] developed a method to couple event model streams using event model interfaces. They define a basic set of supported event models. Communicating components are coupled strictly using these models. The coupling enables a new approach to system level timing analysis [49].

3.2.2 Events in Real-time System Development

In real-time system development and implementation, an event often is understood as a programmatic means to actively trigger some kind of system action at runtime. A real-time operating system can use events to trigger some execution or to communicate messages or control between operating system execution entities, like tasks. In an AUTOSAR system for example, there exists a predefined set of so-called *timing events* that can trigger the execution of runnable entities. These events are part of the implementation of the runtime environment.

3.2.3 Definition of Observable Events

We first mentioned observable events in [82] as a basis for systematic timing analysis and explained that a well-defined set of possible observable events of a system must be defined. Additionally, it is important for timing analysis that the event occurrences can exactly be determined. In [83] we add the information that an observable event represents some kind of *action* in the system at

which data or control is handed over from one component to another. In this section we provide a clear definition of observable events as used in our work.

For a definition of the term, we first need to clearly distinguish between an observable event and its occurrences. The event models mentioned in Section 3.2.1 describe an event’s occurrence behavior. Each event instance of an event stream represents one event occurrence. The event itself consequently is the abstraction of single occurrences. It just denotes that events *of this type* can be observed, without describing when the observation can be made. To highlight the difference, we call this abstraction an *observable event*. It is a concept to abstract from concrete system behavior, which leads to the concrete time instances of event observations.

In summary, observable events differ from the two before mentioned usages in literature and development as follows:

- Observable events do not trigger any action like operating system events. We use the concept to represent certain system behavior that can be observed from outside the system passively.
- Observable events do not describe a certain occurrence behavior like event streams but represent a class of events. An event stream leads to the observation of single event instances (or occurrences) of that class at runtime.

We define observable events as follows:

Definition 3.1. *An observable event is an action performed by a system that can be observed and measured from outside the system. An occurrence of an observable event can be observed every time the system performs that action during execution.*

For timing specification and analysis especially data and control paths through the system are focused. Thus, in our context especially the observable events are of interest, where data or control is handed over from one system component to another [83]. More precise, in our work the set of possible observable events is limited to observable behaviors that may be objective for timing constraints. According to our used system model described in Section 3.3 we provide a well-defined set of system states and corresponding observable events for that system in Section 3.4.

Finally, to depict the difference between operating system events, observable events, event models and their occurrences consider Example 3.2.

Example 3.2. Let *task X activated* be an observable event. It is defined as the transition from the state *task X suspended* to the state *task X ready*, according to the task state model shown in Figure 2.2. Every time when task X is activated an occurrence of that event can be observed as a certain instance of the defined observable event. However, neither the observable event nor its occurrences are an operating system event that actually activates the task. The occurrence behavior of the task activation can be described using any of the available event models.

3.2.4 Definition of Event Chains

We already showed in a previous publication how the event chain concept can be used for timing analysis of a time-triggered system [82]. After we defined observable events in the previous section, we now provide the definition of an event chain. Event chains make use of observable events.

Definition 3.3. *An event chain refers to two observable events, namely the chain's stimulus event and the chain's response event. The semantics of an event chain is that an occurrence of the response occurs as causal consequence of each occurrence of the stimulus. Stimulus and response thus also have a clear temporal order.*

Note the causality of the stimulus and response events in this definition. According to Broy [10] we define causality as a logical dependency between the two events of an event chain. More precisely, we define it according to the *liveness* definition, i. e. the stimulus occurrence *leads to* the response occurrence.

Event chains can be refined with so-called sub-chains. A sub-chain itself is an event chain. Using sub-chains, a chain of sequential events can be modeled.

Example 3.4. Additionally the Example 3.2, let *task X terminated* be the observable event of the termination of task X. This observable event occurs each time after and occurrence of the observable event *task X activated* as a causal consequence. This can be expressed with an event chain that has *task X activated* as its stimulus and *task X terminated* as its response.

3.3 System Architecture Model Used in this Thesis

3.3.1 System Definition

Our methodology follows the model-based development approach as described in Section 2.3. The term *system* in our understanding actually denotes a *system model* that contains all information needed for our approach. The system model is similar to other models known from literature and industry. For example, AUTOSAR has established as system modeling method in the automotive industry [17]. The system model presented in this section is the basis for the rest of this thesis. It is similar to AUTOSAR but neglects a lot of details, which are not necessary for our work.

Figure 3.2 depicts a system that is embedded into its environment with a proper interface. Such systems, which have an interface to their environment, are called *open systems*. In contrast, closed systems do not have such an interface. That means they can neither react to input from their environment nor influence the environment. Of course, automotive systems rather are considered as open systems, because they have sensors and actuators as interfaces to their environment. In the context of this thesis we only mean electric/electronic interfaces of a system that are represented by sensors and actuators.

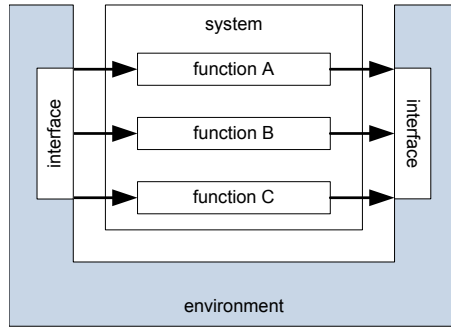


Fig. 3.2. A system, its interface and the environment.

According to Broy et al. [13], a system is a group of interacting, interrelated, or interdependent elements forming a complex whole and providing a set of services that are used by a user to carry out a specific purpose. The user can be a person or a group of persons, an enterprise or another system. In our case, the user of an automotive system is the driver who uses system services, which are also called functions or functionalities. A lot of such modern software-based or software-enabled automotive functions exist (see Section 2.2). A system is clearly separated from its environment (or context) and thus has a system boundary or interface. A system can interact with its environment at its boundary. The interaction with the environment can be in terms of force or energy, carried out using actuators and made available to the system using sensors (see Section 2.2.1).

When real-time systems are modeled and analyzed, this is mostly done to answer questions regarding time, timing constraints or timing behavior of the system. Therefore a precise definition of the concept of time as it is used in the system is necessary. There are mainly two different time concepts, namely continuous time and discrete time [10]. Time continuous systems assume a natural process of time and events can occur at arbitrary points in time. Computer systems however are executed technically on a digital time discrete machine. That is, most models of time of embedded systems are time discrete systems. Our system model is assumed to be time discrete. The time line is built up by equal intervals. Time advances in *ticks*. The model abstracts from the actual unit (e. g. seconds, milliseconds, or microseconds) of such ticks.

3.3.2 System Architecture Description Overview

For our methodology, a software architecture model, i. e. a model of the static structure of the system, is sufficient. The system's dynamic behavior, i. e. its expected runtime actions and interactions, is not in our scope. Thus, our system model does not contain a model of the system behavior. Furthermore, the model only contains the elements and attributes that are important for our approach. Other models of a static system architecture may be more comprehensive. AUTOSAR for example offers a lot more such elements and attributes.

Because of the growing complexity of modern automotive systems, models (i. e. formal descriptions) of the systems architecture become more and more important. In their comprehensive survey at different levels of system architectures Broy et al. [13] state that the introduction of an architecture paradigm is a proven way to reduce this complexity. The standardization effort done for AUTOSAR [17] is an evidence for the increasing importance of system architecture models for the automotive industry. Such architecture models are required in projects, where large systems are developed by many different organizations, as in the automotive industry. They are used to describe and evaluate the system, communicate such descriptions among stakeholders and organizations, managing the development activities and of course to verify the compliance of the system's implementation.

Broy et al. [13] define system architectures on six *levels of generality*. From the most concrete architecture to the most generic one, these are the following.

1. *Product Architectures* define the architecture for a concrete product.
2. *Product-line Architectures* define the architecture for a specific product line.
3. *Organization-specific Architectures* tailor a domain-specific architecture for a concrete company.
4. *Domain-specific Architectures* collect concepts, which are specific to a domain. AUTOSAR is such an architecture for the automotive industry.
5. *Common Architecture Frameworks* define concepts which are necessary for any kind of system.
6. *Meta Architecture Frameworks* introduce common terms like component or interface, which are fully independent of any kind of system.

The introduction of architectural terms by a formal meta architecture framework (level 6) is not explicitly done in this thesis. However, our system model is based on a common architecture framework (level 5), which is similar to the one presented by Broy et al. in [13] and [12]. AUTOSAR is one example of a domain-specific architecture (level 4) that is used in the automotive industry. Because the AUTOSAR model is very complex and offers a lot more details than necessary for our approach, we use an own system model as such a domain-specific architecture. Our system model and our approach can be mapped to AUTOSAR. Furthermore, we show how our model differs from the one by Broy et al. in this section. The three most concrete architectures (levels 1 to 3) are not further investigated. Our approach is applied only to a more abstract domain-specific architecture.

The system architecture model used in this thesis is partitioned into three main parts, also called layers. These layers represent an automotive system on three levels of abstraction. The relation of the three layers is shown in Figure 3.3, which is borrowed from Feilkas et al. [30]. The expression "low detail" in Figure 3.3 refers to the highest level of abstraction and a coarse granularity of modeling, high detail refers to the opposite.

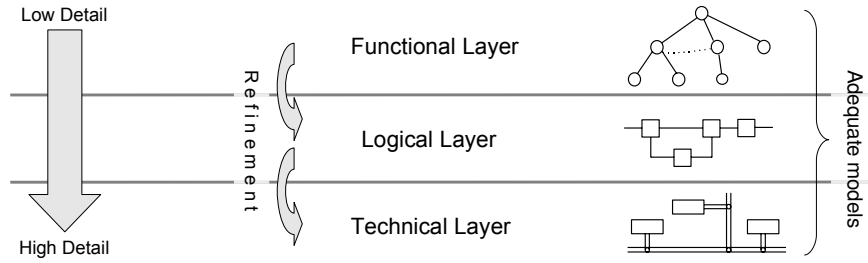


Fig. 3.3. Three abstraction layers of the system architecture model according to Feilkas et al. [30].

- **Functional Architecture:** description of functions and their external interfaces from a black box perspective, as they are provided by the system to its environment
- **Logical Architecture:** description of components and the component network, which is formed by their communication and realizes the functions of the system
- **Technical Architecture:** description of the technical realization using ECUs, communication busses and appropriate execution models for both

Figure 3.4 shows an overview of the system architecture model that we use for our approach. It structures the entire system model by the above-mentioned three views, namely function, logical and technical architecture.

We define a separate timing model called TIMEX to specify timing constraints for a system. TIMEX is described in Section 5.3. In this section we describe the basic system model we use throughout this thesis. The observable events that are provided by that model for the later use in TIMEX are described in the subsequent Section 3.4.

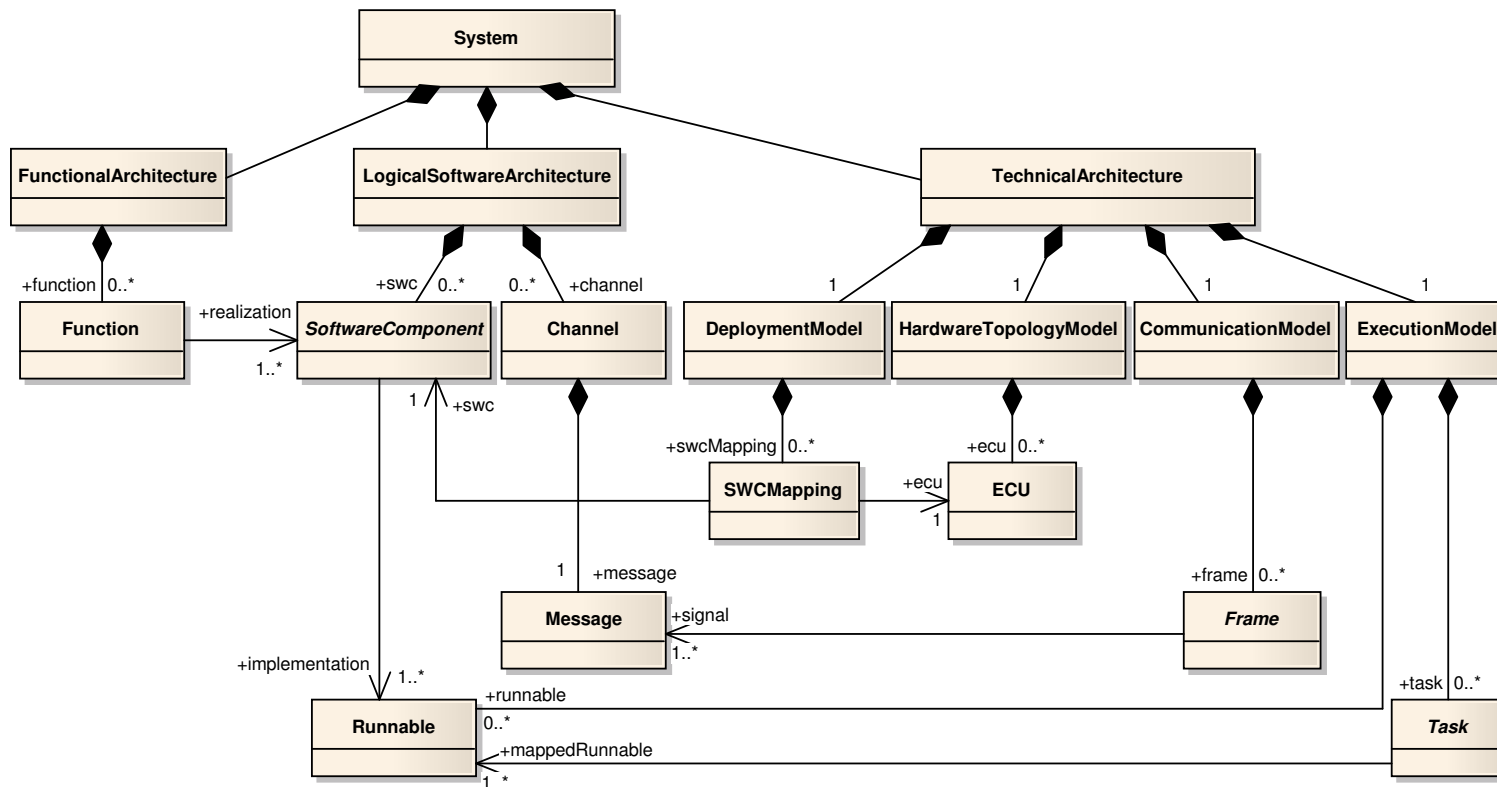


Fig. 3.4. Overview of the system architecture model used in this thesis.

3.3.3 Functional Architecture Model

Description of the Functional Architecture

The functional architecture model used in this thesis is quite simple. Basically, it is used to model solely two things:

- functions that are realized by the system
- the interface of each function to its environment

Later, the modeled functions are used to define function-triggered timing constraints. Therefore it is neither necessary to introduce a concept for function decomposition nor to provide the possibility to capture other than timing constraints. Functional decomposition, dependencies to other functions, and a comprehensive model of functional requirements is available in the architecture framework of Broy et al. [13, 12, 93], for example. Harhurin et al. [45] and Gruler et al. [41] describe a formal foundation for service hierarchies to model functional architectures and function compositions. For our approach we neglect this level of detail for the functional architecture.

A behavioral model for functions is not required in our system model. The functional architecture focuses on the input and output, i. e. the interfaces, of each function using sensors and actuators. The model thus can only be used for a black box description of functions, which is sufficient for our approach.

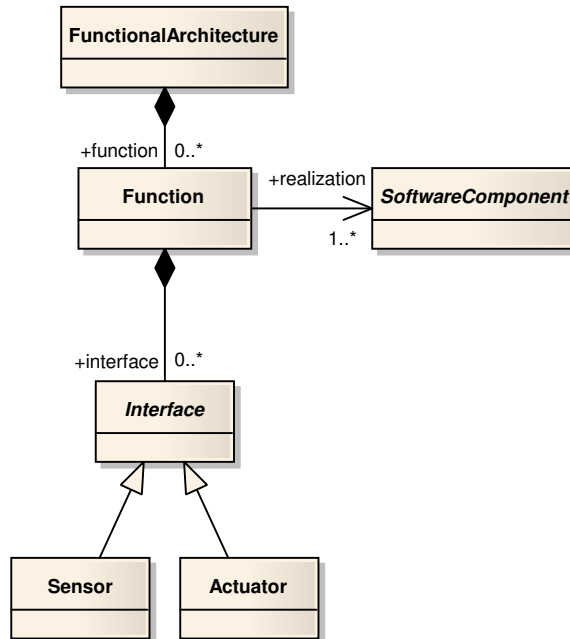


Fig. 3.5. The functional architecture of the system model.

An overview of the functional architecture is depicted in Figure 3.5. The functional architecture contains a set of functions. Each function is modeled as a black box. Later it is realized by at least one component of the logical architecture. The whole functional architecture defines the boundary of the system to its environment from the user perspective. Each function can have sensors and actuators as interface to the environment.

Formal Definition of Functions

A function f with sensor interfaces $s_i \in S_f$ and actuator interfaces $a_j \in A_f$ is denoted by a tuple

$$f = \langle S_f, A_f \rangle$$

S_f is the set of all sensor interfaces and A_f is the set of all actuator interfaces. S_f and A_f thus define the syntactic interface of function f . Semantic interfaces define which input values result in which output values, i. e. the functional behavior. A semantic interface definition is not required in our model, because the model does not contain the behavior of functions. Furthermore, the function interfaces do not have a type. Sensor interfaces make a message available to the function. Actuator interfaces make a message available to the environment. The message type is not relevant in our system model.

The functional architecture FA is the set of all functions f_k provided by the system.

$$FA = \{f_k\}$$

Figure 3.5 shows the reference called realization from function to component. This reference actually is the link between functional architecture and logical architecture because it determines the components that realize each function. This realization relation is formally denoted as a function from the set of all functions FA to the power set of all components C .

$$realization : FA \rightarrow C^*$$

3.3.4 Logical Software Architecture Model

Description of the Logical Software Architecture

The logical architecture is used to define a set of interconnected and communicating components, which realize the functions of the functional architecture. Usually, like in the model of Broy et al. [13, 12], technical details of such components are not further specified. Especially, the logical architecture abstracts from a technical realization using software or hardware, i. e. electronic

or mechanical/electrical realization. In our case however, we assume that all components are realized as software components.

All functions of the functional architecture must be represented by an *equivalent network* of components. That means each function must be represented by a set of components that has the same interface. A behavior model for components is not necessary, because the purpose of the model is not to execute or simulate the components. Therefore, a model of the component dynamics would be a prerequisite, which is offered by other logical architecture models, e.g. Broy et al. [13, 12] or Harhurin et al. [45].

The structuring of functions in practice can follow various patterns. Functions can be structured by technical, organizational, semantical, or other aspects. Such patterns are not in the scope of this thesis. Furthermore, a single component can be used to participate in the realization of one or several functions in practice. For simplification, we assume that every component is only used once in the realization of exactly one function.

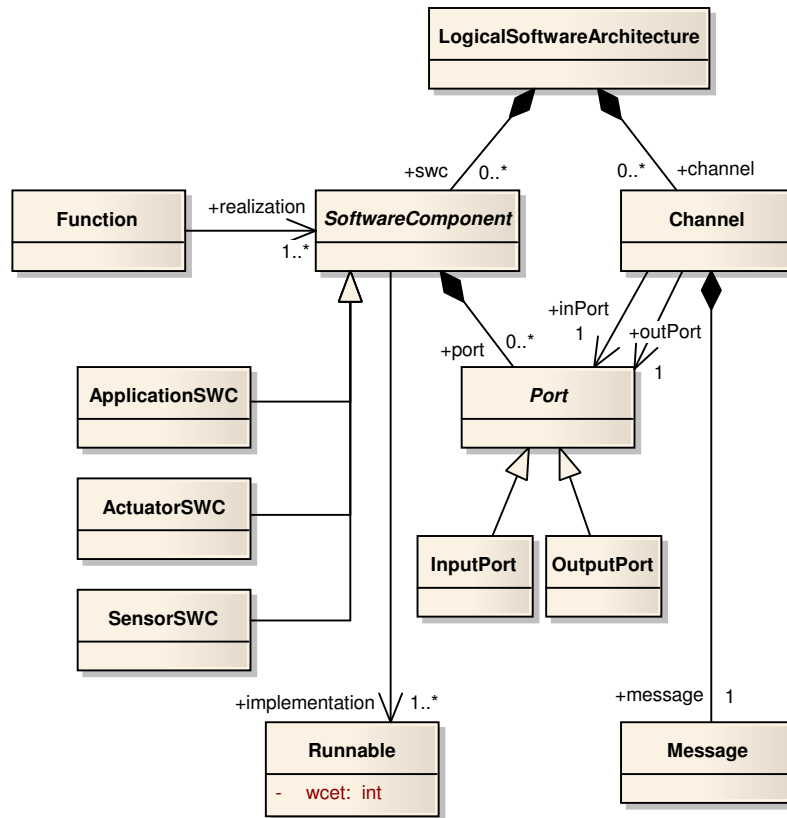


Fig. 3.6. The logical software architecture of the system model.

Figure 3.6 shows the logical software architecture model. The basic entity to model a software architecture is a software component (SWC). The software architecture of the system is modeled as a set of interconnected software components. Communication between software components is modeled using

channels and ports, which are described later. A software component is of one of the types application, actuator or sensor software component. The logical software architecture contains all interconnected SWC. For simplicity reasons, the software architecture in our model is not structured hierarchically to refine software components internally again with a network of components. Thus our model solely reflects the flattened result of a previously potentially hierarchically modeled structure. Harhurin et al. [45] describe a formal foundation for hierarchical component networks.

Each SWC references a set of so-called runnables that model pieces of executable code and cannot be further refined. Runnables make up the internal structure of a software component. They are therefore part of the technical architecture.

Three special types of software components exist. Sensor and actuator SWC can be used to model sensor and actuator access. Each sensor of an ECU needs a corresponding sensor SWC and each actuator needs a corresponding actuator SWC. For every sensor interface of the functional architecture one sensor SWC exists on the logical architecture. Accordingly, for every actuator interface of the functional architecture one actuator SWC exists on the logical architecture. Thus sensor and actuator SWC represent the function interfaces on the logical component architecture. Application SWC are used for software-based realizations without sensor or actuator access.

A SWC can have so-called ports. On an input port messages are received by the SWC. On an output port messages are sent by the SWC. If one SWC sends a message on one of its output ports to an input port of another SWC, this is modeled with a channel between these two ports. The channel actually contains the sent message. For our approach, we neglect data types of messages. We are only interested in the existence of a message that is exchanged between two components. For all connectors we assume direct asynchronous communication, often also called sender-receiver communication. A message is just pushed from a sender to a receiver without any synchronization. That means the sender does not wait for any answer of the receiver and continues its execution asynchronously to the receiver.

Formal Definition of Components

A component c with input ports $i_i \in I_c$ and output ports $o_j \in O_c$ is denoted by a tuple

$$c = \langle I_c, O_c \rangle$$

I_c is the set of all input ports and O_c is the set of all output ports. I_c and O_c thus define the syntactic interface of component c . A semantic interface definition is not required, because the model does not contain the behavior of components. Furthermore, the component ports do not have a type. Input ports make a message available to the component by receiving it from the sender component that shares the same channel. Output ports send a message

to the receiver component that shares the same channel. The message type is not relevant in our system model.

A channel l that connects input port i and output port o with message m is denoted by a tuple

$$l = \langle i, o, m \rangle$$

The logical architecture LA contains the set of all components $c_m \in C$ of the system that are used to realize all system functions of the functional architecture and the set of all channels $l_n \in L$ that connect the components to a component network. LA is denoted by the following tuple.

$$LA = \langle C, L \rangle$$

Figure 3.6 shows the reference called implementation from component to runnable. This reference actually is the link between logical architecture and technical architecture because it determines the runnables that are the implementation of a component. This implementation relation is formally denoted as a function from the set of all components C to the power set of all runnables R .

$$\text{implementation} : C \rightarrow R^*$$

Of course, we assume some side conditions for the logical architecture. First, the network of all components C must be a valid realization of all functions FA . That implies the following conditions.

- Each sensor interface of a function must be represented by a sensor SWC with an according input port.
- Each actuator interface of a function must be represented by an actuator SWC with an according output port.
- Each function must unambiguously be realized with components.

Second, we assume that every component is properly implemented by at least one runnable. Each runnable can only be used for the implementation of one component.

3.3.5 Technical Architecture Model

The technical architecture describes the realization of the system using software and hardware. As depicted in Figure 3.7 the technical architecture in our model consists of the four sub-models hardware topology model, deployment model, execution model, and communication model. Basically, the entire technical architecture is used to refine the following aspects of the logical architecture.

- It models the mapping of software components of the logical architecture to the ECU network of the technical architecture.
- It models the structure within a SWC of the logical architecture, where only its port interfaces are visible, with runnables.
- It refines the communication between two SWC when remote communication over a bus is necessary with the concept of signals in frames.

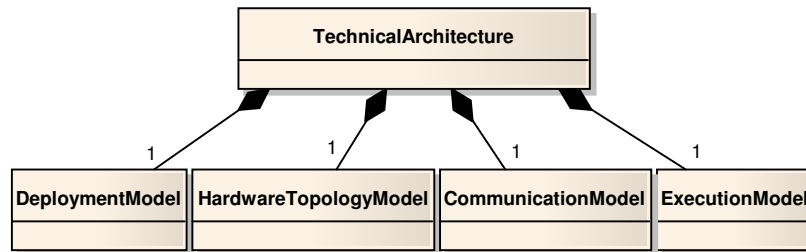


Fig. 3.7. The technical architecture of the system model consists of four sub-models.

We describe each of the four sub-models shown in Figure 3.7 in detail in the following four subsections.

Hardware Topology Model

The hardware topology of the system is modeled as a network of ECUs connected to exactly one communication bus. In practice of course, networks with several busses exist, which are linked with gateways. For simplification, we assume only one communication bus.

In our context, an ECU solely serves as a container for software components. ECU specific basic software (i. e. not application software components) as defined by AUTOSAR and hardware details are neglected. Sensor and actuator usage is sufficiently modeled using sensor and actuator SWC in the model of the functional software architecture. Figure 3.8 displays the hardware topology model.

There are two types of busses supported in the hardware topology model, namely an event-triggered (ET) and a time-triggered (TT) bus. We assume a CAN network for the event-triggered bus and a FlexRay network for the time-triggered bus. In a FlexRay network, the connected ECUs can either be synchronized to the bus or unsynchronized (see Section 2.1.4). So there are three different system types:

- event-triggered system
- synchronized time-triggered system
- unsynchronized time-triggered system

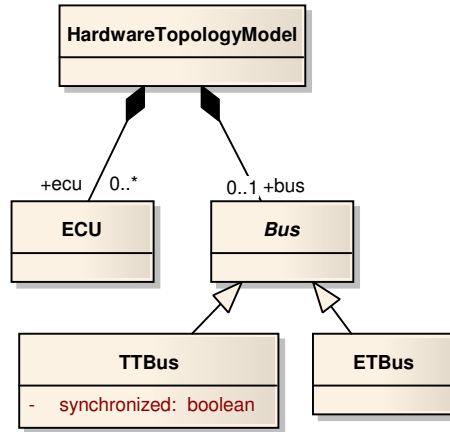


Fig. 3.8. The hardware topology model of the system model.

Later, the differentiation between the three system types is important for the generation of subsystem timing requirements from the given function-triggered timing constraints.

Formally, a hardware topology model HM consists of ECUs $e_i \in E$ and a bus b . HM is denoted by the tuple

$$HM = \langle E, b \rangle$$

E is the set of all ECUs.

The boolean attribute *synchronized* that indicates whether a TTBUS b offers a synchronous time base for all ECUs is denoted by the following function.

$$\textit{synchronized} : b \rightarrow \mathbb{B}$$

Deployment Model

The model of a complete system consists of a software architecture, a hardware topology and, most notably, the software mapping of software components to ECUs, often also called deployment. Our deployment model, which is also part of the technical architecture, is depicted in Figure 3.9.

A software component mapping describes which software component is executed on which ECU. As a result, all connectors of the logical software architecture model can now be identified as local or remote connectors. For local connectors immediate data transmission is assumed. In practice, synchronous local communication can be realized as local function calls. Asynchronous local communication can be realized as shared memory access. The messages of remote connectors however have to be transmitted over a communication bus that connects both ECUs, which run the communicating software components.

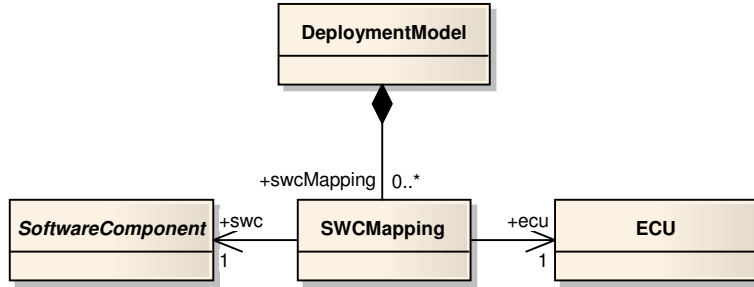


Fig. 3.9. The deployment model of the system model.

Formally, a software component mapping d that maps SWC c to ECU e is denoted by the following tuple.

$$d = \langle c, e \rangle$$

A deployment model DM is the set of all mappings d_i of the system.

$$DM = \{d_i\}$$

As every software component can only be mapped to exactly one ECU, the following condition for the cardinal number of a set of components C holds (we do not support several so-called instantiations of components, like AUTOSAR for example).

$$|C| = |DM|$$

Communication Model

Each message that is remotely exchanged between two software components that are mapped to different ECUs in the technical architecture, is represented by a so-called *signal*. The communication model is depicted in Figure 3.10.

The communication model contains the set of all frames. A set of messages is mapped into a frame that is transmitted over the communication bus. Such messages are then called signals. This is represented by the according reference called *signal* in Figure 3.10. A frame is sent by exactly one ECU and can be received by all ECUs that are connected to the bus. The contained signals be handed to the software components on the receiver ECUs if required. We assume that each signal can only be mapped to a frame once. Our model does not support multiple signal transmissions over several frames.

Every frame has some timing properties, which depend on the bus type. For an event-triggered (i. e. CAN) frame we assume the following timing relevant attributes in our communication model:

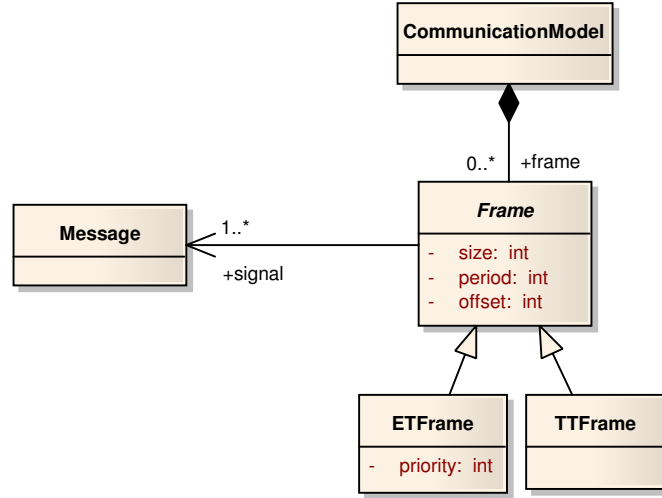


Fig. 3.10. The communication model of the system model.

- set of mapped signals
- frame size as result of the contained signals
- frame period
- the frame's start offset from the schedule start
- frame priority

Given the set of all frames F , the timing properties size, period, offset, and priority of an $ETFrame f_{ET} \in F$ are denoted by function $frame_{ET}$ as follows.

$$frame_{ET} : F \rightarrow \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$$

Note that not necessarily all timing properties must be set for event-triggered frames. Especially the period and offset properties are not always necessary. We investigate several task models of different scheduling approaches in Chapter 4 that make use of only some of the provided properties. Unset properties can be indicated by the value \perp of the according property of function $frame_{ET}$. Our system model is able to cover all task models.

The size of a FlexRay frame is static and depends on the FlexRay configuration. As described in Section 2.7, we assume two static configuration parameters base cycle and system cycle for a FlexRay network. The base cycle is the minimum possible period of each frame. The system cycle is the overall cycle of the network after which all execution and transmission action repeats. The system cycle thus is the maximum possible period of each frame. For a time-triggered (i. e. a FlexRay) frame we assume the following timing relevant attributes in our communication model:

- set of mapped signals
- frame size as result of the contained signals
- frame period
- the frame's start offset from the system start (common time base)

Given the set of all frames F , the timing properties size, offset, and period of a $TTFrame$ $f_{TT} \in F$ are denoted by function $frame_{TT}$ follows.

$$frame_{TT} : F \rightarrow \mathbb{N} \times \mathbb{N} \times \mathbb{N}$$

The so-called communication matrix models, which ECU reads or writes which frame and thus reads or writes which signal. This information can be gained from the software architecture model, which provides connectors between SWC, and the software mapping given by the deployment model, which determines remote connectors. Thus, the communication matrix is not explicitly modeled.

Given the set of channels L of a logical architecture, the set of all messages M is defined as follows.

$$M = \{m \mid \exists \langle in, out, m \rangle \in L\}$$

The mapped signals of each frame $f \in F$ are formally denoted by the function $signals$, where M^* is the power set of all messages M .

$$signals : F \rightarrow M^*$$

Formally, a communication model CM is a set of frames f_j that are remotely transmitted over the bus.

$$CM = \{f_j\}$$

Execution Model

The software components that are mapped onto an ECU are executed by the operating system running on that ECU. More precisely, the runnables that implement the components have to be executed by the operating system. Therefore, a set of runnables is mapped to operating system tasks. The execution model contains all runnables that implement the systems components. A task is the actual operating system element that executes runnables on one specific ECU. A task has different timing properties, depending on the operating system type. However, independently of this type, each tasks contains a set of mapped runnables. As runnables model pieces of executable code, they

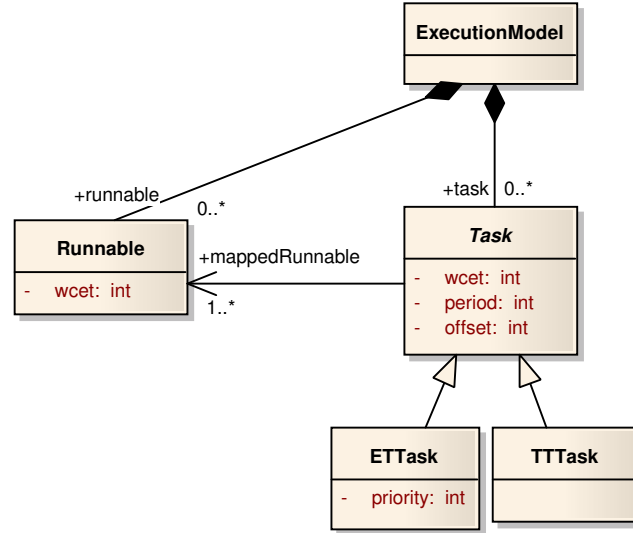


Fig. 3.11. The execution model of the system model.

have an attribute called WCET. This is the runnable's WCET on the target ECU.

Similar to the supported communication bus types, our execution model, which is shown in Figure 3.11, supports two types of operating systems. An event-triggered operating system is assumed for an ECU in an event-triggered, i. e. CAN-based, network. A pure time-triggered operating system is assumed for an ECU in a time-triggered, i. e. FlexRay-based, network.

In an event-triggered network also an event-triggered operating system for ECUs is assumed. Its tasks thus have the following timing-relevant attributes.

- set of mapped runnables
- task worst case execution time, which is the sum of the WCET of the contained runnables
- task period
- task offset from the schedule start
- task priority

The timing properties worst case execution time and priority of an ETask $t_{ET} \in T$ are denoted by function $task_{ET}$ as follows.

$$task_{ET} : T \rightarrow \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$$

As described in the previous section for event-triggered frames, not all timing properties must be set for event-triggered tasks, depending on the applied task model.

Accordingly, we assume a simple time-triggered operating system for ECUs on a time-triggered network. The following timing properties are available for time-triggered tasks:

- set of mapped runnables
- task worst case execution time, which is the sum of the WCET of the contained runnables
- task period, which must be an integer divisor of the system cycle and a multiple of the base cycle
- the task's offset from its cycle start time

The timing properties worst case execution time, offset, and period of a TT-Task $t_{TT} \in T$ are denoted by function $task_{TT}$ follows.

$$task_{TT} : T \rightarrow \mathbb{N} \times \mathbb{N} \times \mathbb{N}$$

The timing property WCET of a runnable is denoted by function $wcet$, where R is the set of all runnables.

$$wcet : R \rightarrow int$$

The mapped runnables of each task $t \in T$ are denoted by function $runnables$, where R^* is the power set of all runnables R .

$$runnables : T \rightarrow R^*$$

The WCET of a task is defined as the sum of the WCET of all contained runnables.

Formally, an execution model is a tuple EM of a set of runnables $R = \{r_i\}$ and a set of tasks $T = \{t_j\}$.

$$EM = \langle R, T \rangle$$

3.3.6 Example System Model

To summarize the system model section, we now define an example system model using the introduced formalism. Figure 3.12 shows a visualization of the system model, which is formally specified thereafter (execution model and communication model are omitted in the figure).

The system consists of only one function *damperControl*, which controls a chassis damper according to a tilt sensor input to stabilize the car. The function has only one sensor interface *tiltSensor* and only one actuator interface *damper*.

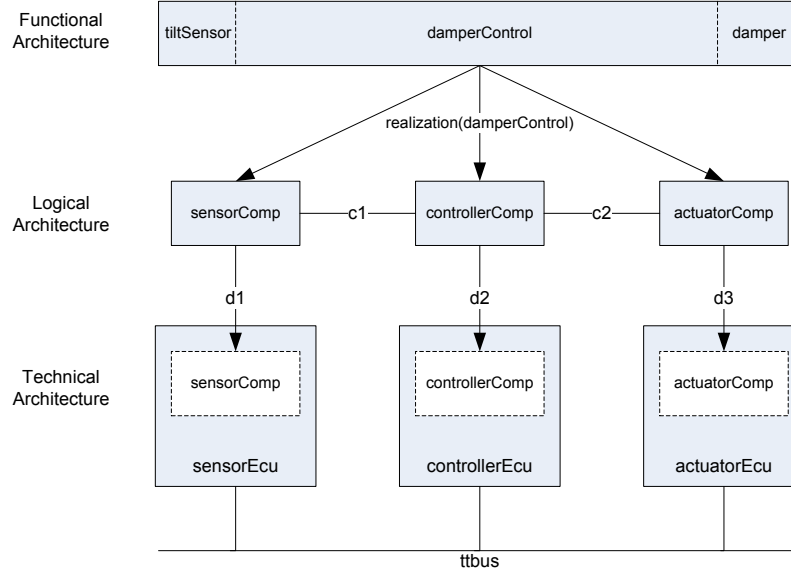


Fig. 3.12. Visualization of the formally specified system model example.

$$\begin{aligned}
 \text{damperControl} &= \langle \text{Sensors}, \text{Actuators} \rangle \\
 \text{Sensors} &= \{ \text{tiltSensor} \} \\
 \text{Actuators} &= \{ \text{damper} \}
 \end{aligned}$$

The functional architecture of our example is

$$FA = \{ \text{damperControl} \}$$

The function is realized by three components in the logical architecture *LA*. These are a sensor component *sensorComp*, an application component *controllerComp* and an actuator component *actuatorComp*. The set of components thus is

$$\text{Components} = \{ \text{sensorComp}, \text{controllerComp}, \text{actuatorComp} \}$$

The *realization* relation, which maps a function to components, is defined by

$$\text{realization}(\text{damperControl}) = \{ \text{sensorComp}, \text{controllerComp}, \text{actuatorComp} \}$$

The three components are further defined by.

$$\begin{aligned}
 \text{sensorComp} &= \langle \emptyset, \{ \text{sensorOut} \} \rangle \\
 \text{controllerComp} &= \langle \{ \text{controllerIn} \}, \{ \text{controllerOut} \} \rangle \\
 \text{actuatorComp} &= \langle \{ \text{actuatorIn} \}, \emptyset \rangle
 \end{aligned}$$

To define the network of these communicating components, the set of channels is required as follows.

$$\begin{aligned}
Channels &= \{c1, c2\} \\
c1 &= \langle sensorOut, controllerIn, m1 \rangle \\
c2 &= \langle controllerOut, actuatorIn, m2 \rangle
\end{aligned}$$

This completes the logical architecture of the example.

$$LA = \langle Components, Channels \rangle$$

The hardware topology model HM consists of three ECUs and one time-triggered bus.

$$HM = \langle \{sensorEcu, controllerEcu, actuatorEcu\}, ttbus \rangle$$

The ECUs are synchronized to the bus.

$$synchronized(ttbus) = true$$

The deployment model DM consists of three mappings, one for each component.

$$\begin{aligned}
DM &= \{d1, d2, d3\} \\
d1 &= \langle sensorComp, sensorEcu \rangle \\
d2 &= \langle controllerComp, controllerEcu \rangle \\
d3 &= \langle actuatorComp, actuatorEcu \rangle
\end{aligned}$$

The communication model CM of the example system contains two time-triggered frames.

$$CM = \{f1, f2\}$$

Remotely exchanged messages are mapped to the frames as follows.

$$\begin{aligned}
signals(f1) &= \{m1\} \\
signals(f2) &= \{m2\}
\end{aligned}$$

The two frames have the following timing properties, which must be specified for time-triggered frames (size, offset and period).

$$\begin{aligned}
frame_{TT}(f1) &= \langle 64, 2, 10 \rangle \\
frame_{TT}(f2) &= \langle 64, 6, 10 \rangle
\end{aligned}$$

The execution model EM of the example system contains three runnables and three tasks.

$$\begin{aligned}
EM &= (Runnables, Tasks) \\
Runnables &= \{r1, r2, r3\} \\
Tasks &= \{t1, t2, t3\}
\end{aligned}$$

The execution times of the runnables are

$$\begin{aligned}
wcet(r1) &= 1 \\
wcet(r2) &= 2 \\
wcet(r3) &= 1
\end{aligned}$$

The runnables are mapped to the tasks as defined by function *runnables*.

$$\begin{aligned}
runnables(t1) &= \{r1\} \\
runnables(t2) &= \{r2\} \\
runnables(t3) &= \{r3\}
\end{aligned}$$

The tasks have the following timing properties, which must be specified for time-triggered tasks (size, offset and period).

$$\begin{aligned}
task_{TT}(t1) &= \langle 1, 0, 10 \rangle \\
task_{TT}(t2) &= \langle 2, 3, 10 \rangle \\
task_{TT}(t3) &= \langle 1, 7, 10 \rangle
\end{aligned}$$

The *implementation* relation, which maps a component to runnables, is defined by

$$\begin{aligned}
implementation(sensorComp) &= \{sensorRunnable\} \\
implementation(controllerComp) &= \{controllerRunnable\} \\
implementation(actuatorComp) &= \{actuatorRunnable\}
\end{aligned}$$

The technical architecture in summary is

$$TA = \langle HM, DM, CM, EM \rangle$$

3.4 Observable Events of the System Model

The set of observable events in a system can be very large depending of the accuracy of the system model. The more details a system model has got, the more events can potentially be observed in that system model. It is useful to concentrate on those events that are important for the specification of timing constraints and thus for the definition of event chains.

According to the supported types of function-triggered timing constraints (Section 3.5.3), the collaboration workflows that we cover (Section 3.1.2),

and our system model (Section 3.3) we propose the following set of observable events. They are grouped by the three abstraction layers of the system model. On each layer individual observable events exist.

We define the observable events of our system model based on the concept of *actions* as introduced by the system specification method of Broy et al. [11]. Actions represent basic activities in a system as well as at its boundary to the environment. Examples for actions are sending messages between components or pressing a button at the system's interface.

3.4.1 Observable Events of the Functional Architecture

As defined in Section 2.2.1, the system's interface is represented by its sensors and actuators. These are the boundary of the system to its environment, also called the system's interface. In the functional architecture model defined in Section 3.3.3 the system's functionality is partitioned to a group of functions. Each function can have sensor and actuator interfaces. All these interfaces together represent the systems interface to its environment. All observable events of the functional architecture thus represent the whole set of observable events of the system from a user's viewpoint, i. e. with the system seen as a black box.

In our functional architecture there are two types of observable events.

- A *sensor observable event* can be observed at a sensor interface of a function when data is made available to the system through this interface.
- An *actuator observable event* can be observed at an actuator interface of a function when interaction with the environment is carried out through this interface.

According to Broy et al. [11], we define so-called *environmental actions* for the function interfaces. These are actions that occur at the system's boundary. For a sensor interface, we define one action that makes data available to the system through the sensor. Accordingly, we define one action for an actuator interface that represents the interaction of the system with its environment.

Given a function $f = \langle S_f, A_f \rangle$. For every $s \in S_f$ we denote the environmental input action, which we briefly call *sensor event*, by

$$\text{sensor}(s)$$

For every $a \in A_f$ we denote the environmental output action, which we briefly call *actuator event*, by

$$\text{actuator}(a)$$

Given a functional architecture FA . We can now define the set of all its sensor events $Sensor(FA)$.

$$Sensor(FA) = \{sensor(s) \mid \exists \langle S, A \rangle \in FA : \exists s \in S\}$$

The set of all its actuator events $Actuator(FA)$ is defined as follows.

$$Actuator(FA) = \{actuator(a) \mid \exists \langle S, A \rangle \in FA : \exists a \in A\}$$

The set of all observable events of the functional architecture $Interface(FA)$ is defined as the following set union.

$$Interface(FA) = Sensor(FA) \cup Actuator(FA)$$

3.4.2 Observable Events of the Logical Architecture

Additionally to the observable events at the system interface, the logical architecture reveals some more observable events within the system. More precisely, the observable events of the logical architecture occur at the ports of its software components.

Again, we rely our definition of observable events on the action concept of Broy et al. [11], who also call this kind of actions *system actions*, in contrast to the environmental actions of the functional architecture that we defined before. Components communicate via output and input actions asynchronously with each other, as we defined in Section 3.3.4. In our system model this is indicated by the message that is contained by each channel. A channel connects one output port with one input port. Input and output actions occur when a software component sends or receives a message on its input and output port on the according channel.

Accordingly, we define two types of observable events for the logical architecture.

- A *send message observable event* can be observed at an output port every time the software component sends the message of the connected channel.
- A *receive message observable event* can be observed at an input port every time the software component receives the message of the connected channel.

Given a component $c = \langle I, O \rangle$ and a channel $l = \langle in, out, m \rangle$. We denote the port output action "component c sends message m on port out ", which we briefly call *send event*, by

$$send(m, c) : out \in O$$

We denote the port input action "component c receives message m on port in ", which we briefly call *receive event*, by

$$receive(m, c) : in \in I$$

Given a logical architecture $LA = \langle C, L \rangle$. We define the set of all its send events $Send(LA)$ as follows.

$$Send(LA) = \{send(m, c) \mid \exists \langle in, out, m \rangle \in L : \exists c = \langle In, Out \rangle \in C : out \in Out\}$$

We define the set of all its receive events $Receive(LA)$ as follows.

$$Receive(LA) = \{receive(m, c) \mid \exists \langle in, out, m \rangle \in L : \exists c = \langle In, Out \rangle \in C : in \in In\}$$

The set of all observable events of the logical architecture $Port(LA)$ is defined as the following set union.

$$Port(LA) = Send(LA) \cup Receive(LA)$$

3.4.3 Observable Events of the Technical Architecture

The technical architecture enables further refinement of the system model and thus reveals additional observable events. These can be observed focusing on remotely exchanged messages between two ECUs, which we called signals in Section 3.3.5. Of course, our system model would enable the definition of additional observable events of the technical architecture, for example when focusing on runnables. However, as we explain in Section 5.3, these are not necessary for our TIMEX model. Our supported collaboration workflows (Section 3.1.2) do not require the timing model to disclose internals of software components. Therefore, observable events for the hand over of actions to or from runnables are not necessary.

The concept of actions was introduced by Broy et al. [11] basically to describe interaction between components and interaction of the system with its environment. For the definition of observable events of the technical architecture we stick to the action concept and extend it to signals that are sent over a bus.

A signal represents a message that is remotely exchanged between software components using a communication bus. Two important events can be observed during the transmission of a signal.

- A *signal queued for transmission observable event* can be observed when a signal is ready to be transmitted by the sending ECU on the bus. Technically this is the time when the frame that contains the signal is present in the send buffer of the according ECU's communication controller.

- A *signal transmitted observable event* can be observed when a signal has been transmitted on the bus and thus simultaneously been received by all ECUs that are connected to the bus. Technically this is the time when the frame that contains the signal is present in the receive buffer all according ECUS' communication controllers.

For simplification, we first define the set L_{remote} that contains all channels of the logical architecture, which are realized as remote communication on the technical architecture. Given a logical architecture $LA = (C, L)$ and a deployment model DM . The set of all remote channels L_{remote} is defined as follows.

$$\begin{aligned}
L_{remote} = \{ \langle in, out, m \rangle \in L \mid \\
& \exists c1 = \langle In1, Out1 \rangle \in C \wedge \exists c2 = \langle In2, Out2 \rangle \in C : \\
& in \in In1 \wedge out \in Out2 : \\
& \exists \langle c1, ecu1 \rangle \in DM \wedge \langle c2, ecu2 \rangle \in DM : \\
& ecu1 \neq ecu2 \}
\end{aligned}$$

Every signal is sent by exactly one ECU and received by all ECUs, which are connected to the bus. Thus, similar to the input and output actions of messages at component ports, we define transmit and receive actions for signals at ECUS.

Given a deployment model DM , an ECU ecu and a message m . We denote the transmit action "ECU e transmits message m ", which we briefly call *queued event*, by

$$queued(m, ecu) : \exists \langle in, out, m \rangle \in L_{remote} : \exists \langle \langle In, Out \rangle, ecu \rangle \in DM : out \in Out$$

The formula literally states that the queued event can be observed for every message that is remotely transmitted and whose sending component is on the according ECU.

A signal is received by all ECUs at the same time. From the timing viewpoint it is thus sufficient to define only one receive action. Therefore the common bus instead of each single ECU as reference is appropriate. Given a hardware topology model $HM = (E, b)$ and a message m . We denote the receive action "message m received by all ECUS on bus b ", which we briefly call *transmitted event*, by

$$transmitted(m, b) : \exists \langle in, out, m \rangle \in L_{remote}$$

The formula literally states that the transmitted event can be observed on the bus for every message that is remotely transmitted.

Given logical architecture with a set of Channels L and a technical architecture TA with a deployment model DM and a hardware topology model $HM =$

(E, b) . The set of all queued events of the technical architecture $Queued(TA)$ is defined as follows.

$$\begin{aligned}
 Queued(TA) = & \{queued(m, ecu) \mid m \in L_{remote} \wedge \\
 & \exists \langle In, Out \rangle, ecu \in DM : \\
 & \exists \langle in, out, m \rangle \in L : out \in Out\}
 \end{aligned}$$

The set of all transmitted events $Transmitted(TA)$ of the technical architecture TA with bus b is defined as follows.

$$Transmitted(TA) = \{transmitted(m, b) \mid m \in L_{remote}\}$$

The set of all observable events of the technical architecture $Signal(TA)$ is defined as the following set union.

$$Signal(TA) = Queued(TA) \cup Transmitted(TA)$$

3.4.4 Example for all Observable Events of a System Model

Based on the formalism for observable events, we can now define all such events for the example system model of Section 3.3.6. Figure 3.13 shows the temporal order of all these observable events. Note that this ordering of events is only possible, because all events belong to the same data path of the system.

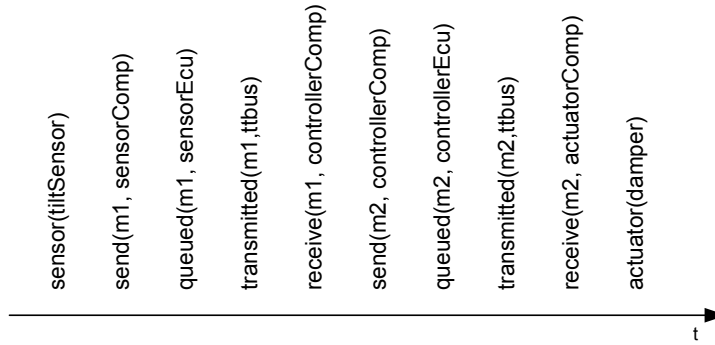


Fig. 3.13. The temporal order of all observable events of the system model example.

Given the functional architecture FA , the following sensor and actuator events exist.

$$Sensor(FA) = \{sensor(tiltSensor)\}$$

$$Actuator(FA) = \{actuator(damper)\}$$

The set of all available sensor and actuator events thus is

$$Interface(FA) = \{sensor(tiltSensor), actuator(damper)\}$$

Given the logical architecture $LA = \langle Components, Channels \rangle$, the following send and receive events exist.

$$Send(LA) = \{send(m1, sensorComp), send(m2, controllerComp)\}$$

$$Receive(LA) = \{receive(m1, controllerComp), receive(m2, actuatorComp)\}$$

The set of all available send and receive events thus is

$$Port(LA) = \{send(m1, sensorComp), receive(m1, controllerComp), \\ send(m2, controllerComp), receive(m2, actuatorComp)\}$$

Given the technical architecture TA , the following queued and transmitted events exist.

$$Queued(TA) = \{queued(m1, sensorEcu), queued(m2, controllerEcu)\}$$

$$Transmitted(TA) = \{transmitted(m1, tbus), transmitted(m2, tbus)\}$$

The set of all available queued and transmitted events thus is

$$Signal(TA) = \{queued(m1, sensorEcu), transmitted(m1, tbus), \\ queued(m2, controllerEcu), transmitted(m2, tbus)\}$$

The set of all available observable events of the example system model is $Interface(FA) \cup Port(LA) \cup Signal(TA)$. We use these observable events later within our TIMEX model to define both temporal relations of the system and its timing constraints. TIMEX is introduced in Section 5.3.

3.5 Function-triggered Timing Constraints

In this section we introduce the most basic concept of our thesis, namely function-triggered timing constraints. Before that, it is necessary to understand the different meanings of timing constraints in real-time systems literature. There are many ways and different abstraction levels to describe such “time bounds” for a real-time system. We summarize these methods in Section 3.5.1, provide our definition of function-triggered timing constraints in Section 3.5.2 and describe the types of function-triggered timing constraints used in this thesis in Section 3.5.3.

3.5.1 Timing Constraints in Real-time Systems Literature

Due to Definition 2.1 of a real-time system timing constraints play a decisive role in real-time systems and timing analysis literature. However, the understanding of what a timing constraint is differs depending on the particular theory and its scope and individual modeling approach. This section provides an overview of the types and meanings of timing constraints in real-time systems literature. Note that the terms *timing constraint* and *timing requirement* are often used somewhat mixed up, even within one publication. Mostly they have the same meaning. In this literature survey we stick to the term timing constraint. However, we explicitly distinguish between a timing constraint and a timing requirement later in our work. Both terms have their own explicit meaning. The difference is explained in Section 3.6.

Basically, all available timing constraints can be divided into two groups, according to their scope:

- component level timing constraints
- system level timing constraints

On both levels, another differentiation can be done according to the real-time system type that is investigated, i. e. a time-triggered or an event-triggered system.

Component Level Timing Constraints

Component level timing analysis focuses on the timing behavior on one resource, i. e. one processor or one bus. Timing constraints on that level mostly target single tasks or frames. Effects of task interaction and communication can only – if at all – be taken into account on that component.

In their fundamental work, Liu and Layland [59] proof that Rate Monotonic Scheduling (RMS) is an optimal priority assignment method for strictly periodic tasks. The component in that case is a single processor. The only timing constraint for a task is a task deadline. A task deadline is the maximum time interval that may elapse after a task starts until the task execution has to

be finished. In their work they assume a fixed task deadline that is equal to the task period. Of course this is a very restrictive assumption. Audsley et al. [3] relaxed that assumption later and also allowed deadlines smaller than the task periods. They proved that also Deadline Monotonic Scheduling with task deadlines smaller than the task's periods is an optimal scheduling strategy. However, the focus of the timing constraints is still just on the level of a single task. There are several other single processor scheduling analysis approaches that assume similar task models and focus on single task completion as the only timing constraint (for example Earliest Deadline First, compared with RMS by Buttazzo [18]; Least Laxity First analyzed for example by Uthaisombut [96]).

Ekelin and Jonsson [27, 26] group task specific timing constraints to four categories:

- Intra-task timing constraints are constraints that restrict the time limits within which a single task must execute. This group contains task period, deadline, release time as well as input and output jitter.
- Inter-task timing constraints express time limits within which two tasks should execute in temporal relation to each other. Constraints of this group originate from the application requirements. They are called distance (due to input/output delay), freshness (due to aging of data), correlation (due to limits on the allowed time-skew in concurrent operations) and harmonicity (divisible sender and receiver periods) constraints.
- Intra-task execution constraints are local to a single task and determine on what processor and with what resources the task should execute.
- Inter-task execution constraints determine in what order, on what processor, and with what resources two or more tasks should execute in temporal relation to each other.

For each of their defined timing constraints Ekelin and Jonsson explain its taxonomy. A constraint thus is natural, implementation-based or artificial. Natural constraints are derived from the system requirements. Implementation-based constraints are a consequence of the chosen hardware, software and its scheduling. Artificial constraints are a result of adapting to limitations in existing scheduling algorithms.

Most constraints used for the analysis of bus scheduling also belong to the group of component level timing constraints. The analysis concepts and models often resemble the ones known from task scheduling on a processor. Marques et al. [61] propose two heuristics for frame packing, i. e. packing signals into communication frames and find a feasible priority assignment for the frames. Thereby, the frame packing has to fulfill a couple of constraints, amongst them are also timing constraints. Each signal has a predefined transmission period and a transmission deadline. Similar to the task scheduling problems discussed above, the deadlines are just constraining the timing behavior of a single signal. Saket and Navet [78] assume the same timing constraints for signals. In their proposed frame packing strategy, they transform the deadline constraints of the signals in a frame to a deadline constraint of the

frame itself. This enables the exploitation of established schedulability tests for frame scheduling, in this case a priority allocation algorithm proposed by Audsley [2]. Still the used timing constraints are frame-specific and thus on a component level.

Typical timing constraints for time-triggered real-time systems differ from the ones mentioned above. In a purely time-triggered system, tasks are scheduled without preemption on a time line in sequential order. Chung and Dietz [21] propose an algorithm to schedule tasks – they call them instructions – in this way. A schedule generated by that algorithm is only valid, if some timing constraints are fulfilled. Therefore, they define four kinds of standardized precedence constraints for instructions. These are before constraint (an instruction must be scheduled before another instruction), after constraint (an instruction must be scheduled after another instruction), concurrent constraint (an instruction must be scheduled at a certain time instance) and exclusive constraint (an instruction must not be scheduled at a certain time instance). Timing constraints in this approach also refer to single tasks, called instructions here, and thus are component level timing constraints.

Though considering component timing, Gerber et al. [37] take more complex timing constraints into account than the models discussed so far. The tasks are organized as a so-called task graph. A path of that graph can represent an end-to-end dependency between an input (sensor) and an output (actuator) of the modeled system. Three kinds of timing constraints can be defined for such end-to-end paths. A maximum input to output delay, called freshness constraint, constrains the time that may elapse between the sensor input and the according actuator output. The sensor input can be constrained with an input sampling constraint that bounds the maximum time-skew between inputs, called correlation constraint. A separation constraint can be used to constrain the maximum allowed jitter between consecutive value updates on an output channel. In their publication they present a method to derive intermediate timing properties from the constraints. These properties are task periods, offsets and deadlines, i. e. task-based constraints as used in the other component level approaches described before.

System Level Timing Constraints

Although categorized as component level timing constraints, the work of Gerber et al. presents an example of more complex timing constraints than just constraining task properties. However, in contrast to their work, such end-to-end timing constraints are also often considered on a system level and expressed over end-to-end paths across different resources (processors and communication busses). Task- or message-based timing constraints are considered as so-called timing properties, like in the definition of a timing augmented system model in Section 2.3.3. The overall timing behavior, given by such timing properties, must fulfill the end-to-end constraints defined on system level.

Saksena [79] defines timing constraints on three levels (see Figure 3.14). The first level are requirements for the performance of a system. Timing constraints are rather implicit on that level and represent requirements that stem

from physics of the system or from customer requirements. Typical timing constraints are specified on the second level, which is called system level, to ensure that the desired performance requirements are fulfilled. System level timing constraints are derived from performance requirements. The author identifies typical kinds of such constraints. These are end-to-end delays, sensor sampling, output jitter and maximum activation periods for computations. As third level for timing constraints the author identifies some task level timing attributes that were already investigated in Section 3.5.1. These are for example task specific periods, deadlines and phases (i. e. offsets). Task level timing constraints are derived from system level timing constraints. The dependency of these different constraint types is shown in Figure 3.14, which is taken from [79]. The figure also shows backward arrows. According to the authors these arrows indicate that new task level or system level constraints must be derived, if no scheduling can be found. The performance requirements remain invariant.

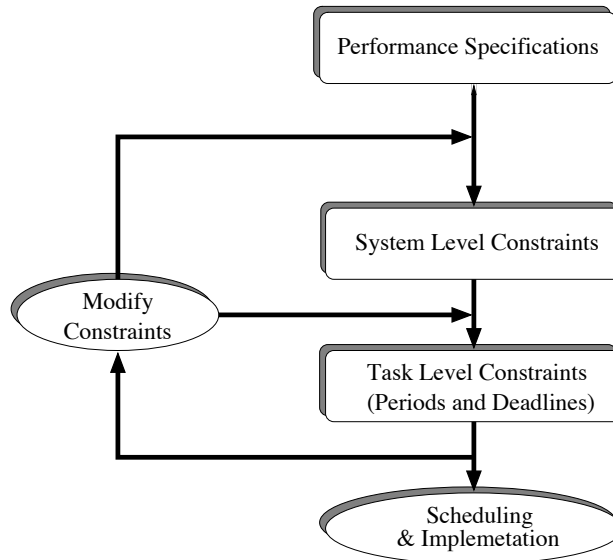


Fig. 3.14. Constraint types and their relation according to Saksena [79].

A real system level end-to-end view on timing constraints is considered by Schild and Würtz [84]. They focus on strictly time-triggered systems with several distributed processing units that are connected with exactly one communication bus. The software system consists of a task graph with asynchronous communication between tasks (graph nodes), modeled as graph edges. End-to-end latency constraints can be used to express a maximum latency between one task's start time and another task's end time, including potential bus communication. The paths through the resulting static task and frame schedules must fulfill their latency constraints. Again, this approach uses system level timing constraints to generate appropriate task level timing constraints.

Richter et al. [74, 49] developed a scheduling and timing analysis method based on event model interfaces between system components. Components are tasks

and messages. Basically, the approach does timing analysis of the given system. The approach enables verification of and reasoning about end-to-end paths through an event-triggered system with respect to path latencies. The analysis output answers the question whether end-to-end latency constraints are fulfilled. In a recent work, Feiertag et al. [29] further investigated the semantics of such end-to-end latency constraints. The authors explain, how so-called over- and undersampling effects can occur along a communication path. These effects basically stem from read and write access to an exchanged signal through registers and bus messages, if read and write cycles have different cycle times. Thus, four different path delay semantics can be identified. The maximum data age semantics (or maximum path delay) and the first observable reaction semantics are the two timing constraint semantics with most practical importance.

These two semantics for latency constraints are also implemented by the AUTOSAR Timing Specification [4]. It provides a very generic concept to a) model timing dependencies in an AUTOSAR system and b) specify timing constraints for the system. An overview of the AUTOSAR Timing Specification is provided in Section 2.3.2.

A perspective on timing constraints in automotive real-time systems, especially for body electronics, is given by Gehrke et al. [36]. For the body functions the authors investigated they define four types of such timing constraints. These can be categorized as system level constraints, although the authors do not distinguish the constraint types, like we do in this survey. The first two rather similar constraint types are maximum and exact execution time. According to the examples given, these types basically denote latency constraints, namely a maximum and an exact (with tolerance) latency between two events. Their third type is synchronicity of several events within a temporal tolerance. The fourth type are conditional timing constraints for complex runtime scenarios, where the existence of an event requires the existence of another event within a defined time bound. In a car's body electronics especially latency and synchronicity timing constraints are typical.

3.5.2 Definition of Function-triggered Timing Constraints

As the related work survey in Section 3.5.1 shows, two basic understandings of timing constraints emerged in real-time systems theory. Component level, or task level timing constraints are used to imply local bounds to a process, task, frame or any other local component. System-level timing constraints are used to constrain different inter-component end-to-end relations of mostly distributed real-time systems. Especially Schild and Würtz [84] as well as Saksena [79] share a similar idea of separating system level end-to-end timing constraints and task level timing properties, which are derived from the end-to-end constraints.

Both ideas, however, already assume a certain software system, hardware topology and software mapping. This is information that – in our sense – already is part of the system implementation. That means according to these author's definitions, timing constraints can be captured if the system implementation is already known and just task level timing properties need to be

derived. Function-triggered timing constraints take the idea of different levels of constraint semantics one step further, as described now.

As described in Section 2.2, an automotive system is developed to realize a set of automotive functions. These mostly are customer functions, i. e. functions, which the car provides to the driver and which are recognizable by the driver. The execution of these functions – however implemented – often is tied to some timing related boundaries.

In our understanding, timing constraints are either a result of functional physics or requirements from a customer’s perspective. Timing constraints in our context refer to a function, regardless of the function’s implementation in the car. Therefore we call such kind of timing constraints *function-triggered timing constraints*.

Definition 3.5. *A function-triggered timing constraint constrains a timing attribute of an automotive function by means of an end-to-end relation, abstracting from all implementation details and their timing properties.*

An implementation by means of hardware and software must then be chosen in a way that all function-triggered timing constraints are always fulfilled. An implementation contains a lot of so-called timing properties that influence the timing behavior. Focusing on software, such properties are execution times, task and frame periods, task and frame priorities. Also the software composition itself as well as its mapping to the car’s hardware are part of the implementation. Basically, a complete timing-augmented system model as defined in Section 2.3.3 represents the system implementation.

The resulting implementation’s timing behavior must be correct with respect to the function-triggered timing constraints. Thus the properties itself must be chosen with respect to these constraints. Timing properties are implementation details and shall not be the target of such invariant, global timing constraints. In other words, the timing constraints in our methodology (see Section 5.5) are not implementation-driven but, as mentioned before, function-triggered. Two examples of function-triggered timing constraints are given in Example 3.6.

Example 3.6. The damper of a car must always be actuated not later than 25 milliseconds after a sensor has read the sensor input. The sensor input must be read every 10 milliseconds.

3.5.3 Function-triggered Timing Constraint Types

As shown in Section 3.5.1 there are many different types and semantics of timing constraints in real-time systems literature. Basically, such constraints can be divided into system level and component level timing constraints. In Section 3.5.2 we introduced the definition of function-triggered timing constraints. On the implementation-independent level of an automotive function we identified three typical timing constraint types. These three types cover most timing constraints for automotive functions in practice.

In our work, we support the following three types of function-triggered timing constraints:

Latency Constraint

A latency constraint constrains the latency between two successive observable events. Given an event chain with a stimulus event and a response event, a latency constraint refers to this event chain. The stimulus and response events must be "end events" of a function, for example external sensor and actuator events, as defined in Section 3.4.

Example 3.7. From reading the acceleration sensor until the execution of the damper actuator there must be a maximum latency of 10 milliseconds for the damper control function.

Triggering Constraint

A triggering constraint constrains the occurrence behavior of an observable event. If the targeted event is the stimulus of an event chain that models the end-to-end path of a function, then the triggering constraint constrains the occurrence behavior of that function.

Example 3.8. The damper control function must be triggered with a period of 10 milliseconds, i. e. it must be executed every 10 milliseconds.

Synchronization Constraint

A synchronization constraint relates the temporal synchronicity of several observable events. Using this kind of function-triggered timing constraint, several event chains can be synchronized, either on their stimulus or their response event. Synchronization is useful, if several event chains represent a function in the system.

Example 3.9. All four dampers of the car have to react synchronously, after the common stimulus sensor event has been read.

As these examples illustrate, the concept of observable events as introduced in Section 3.2.3 is essential for our timing model. In Section 5.3 we formalize our timing model and further detail the usage of such events.

3.6 Requirements and Guarantees

In our work we concentrate on the fact that several different teams develop one automotive real-time system, i. e. it is developed in a distributed development process. This fact induces some interesting challenges regarding timing constraints:

1. End-to-end timing constraints, which in our case are function-triggered timing constraints, are initially defined disregarding information which team is responsible for which part of the system. Mostly, such team allocations of subsystems are not even known when timing constraints are defined.
2. Every team focuses on the development of its own subsystem. A team eventually does not know the complete implementation context of its subsystem. For example the development of an ECU must start, before the rest of the network is completely known, especially before the network scheduling is completely implemented.
3. Often, not all implementation details of a subsystem are given to the system designer due to protection of intellectual property (IP). However, from a timing viewpoint it is necessary to reveal at least the timing behavior of the subsystem. It must be possible to do this without violation of IP protection.
4. The subsystem specifications must include a clear specification of its desired timing behavior, i. e. subsystem specific timing constraints.
5. The desired timing behavior of a subsystem depends on the timing behavior of its context. If for whatever reasons the context timing behavior changes, the subsystem's desired timing behavior may change, too.
6. The system designer is the only one who knows the complete system context. Thus, this role must manage the system's timing behavior as a central instance.
7. The implementation of a subsystem, done by the according team, leads to a certain timing behavior. This behavior must be reported by the team to the system designer, who integrates all subsystems.

To provide a solution for all these challenges enumerated above, a concept for subsystem specific timing constraints is necessary. We call such subsystem specific timing constraints *timing requirements*.

Definition 3.10. *A timing requirement is a subsystem specific timing constraint that constrains the timing behavior of the sub-function that is implemented by the subsystem. Timing requirements are derived from function-triggered timing constraints. Timing requirements can vary during development when the subsystem's context timing behavior changes.*

We use timing requirements for subsystems as basis for a solution to challenges 1 to 6 in the list above. The solution details are discussed in Chapter 5. To report a subsystem's timing behavior back to the system designer, we use the concept of *timing guarantees*.

Definition 3.11. *Every timing requirement is accompanied by its according timing guarantee, which must fulfill the requirement.*

Our concept of requirements and guarantees has similarities with a general approach called *contract-based design* [64]. In computer science this is an approach to develop software systems based on a component model. The interfaces of components are precisely described in a formal and verifiable way. For different aspects of functional or non-functional behavior components give assertions based on assumptions about the environment. This creates a *contract* between components. Brunel et al. [16] proposes a contract-based design approach motivated by an automotive example. Safety and timing properties (latency, triggering) can be expressed in a language similar to Linear Temporal Logic. The approach however considers neither function-triggered timing constraints nor an iterative finding of appropriate requirements as proposed by our work. Rich Components of Damm et al. [22] are another concept to add non-functional requirements to component models. The difference of our timing requirement/guarantee approach is that requirements and guarantees always are a pair that is valid for a data path through a subsystem. A guarantee is used to give feedback to the system designer about the current timing behavior of that subsystem. The other subsystems however do not use that guarantee as an assertion (like in contract-based design). All subsystem developers only take their requirements as input, which must be fulfilled. Requirements and guarantees are isolated for the subsystems.

3.7 Summary and Comparison of Timing Constraint Types

3.7.1 Summary of Related Work on Timing Constraints

Table 3.1 summarizes the related work on timing constraint types discussed in Section 3.5.1. The related work is categorized by two dimensions. The first one is the scope of the timing constraints used in the according model, either system level or component level. The second dimension is the type of real-time system that is supported by the approach, either event-triggered or time-triggered systems.

System type \ Constraint scope	Event-triggered	Time-triggered
Component level	Liu and Layland [59] Audsley et al. [3] Marques et al. [61] Saket and Navet [78]	Chung and Dietz [21] Gerber et al. [37] Saksena [79]
System level	Richter et al. [74, 49] Feiertag et al. [29]	Schild and Würtz [84]

Table 3.1. Categorization of timing constraints in related work.

Table 3.1 should make two things clear. All related work focuses on one specific system type. Timing constraints are always defined for and used in either an event-triggered or a time-triggered system. Second, if system level constraints are used, this is done to directly derive component level constraints (or timing properties for components) from them.

3.7.2 Comparison of Constraint Types

In this section we finally compare related work on timing constraints with our concept of function-triggered timing constraints and timing requirements and guarantees for development teams. Figure 3.15 depicts these types. The figure orders constraints by their level of abstraction, ranging from implementation specific (constraints for actual timing properties) to very abstract performance requirements. A similar constraint ordering was already discussed in Figure 3.14, which is borrowed from [79].

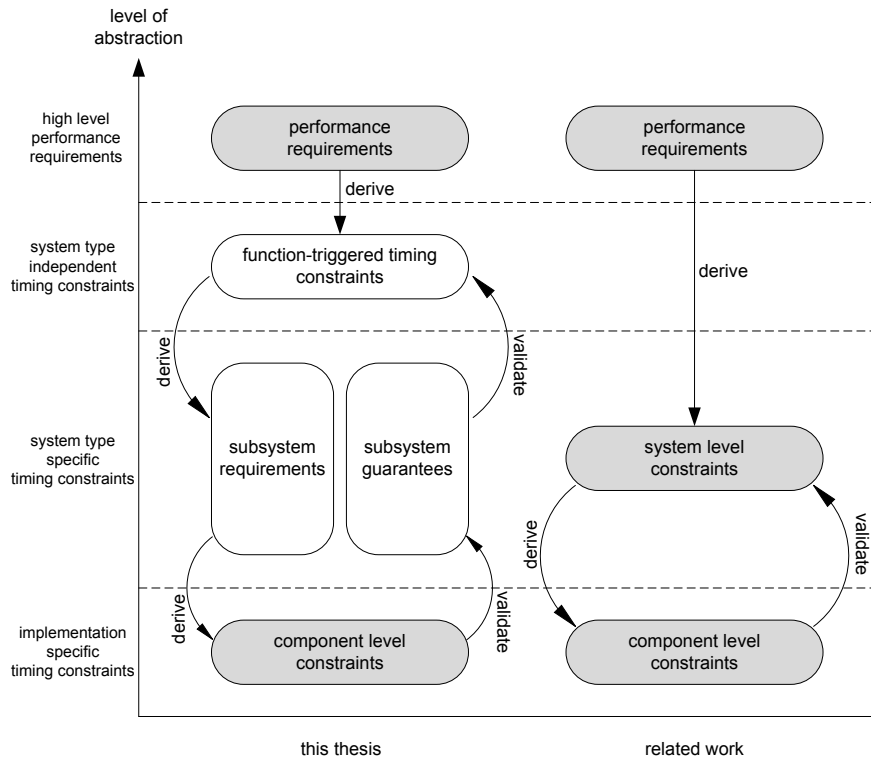


Fig. 3.15. Comparison of timing constraints in this thesis and in related work.

The approach proposed in this thesis differs from the related work as follows. First, function-triggered timing constraints are declared as implementation independent end-to-end constraints. That means, they abstract from all implementation details of a function, especially the system type. Our work explicitly targets time-triggered and event-triggered real-time systems.

Second, our approach introduces the new intermediate level of timing constraints, which we call timing requirements and timing guarantees, as introduced in Section 3.6. These are derived from the function-triggered timing constraints and belong to the responsibility of one development team. Whereas related work concentrates on the derivation of component level constraints (i. e. for timing properties) from system level timing constraints, our approach introduces the concept of requirements and guarantees.

The Global Scheduling Problem

In this chapter we show how the scheduling of a system can be developed using the techniques that current real-time literature offers. Developing the scheduling of a system means to define, or configure, all timing properties of the system. The result of this process is a specific timing behavior, which must fulfill the system's timing constraints.

We describe a typical system timing development and some assumptions that we make for such systems. The so-called global scheduling problem is described both for an event-triggered and for a time-triggered system. After that we explain why the standard global scheduling approaches are not sufficient for a distributed development environment, which is typical for the automotive industry as described in detail in Section 3.1.

4.1 Usage of our System Model

4.1.1 Assumptions

In Section 3.3 we already introduced our system architecture model. The description of the global scheduling problem presented in this chapter is also based on that system model. The system model is part of the overall *timing-augmented system model*. It consists of several sub-models, which represent system model information on different abstraction layers. For the global scheduling problem we focus on the technical architecture model. We assume the functional architecture as given, because the available functionality of the car is already known. Also the software architecture model, i. e. the realization of the functionality by means of software components, is assumed to be given. In the technical architecture the hardware topology is also assumed to be completely available. That means it is not in the scope of our global scheduling problem to find an optimal hardware design.

During system development in a typical automotive design flow as described in Section 2.2, the mapping of the software components onto the available hardware is a very complex step. The outcome of this step is the deployment model. The deployment model influences the system's timing behavior because the software component mapping defines which communication paths

between components are local or remote communication. Especially remote communication over a bus network can lead to considerable high time consumption and depends on the schedule design. For the description of the global scheduling problem, we take a deployment model as a basis. So there are two remaining sub-models, which span the *configuration space* of the problem:

- execution model, which defines the mapping of runnables to tasks and the scheduling of tasks
- communication model, which defines the mapping of signals to frames and the scheduling of frames

Assumption 4.1 *For the global scheduling problem as well as for our approach to improve distributed development, the functional architecture and logical software architecture models are assumed to be given. Further the hardware topology and deployment model are assumed to be given. The global scheduling is influenced by the design decisions made in the communication model and execution model.*

Developing a deployment model is often referred to as *allocation*. Tindell et al. [94] developed a method to allocate tasks to processors. Given a set of tasks with mutual dependencies. That means one task can consume data produced by another task. The method assigns deadlines to the tasks depending on the kind of task communication, which is either local or remote. Locally communicating tasks are assigned a deadline equal to their period. Remotely communicating tasks have shorter deadlines, because the message produces an additional sending delay. The approach is somewhat different to our system understanding, because the authors assume the task mapping to be performed before allocation. Nevertheless they show an interesting approach to the allocation problem, which we explicitly do not consider in our work.

According to our system architecture model (see Section 3.3) software components of the logical architecture are refined on the technical architecture using runnables. These runnables actually are part of the execution model. So it is special engineering work to define the set of runnables for each software component. However for the global scheduling problem we assume that the set of runnables is already given. The definition of the execution model here means defining tasks, assigning the runnables, and finding appropriate timing properties for the tasks.

Concluding, we state that task and communication scheduling have the greatest influence on the timing behavior of an automotive real-time system.

The vehicle electrical system of modern cars consists of several sub-networks that are interconnected with gateway components. The different networks potentially have a different system type, i. e. they are event-triggered or time-triggered. Gateway components must be able to interlink all these different types to enable cross-network communication. Each of these networks in a car often is dedicated to a specific domain, e. g. solely for chassis or solely for body electronics. Therefore sub-networks are often also called *clusters*. Each cluster is realized using one specific system type, because the domains have different demands with respect to real-time behavior and criticality. A

chassis network typically requires a time-triggered system, because chassis functions are classical closed loop control systems and have to fulfill hard real-time constraints. A body domain network typically is realized using an event-triggered approach, because it mainly contains driver-triggered comfort functions with predictive triggering behavior.

In our work we concentrate on the differentiation of the two main system types, which are event-triggered or time-triggered systems. We show how function-triggered timing constraints are translated into requirements for subsystems, depending on the system type. Therefore, it is not necessary to include gateways, because the concepts for each system type can easily be combined for more complex networks.

Assumption 4.2 *A system in our context has one of the two main types, event-triggered or time-triggered. A system has only one communication bus that connects all ECUs.*

Gateways are not in our scope and a system in our meaning is what is usually referred to as cluster.

In this work we distinguish local and remote communication between software components. Local communication is technically realized with shared variables or local function calls. Basically, shared variables could also be used for inter processor communication on one ECU. However, we exclude such multiprocessor ECUs from our considerations.

Assumption 4.3 *We assume that each ECU has exactly one processor that executes the software. Each ECU thus needs one processor schedule for the task set it executes.*

Concluding, global scheduling means to develop exactly one task schedule per ECU and a communication schedule for the one available bus. The bus schedule is captured in the communication model, which carries all frames and their according timing properties. The processor schedules in our system model are merged in one system wide execution model, which contains the tasks of *all* processors. In practice, such an execution model would probably be split by the available ECUs.

4.1.2 Timing Properties

The available timing properties depend on the system type. For both supported types special timing properties in the communication model and execution model exist for tasks and frames. Our system model contains all these timing properties.

The mapping of signals to frames is realized by the *signals* function in our system model. Messages are directly mapped to bus frames and then called signals. In practice often the intermediate container concept called *PDU* is part of the communication model. That is, signals are grouped to PDUs, which

in turn are mapped to frames. The concept of a PDU (protocol data unit) stems from the layered network architecture models. Simply put, a PDU adds network layer and communication protocol specific data to a group of signals. AUTOSAR for example supports three such layers in its communication stack, which are represented by signals, PDUs and frames. PDUs there are groups of remotely exchanged signals with an abstraction of the concrete bus type. The concept of the PDU adds additional complexity to the entire communication configuration process, which we neglect in our work.

Assumption 4.4 *In our communication model, signals are directly mapped to bus frames, without the intermediate concept of PDUs (protocol data units).*

4.1.3 System Model Example

We presented an example system model in Section 3.3.6 using our formalism to describe all six sub-models. For the global scheduling problem description we now focus on the two basic sub-models execution model and communication model of that example. In the initial example we added random values for the basic timing properties of the system's tasks and frames just to demonstrate the formalism.

In this chapter we now show, which techniques from the literature can be used to a) determine real timing property values of frames and tasks and b) how the timing behavior of the system, which is induced by these timing properties, can be analyzed.

For the global scheduling problem we assume variables instead of random values for the main timing properties of the example model. The output of global scheduling is an appropriate instantiation of these variables with respect to the timing constraints, which must be defined for the realized function.

In addition to these timing properties, also the mapping relations *signals* and *runnables* must be defined as part of the execution and communication models. In the example system model of Section 3.3.6 there are two frames and one task per ECU, which we also chose randomly. Actually the definition of these mapping relations is part of the global scheduling problem.

We sum up all steps that are necessary to be performed in the global scheduling problem in Section 4.3.

4.2 Timing Constraints

To point up the global scheduling problem, the system also needs to have some timing constraints that it must fulfill. The system model only contains timing properties in the communication and execution models. As we show the problem of global scheduling for a system, all timing constraints are considered as system level timing constraints, as described in the literature survey in Section 3.5.1. The timing properties must lead to a timing behavior, which fulfills the constraints.

Several types of possible timing constraints exist, as we described in Section 3.5.1. To demonstrate the global scheduling problem, a type of system level constraint is appropriate, which involves the entire system model example. Therefore the most typical type is a latency constraint for a function, which also is the most commonly used timing constraint in other scheduling and timing analysis literature.

For the purpose of this chapter, an informal description of the timing constraints is sufficient. We developed a timing model for a formal specification of function-triggered timing constraints, which we present in Chapter 5 based on the results of the analysis of the global scheduling problem, especially with a focus on distributed development.

4.3 Global Scheduling Problem Description

4.3.1 Definition of Global Scheduling

Based on a system model as described in Section 3.3 and the assumptions made in Section 4.1.1 we now formulate the single steps of the *Global Scheduling Problem*. Therefore, a definition of global scheduling is required.

Definition 4.1. *Given a system model that consists of a functional architecture model, a software architecture model, a hardware topology model, and a deployment model, Global Scheduling is the process of finding both the execution model of each ECU and the communication model of the bus network such that global, i. e. system-wide, end-to-end timing constraints are fulfilled. Additionally also basic schedulability requirements must be fulfilled, which are independent from timing constraints.*

The origin of the global scheduling problem is the distributed character of automotive networks. Modern car functions are distributed over several ECUs (see Section 2.2). Timing constraints on a function level thus influence the consideration of the timing behavior of the whole system, not only of single components like a single ECU or only the bus. The schedules of these components automatically get coupled, i. e. changes of one components schedule can influence the system's timing behavior and thus induce the necessity of a different schedule for another component. We will describe this in more detail for event-triggered systems in Section 4.5 and for time-triggered systems in Section 4.6. These observations lead to Assumption 4.5.

Assumption 4.5 *In a distributed system, the timing property configurations of the system's components (ECUs and bus) are coupled because they jointly influence the system's timing behavior.*

The result of the global scheduling process as defined above is a complete timing-augmented system model as described in Section 2.3.3. The ECU execution models, which are merged in the one execution model of our system

architecture model, and the bus communication model comprise all timing properties of the system model. As described in Section 2.1.3, these timing properties influence the system’s timing behavior to a great extent, besides the deployment model, which we assume as given according to Assumption 4.1. The timing behavior in turn must fulfill the given timing constraints. This correlation is depicted in Figure 4.1.

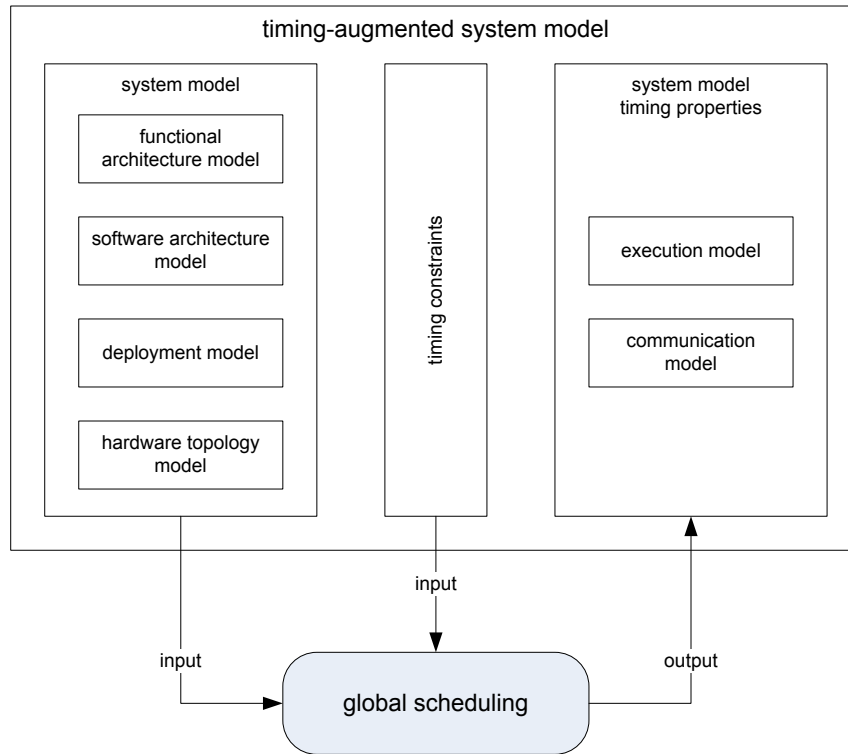


Fig. 4.1. Timing-augmented system model as input and output of global scheduling.

Timing properties have been defined as part of the system implementation in Section 2.1.3. Global scheduling thus is used to accomplish the overall system implementation to a complete timing-augmented system model and to find appropriate timing properties. This part of the system implementation is also referred to as *timing configuration*. Global scheduling is the process of finding a timing configuration for a system to accomplish the timing-augmented system model in a way that timing constraints are fulfilled.

Global Scheduling can be automated, as we will see at the related work approaches we analyze later in this chapter. According to Assumption 4.1 it is the scope of our work to support the development of such a timing configuration that fulfills all timing constraints. It is not in the scope to find an appropriate deployment, or even to automatically define a software architecture. Early work on deploying software to hardware, often also called *allocation*, was presented by Tindell et al. [94]. We already outlined their approach in Section 4.1.1.

Before we explain global scheduling in detail we make one temporary assumption for the analysis of the problem. For the analysis of the global scheduling problem we neglect roles, teams and the subsystems they account for. So, Assumption 4.6 particularly means that no typical distributed development limitations are induced.

Assumption 4.6 *For the global scheduling related work analysis, we assume that the system is completely developed by one team and we temporarily neglect distributed development limitations.*

After we explained related work approaches to all or some steps of the global scheduling problem in Section 4.5 and Section 4.6, we point out where distributed development limitations take effect in these approaches in Section 4.7. That is why Assumption 4.6 is not admissible and just temporarily valid to explain the other approaches. In the automotive practical systems engineering all these limitations exist. The related work however does not take those into account. We show our approach and solution to these limitations in Chapter 5 and Chapter 6.

4.3.2 Global Scheduling Steps

Regardless of the system type that is actually used, several abstract global scheduling steps have to be performed. According to Assumption 4.1 we define the following steps, which make up the global scheduling process based on our system architecture model.

1. *Task Mapping* is the step of mapping a given set of runnables to a set of operating system tasks. The deployment model defines, which software components belong to which ECU. Because of Assumption 4.3 all runnables of the software components of one ECU are executed by one processor. To be executed by the operating system, the runnables must be contained in tasks. Task mapping is performed for each ECU. In our system model the execution model collects all task mappings of all ECUs. The deployment model implicitly gives the allocation of tasks to processors. One common attribute of both task types in our system model is the task WCET. It is the sum of the WCETs of the contained runnables.
2. *Task Scheduling* is the step of assigning appropriate timing properties to the tasks. The concrete timing properties of the tasks depend on the system type, i. e. on the task type, which is either an *ETTask* or a *TTTask* in our system model (see Section 3.3.5). Task mapping is also actually performed per ECU, although the one execution model here collectively captures all tasks.
3. *Frame Mapping* is the step of mapping signals to frames. The deployment model defines which connectors between software components are remote connectors, namely the ones where sending and receiving SWC are deployed to different ECUs. Exchanged messages of remote connectors are represented by a signal. Actually the concept of signals only exists implicitly in our system model by referencing a message from a frame. The

remote transmission of signals is realized using frames. Frame mapping is performed for the system's communication bus to be able to transmit signals. According to Assumption 4.2 we assume only one communication bus in our network. One common attribute of both frame types in our system model is the frame size. Because we assume the same size of all signals in a frame, the frame size actually is the maximum number of contained signals.

4. *Frame Scheduling* is the step of assigning appropriate timing properties to bus frames. Again, the concrete timing properties depend on the system type. We call the two available frame types *ETFrame* and *TTFFrame* in our system model (see Section 3.3.5). The communication model holds all frames and thus contains all timing properties of the one available bus.
5. After steps 1 to 4 the system model is completed and all timing properties are set. *Timing Analysis and Verification* is the next step to check the fulfillment of timing constraints of the timing-augmented system model. Especially in the context of system level timing constraints, a so-called timing analysis must be performed based on the timing-augmented system model. On component level some available scheduling approaches already comprise the fulfillment of task or frame level timing constraints, as we will explain in the next sections. System level constraints are usually verified by timing analysis.

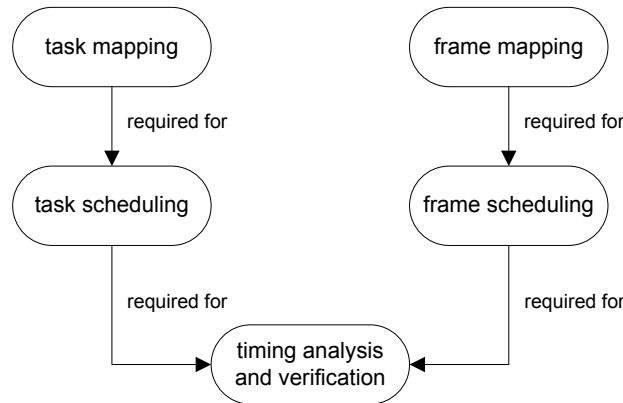


Fig. 4.2. Dependency of the five global scheduling steps.

The dependency of all five steps is shown in Figure 4.2. The result of these steps is a complete system model according to Definition 4.1 and as depicted in Figure 4.1. We describe the concrete, system type dependent realizations of each of these steps in Section 4.5 and Section 4.6 and we provide an overview of related work, which covers these steps.

4.4 Task Mapping and Frame Mapping Approaches

Before all standard scheduling and timing analysis techniques can be applied to our system model, task mapping and frame mapping must be performed. The basic software execution entities of our model are runnables. All approaches however expect tasks to be scheduled. The same applies for signals and frames.

A very simple approach to task and frame mapping would be to define every runnable as a task and every signal as a frame and then proceed with the task and frame scheduling steps. This approach however has some practical disadvantages:

- The amount of tasks that an operating system can handle is often limited. The software complexity of real automotive systems would exceed such upper bounds and thus engineers are forced to group runnables to tasks. Similar, the amount of remotely exchanged signals in practice is far beyond the amount of configurable frames on a typical bus system.
- At runtime, task switching produces time and memory overhead for the operating system. The state of a task must be stored when it is preempted to be able to continue its execution later. On a bus network every frame carries protocol information additionally to the actual data. For that reason the amount of data that is exchanged on the bus is more than the actual signal data. Signal groups that are mapped to a frame produce less data overhead.
- After task and frame mapping the according scheduling steps follow. All task or frame timing properties must be assigned. The complexity of such scheduling approaches depends on the amount of tasks or frames to be scheduled.

Because of these three reasons, engineers need a sophisticated method of mapping runnable entities to tasks and signals to frames. Some simple practical rules are obvious and can be applied by engineers both for event-triggered and time-triggered systems:

- Periodically executed runnables and periodically transmitted signals can only be assigned to the same task, if the smallest period is a divider of all other periods, i.e. the periods must be harmonic (see Definition 2.9).
- If the execution of a runnable B depends on the execution of another runnable A, then A should be scheduled before B within the task.
- Runnables with shared memory access should be mapped to the same task to avoid race conditions.

Frame mapping has attracted only a little attention in real-time systems literature [78]. In the following we outline one approach for event-triggered systems.

Navet et al. [78, 61] developed a method for frame mapping (they call it frame packing) of a priority-based communication network such as CAN. The prob-

lem they solve with their approach is the following. They assume a complete system with several ECUs and one bus. Every ECU sends signals over the bus. Every signal has the three given timing properties size, production period and a deadline. The approach seeks for a frame mapping such that all signal deadline constraints are met. A period and a priority is assigned to the frames, i. e. the approach also performs the frame scheduling step. Additionally, the approach aims for a minimization of the bandwidth consumption, which is an issue of frame mapping in general as described above.

Approach	Step	Input	Output	Conditions
Navet et al. Frame Packing	frame mapping and scheduling	signals: period deadline size	frames: period priority	<ul style="list-style-type: none"> signal deadlines known

Table 4.1. Frame Packing according to Navet et al. [78, 61].

Table 4.1 summarizes the input and output of the frame mapping approach by Navet et al. [78, 61]. All output frame properties are also available in our communication model. However, we do not explicitly provide such properties for signals, because we only assume timing requirements for signals, like requirements for signal periods and offsets. The system model does not contain such timing requirements. These are covered by our new TIMEX timing model presented in Chapter 5.

4.5 Scheduling Approaches for Event-triggered Systems

In this section we analyze related work approaches to the scheduling and analysis of event-triggered systems. According to Assumption 4.6 we temporarily neglect the fact that automotive systems are developed in a collaboration of several teams and apply the concepts and approaches to the global scheduling problem of our system model as defined in Definition 4.1. All related work approaches can be grouped in component level and system level approaches (see Section 3.5.1). We consider both types in this survey.

4.5.1 Rate Monotonic Scheduling

In their fundamental contribution to scheduling analysis, Liu and Layland [59] proposed Rate Monotonic Scheduling (RMS) and proofed that it is a so-called optimal scheduling approach for assigning static priorities to tasks. Optimality in this sense means that if RMS does not find a scheduling solution then no other scheduling approach will find a solution as well. Further, no other priority assignment strategy can yield better schedulability.

To work properly RMS defines some assumptions for the task model:

- Tasks are strictly periodic.

- Tasks have a fixed WCET.
- All tasks are preemptive.
- A task's deadline is equal to its period.

The assignment strategy of RMS is simple. RMS assigns the task priorities according their periods. The task with the smallest period (i. e. the highest rate) is assigned the highest priority. Given the above assumptions RMS is optimal. The authors proofed that such a system is schedulable if the processor utilization (see Equation (2.1)) is below a certain bound. The bound depends on the number of tasks. Lehoczky et al. [57], Sha et al. [86] as well as other researchers published less pessimistic and exact schedulability tests based on the processor utilization.

According to our classification made in Section 3.5.1, RMS is a component level scheduling approach. The component in that case is a single processor. The concept however does also fit for communication busses that also fulfill the model assumptions listed above.

Approach	Step	Input	Output	Conditions
Rate Monotonic Scheduling	task or frame scheduling	period deadline wcet	priority	• deadline equals period

Table 4.2. Rate Monotonic Scheduling by Liu and Layland [59].

In Table 4.2 we summarize the input and output of RMS. Both our communication model and our execution model (Section 3.3) provide these properties. An *ETTask* has got the *wcet* and *period* attributes, which are the input parameters of RMS. Further it has the *priority* attribute, which is the output according to RMS. The same applies for the attributes of an *ETFrame*. Also the task deadline is implicitly covered by our model, because of the assumption that the task deadline is equal to its period. However, the deadline assumption of RMS is the most restrictive one. We will discuss deadline limitations of RMS and other approaches in Section 4.7.1.

4.5.2 Response Time Calculation

Joseph and Pandya [51] published a response-time calculation based on RMS. The worst case response time (WCRT) of a task is the actual time it takes to execute a task, including potential interruptions by other tasks with higher priorities. The authors based the response time calculation on the concept of the *critical instant*. This is the case when all tasks are activated at the same time. The authors showed that the critical instant leads to the worst case response time of all tasks and presented a recursive calculation formula.

We summarize the input and output of the response time calculation according to Joseph and Pandya [51] in Table 4.3. The input is all worst case execution times, priorities and periods of the tasks, the output is the worst case response

Approach	Step	Input	Output	Conditions
RMS Resposne Times	timing analysis	period priority wcet	wcrt	<ul style="list-style-type: none"> deadline equals period

Table 4.3. Response time calculation based on Rate Monotonic Scheduling, Joseph and Pandya [51].

time of each task. We call the worst case response time a *scheduling analysis result* that abstracts the underlying timing properties, which were input to the approach.

Tindell et al. [95] developed a method to calculate response times for CAN messages. The approach is similar to the one of Joseph and Pandya [51] but takes CAN specific aspects into account, such as message transmission times, message jitter and message blocking times. The result also is a guaranteed WCRT for all CAN messages.

4.5.3 Deadline Monotonic Scheduling

The basic restriction of the RMS approach is that every task has a deadline that is equal to its period. RMS soon disadvantages especially tasks with low frequency, because they probably get a very low priority. After the initial publication of RMS several researchers improved the RMS concept to also allow task deadlines smaller than and greater than the task periods.

Audsley et al. [1, 3, 2] relaxed the "deadline equals period restriction" and also allowed deadlines smaller than the task periods. According to this priority assignment strategy, which is called Deadline Monotonic Scheduling (DMS), the task with the shortest deadline gets the highest priority. The authors proved that also DMS is an optimal scheduling strategy. Leung and Whitehead [58] proofed that the response time formula of Joseph and Pandya [51] also applies for arbitrary deadlines and developed enhanced schedulability tests for DMS. However, the focus of the timing constraints is still just on the level of a single task or, if the concept is applied to a communication bus with a similar model, a single frame.

Allowing deadlines greater than task periods was investigated by Lehoczky [56, 57]. The basic consequence of this task model is that a new task execution can be triggered before the last one has finished. This leads to multiple task instances. Lehoczky introduced the concept of a *busy period*. This concept generalizes the principle of a critical instant described in Section 4.5.2 by using an extended response time calculation formula.

Approach	Step	Input	Output	Conditions
Deadline Monotonic Scheduling	task or frame scheduling	period deadline wcet	priority wcrt	<ul style="list-style-type: none"> deadline known

Table 4.4. Deadline Monotonic Scheduling, Audsley et al. [1, 3, 2].

The input and output of DMS, summarized in Table 4.4, is the same as we already explained for RMS. The deadline restriction of RMS does not exist for DMS. The necessary input and output attributes of *ETTask* and *ETFrame* are also available in our system model except WCRT, which we defined as timing analysis result.

4.5.4 Compositional Scheduling Analysis

Richter et al. [74, 49] developed a system level timing analysis method called Compositional Scheduling Analysis. It is based on event model interfaces between system components. Components are tasks and frames. The method enables verification of and reasoning about end-to-end paths through an event-triggered system. Basically, the approach does timing analysis, i. e. all component level timing properties for tasks and frames like priorities, periods, etc. are assumed to be given. The analysis output is the worst case response time of paths, i. e. maximum path latencies and jitters. The result can be used to answer the question whether end-to-end latency constraints are fulfilled.

The authors developed a method to couple event model streams using event model interfaces. Communicating components are coupled using a certain set of defined event models. This leads to a structured and modularized analysis technique that decouples complex global dependencies.

The analysis model of this approach captures the system level interactions in simple event models, which are attached to the output and input "port" of every component (task or frame). These event models interface the components. A path in the system basically is a sequence of components that trigger each other. For example, an initial task is triggered (e. g. by the operating system or an external interrupt). It sends a message over a frame and thus triggers the frame transmission. Thereby the task induces an event model to the frame, because they are connected with an event interface. At the end of the path a final output event model results from the initial input event model. On the way through the path, a lot of local scheduling effects in the particular components (on an ECU or the bus) influence the transformation from the input to the output event model. The final output event model and the latency of the path represent the analysis result.

Approach	Step	Input	Output	Conditions
Compositional Scheduling Analysis	timing analysis	priority period wcet offset	path WCRT, output event model	<ul style="list-style-type: none"> all timing properties known

Table 4.5. Compositional Scheduling Analysis according to Richter et al. [74, 49].

Table 4.5 summarizes the input and output of the Compositional Scheduling Analysis approach. The initial event models are based on the standard attributes for tasks and frames that are also available in our system model. If all task and frame attributes are given, the dynamics of local and global scheduling effects along paths can be computed with Compositional Scheduling Analysis. The analysis output can be compared to given timing constraints.

4.5.5 Generating Intermediate Task Deadlines

Gerber et al. [37] presented an approach to derive intermediate task level properties and constraints from given system level constraints (we describe their constraint types in more detail in Section 3.5.1). The scope of their approach is a single ECU without remote communication over a bus. The tasks of the ECU are represented as a so-called task graph. A path through the graph represents an end-to-end sensor actuator dependency. These dependencies can have three kinds of timing constraints, namely latency, input rate and value update jitter for inter-task communication.

In their publication Gerber et al. present an automated design method based on intermediate timing properties. These properties are task periods, offsets and deadlines. Their constraint logic programming approach is used to generate the resulting task set and check a) that the task set is feasible with respect to the processor utilization and b) the intermediate timing properties fulfill the end-to-end constraints. Based on these properties they use other fixed priority preemptive scheduling (FPPS) techniques to check whether a priority-based schedule can be found with the generated properties that fulfills all task constraints. Such other techniques are for example RMS or DMS, which we described in Section 4.5.1 and Section 4.5.3, respectively. If no solution can be found with the current intermediate properties then the algorithm tries to identify potential bottlenecks and generates new periods.

The result of the algorithm is a schedulable set of tasks, which minimizes processor utilization and fulfills all end-to-end constraints.

Approach	Step	Input	Output	Conditions
Gerber et al. Intermediate Task Deadlines	task scheduling and timing analysis	path: deadline rate jitter	task: deadline, offset period	<ul style="list-style-type: none"> • only one ECU • all properties accessible

Table 4.6. Generating intermediate task deadlines according to Gerber et al. [37].

In Table 4.6 we summarize the approach of Gerber et al. The approach is interesting to mention because of the fact that it is somehow related to our approach of generating subsystem timing requirements based on end-to-end constraints, which we call function-triggered timing constraints (see Section 6). However it is limited compared to our work in some respects. First the authors only support one priority-driven ECU. Second their approach searches exhaustively for a global solution, whereas our approach tries to solve timing conflicts locally as far as possible. Finally Gerber et al. assume to have full access and insight to the entire system timing properties. Our approach solely works on the abstraction layer of subsystem timing requirements and guarantees, without deeper insight into the underlying timing properties.

4.6 Scheduling Approaches for Time-triggered Systems

In the time triggered paradigm the entire distributed system typically has a common global time line and all task activations and frame transmissions are triggered at certain predefined points in time. Scheduling time-triggered systems is often also referred to as schedule generation. The challenge of schedule generation is to find a suitable line up of all cyclic tasks and frames using appropriate task and frame offsets from a common reference point in time. This schedule must be in a way, so that every single task has exclusive processor resource access during his execution time and every frame is transmitted on the bus without interruption. The so-called non-preemptive static task schedule has to be developed per ECU and for the bus. As described in Section 2.1.4 finding such a static schedule is an NP-complete problem.

As additional challenge the tasks and frames potentially have to meet deadlines that can be derived from different dependencies amongst the tasks. Mostly the deadlines are derived from end-to-end constraints. The main challenge of schedule generation for static time-triggered systems is that the ECU and bus schedules are tightly coupled with each other (see Section 2.1.4). Thus the effort of schedule generation rises drastically with the number of interconnected remotely communicating tasks.

4.6.1 Frame Mapping and Scheduling

Grenier et al. [38] developed a frame mapping and scheduling approach for time-triggered systems. They target the configuration of the static part of a FlexRay schedule. The authors assume some given timing properties for all signals in the network, which are the production period, offset, size, and deadline. The deadline is the maximal age a signal must have from its production by a sender until it is received. The authors present an algorithm that takes these signal timing properties as input and calculates both an according frame mapping and frame scheduling according to the FlexRay technology. The resulting static schedule leads to a maximum age of every signal.

Approach	Step	Input	Output	Conditions
Grenier et al. Frame Scheduling	frame mapping and scheduling	signals: period offset size deadline	frames: period offset signals age	<ul style="list-style-type: none"> • signal production offset given • signal deadline known

Table 4.7. Frame scheduling according to Grenier et al. [38].

Table 4.7 summarizes the necessary input and output of the approach. Similar to the approach of Navet et al. , which we described in Section 4.4, our system model covers only the frame timing properties, not the signal timing properties.

4.6.2 Frame Scheduling

Nossal and Galla [68] also developed an approach to solve the scheduling problem for one time-triggered bus. Their approach uses a genetic algorithm to obtain a feasible static message schedule for a synchronous communication bus. The approach does not directly target an automotive network but a real-time LAN. The basic concept of that network however is similar to other time-triggered networks, like the static part of the automotive-specific FlexRay bus. Every signal, which they call message, has got a maximum update period, i. e. a maximum sending frequency. The algorithm searches an assignment of these messages to bus slots, such that all messages are transmitted with their required period. Thereby a lot of side conditions have to be met, for example to avoid interference of messages with different periods. We explained such time-triggered specific side conditions along with the idea of cycle harmony in Definition 2.9. A bus slot in the approach of Nossal and Galla actually can also be expressed as an offset from the bus cycle start, similar to our system model. The authors neglect frame mapping and thus assume every message to be mapped to an own frame.

Approach	Step	Input	Output	Conditions
Nossal and Galla Message Scheduling	frame scheduling	period	offset	<ul style="list-style-type: none"> no signal deadlines supported

Table 4.8. Message scheduling according to Nossal and Galla [68].

Table 4.8 summarizes the approach of Nossal and Galla. Note that no deadline constraints are considered by the approach. The output offsets solely are used to schedule all signals with their given period such that no signal collisions occur.

4.6.3 Instruction Precedence Scheduling

Chung and Dietz [21] consider different kinds of precedence constraints of single instructions. The approach however can also be extended to tasks precedence constraints. These constraints solely set up a temporal order of instructions by expressing "before" and "after" relations for pairs of instructions. They propose an adaptive genetic algorithm to schedule all instructions such that all these constraints are fulfilled. Therefore the initial precedence constraints are transformed into standardized timing constraints of the form "instruction X must occur after instruction Y with an offset of n ". Note that these offset values are relative, because they are defined for one instruction relating to another instruction.

If the problem consists of n instructions there are $n!$ potential schedules, i. e. sequences of instructions. Only a few of them fulfill all precedence constraints and finding those is an NP-complete problem [21]. Therefore the authors use a genetic algorithm to find one of these solutions. According to our system

Approach	Step	Input	Output	Conditions
Chung and Dietz Instruction Precedence Scheduling	task scheduling	relative instruction offsets	absolute instruction offsets	<ul style="list-style-type: none"> • no periods • no deadline constraints

Table 4.9. Instruction precedence scheduling according to Chung and Dietz [21].

model, the resulting sequences actually are absolute offset values for instructions. The authors do not consider periods of the instructions. They obviously assume the same period of all instructions. Table 4.9 summarizes their approach and its input and output.

4.6.4 System Scheduling of Tasks and Messages

System level task and message scheduling for time-triggered systems with end-to-end timing constraints is considered by Schild and Würtz [84]. They assume a system consisting of several distributed processing units that are connected with exactly one synchronized communication bus. That means, the entire system shares one common time line.

The tasks executed by the processing units are organized as a task graph. End-to-end latency constraints express a maximum latency between one task's start time and another task's end time. Between these two tasks also bus communication and other intermediate tasks are allowed. The authors neglect frame mapping and directly schedule messages on the bus.

Approach	Step	Input	Output	Conditions
Schild and Würtz System Scheduling	task and frame scheduling	task wcet task period message size message period	task offset message offset	<ul style="list-style-type: none"> • system schedule generated as a whole

Table 4.10. System scheduling according to Schild and Würtz [84].

The authors use a constraint logic programming approach to generate appropriate task and frame timing properties such that given end-to-end constraints are fulfilled. Table 4.10 summarizes their approach. The authors assume both synchronous communication, where all tasks and messages along a path have the same frequency, and asynchronous communication, where so-called over- and undersampling occurs. The task and frame timing properties apply to our execution and communication model, except that the authors use a different terminology (e. g. start time instead of offset) and our communication model does not consider different signal sizes.

4.6.5 System Scheduling with Frame Mapping

Ding et al. [24] proposed a scheduling algorithm for time-triggered systems that explicitly targets automotive systems based on FlexRay. The authors assume to have the software system that has to be scheduled available as a task graph. Each node of the graph is one task and each edge is one message that is exchanged between the two connected tasks. The tasks initially are not allocated to one of the available processors. Task allocation is part of the overall problem. For our survey however we concentrate on the task and frame scheduling part of their approach.

Approach	Step	Input	Output	Conditions
Ding et al. System Scheduling	task and frame scheduling	task wcet task period task deadline message size message period	task offset frame offset	<ul style="list-style-type: none"> • system schedule generated as a whole • task deadlines given

Table 4.11. System scheduling according to Ding et al. [24].

Table 4.11 summarizes the system scheduling approach by Ding et al. [24]. The authors take several kinds of end-to-end latency and synchronicity constraints as input for their approach. Further the deadline, period and WCET of each task are known. Messages have a certain size and are mapped to frames, which in turn are scheduled in a certain FlexRay slot. Such slots are identified by their offset. Task and frame offsets are the output of their approach.

4.7 Limitations of Related Work

The approaches and the timing models that are used in the above-mentioned scheduling approached have some limitations in practice. The limitations basically stem from the fact that automotive systems are developed collaboratively. That is, our temporary Assumption 4.6 is not valid in practice. In the following sections we summarize these limitations, which are the motivation for our new timing model and methodology called TIMEX, which we describe in the subsequent chapter.

4.7.1 Derivation of Subsystem Constraints

Some of the approaches find feasible scheduling solutions based on given task or frame level deadlines. Navet et al. [78, 61] for example developed a sophisticated frame mapping approach where the signals must have signal deadlines, which are transformed into frame deadlines. Deadline Monotonic Scheduling by Audsley et al. [1, 3, 2] has been proven to be a simple and straightforward way of assigning periods to tasks such that they fulfill their deadlines. Its

precursor Rate Monotonic Scheduling by Liu and Layland [59] assumes that the task or frame level periods are already known and equal to the deadlines. For time-triggered systems also the approach of Grenier et al. [38] is based on known signal production rates and signal deadlines. Further Ding et al. [24] assumed statically given task deadlines. Nossal and Galla [68] do not support deadlines at all and rather concentrate on period-based message scheduling.

A practical problem remains unsolved in these approaches. The approaches do not answer the question, where the deadlines and periods come from and how they can be determined. Often task and frame level deadlines depend on more complex system-wide dependencies, such as end-to-end deadlines. Also the origin of task and frame level periods remains unclear. The industry needs an approach to derive all those lower level constraints from the real system-wide timing constraints, which we call function-triggered timing constraints. The actual challenge, which is not tackled by the approaches summarized in this chapter, is the challenge of deriving these lower level constraints, which we call timing requirements for subsystems.

The task scheduling method of Gerber et al. [37] is based on the generation of intermediate task level constraints from end-to-end constraints. It tries to find a task level constraint setup that leads to a fulfillment of the system level constraints. This procedure is somewhat related to our approach. However the task level constraints in this case are just a technique of storing temporary results until a valid solution is found. They are not used as exchange data between development teams like in our approach. Further the method does not consider feedback loops and appropriate responses to unfulfilled task level constraints. Our approach concentrates on such a process using appropriate iterative requirement scheduling algorithms.

4.7.2 Visibility of Timing Properties

Some approaches are not based on statically given task or frame deadlines, but rather they consider system scheduling with system-wide deadlines. An interesting example of such an approach is the work of Schild and Würtz [84], who directly generate an entire and detailed system schedule with all its necessary timing properties. The compositional scheduling analysis by Richter et al. [74, 49] works vice versa. Given a complete system schedule consisting of all timing properties like task and frame periods and offsets, compositional scheduling analysis can determine worst case end-to-end delays of data paths through the system. The results can be compared to given timing constraints for those paths.

These approaches also have an important drawback regarding their practical application. We already described the distributed development environment that the automotive industry is facing in Section 3.1. Distributed development implies some specific requirements for timing models and scheduling approaches, which the investigated approaches and scheduling methods do not fulfill. The system schedule cannot be generated or analyzed at once, because several teams, which are responsible for the timing configuration of only one subsystem, for example one ECU or the communication bus, develop the huge automotive network. Thus a scheduling approach that is applicable

for distributed development must support a timing model with a higher level of abstraction than the pure timing properties, because these are not necessarily visible. According to our assumed collaboration workflows described in Section 3.1 several suppliers develop subsystems that are integrated by the system designer.

Assumption 4.7 *We assume that the suppliers do not reveal all technical details of their subsystems, like the concrete task or frame scheduling, because these are part of their intellectual property.*

All approaches we investigated in this chapter assume to have all timing properties available and visible. None of them provides any means of abstracting from timing property details. In the typical automotive collaboration workflows today the system designer has a central role. The suppliers of ECUs provide him an implementation that fulfills given requirements, amongst those also timing requirements. The suppliers keep their intellectual property as described in Assumption 4.7, which means here the internal timing configuration like task or frame mapping and scheduling. A more abstract level is required as "exchange language" between a central system integrator and the suppliers.

The concept of response times of tasks (Joseph and Pandya [51]) and CAN frames (Tindell et al. [95]) can be seen as abstraction of the underlying timing properties, because a WCRT is a scheduling analysis result that hides those details. Our timing model however is even more abstract, because it is based on the concept of events and event chains, regardless of the actual underlying schedule elements.

4.7.3 Consequences of Changes

The usual pure low-level timing properties that all models of the literature survey are based on do not qualify as target for timing requirements that are exchanged between development teams, because they are too unstable. During development such details are changed rather often. A typical development cycle of a car takes several years. During that time a lot of changes happen to the system under development. The more abstract timing requirements are, the greater is the degree of freedom of the underlying timing properties and the more stable the entire timing configuration is.

The same argumentation also applies for the reuse of subsystems. If a subsystem is integrated in a new environment, it is not necessary to know all exact implementation and timing configuration details. It should be possible to perform timing analysis of the system based on abstract guarantees of the subsystems. The approach of Gerber et al. [37] to generate intermediate task deadlines from end-to-end deadlines is an interesting approach at a glance, because timing analysis is then performed on the level of that deadlines using standard analysis approaches like RMS [59] or DMS [1, 3, 2]. However the approach only focuses on one ECU instead of a system including a communication bus. Further the deadlines are not kept as negotiation basis between

development teams, because the authors do not consider the problem of distributed development.

4.7.4 Conclusion of the Related Work Analysis

To overcome these limitations of scheduling and timing analysis related work with respect to distributed development we created a new timing model and a development methodology. We introduce the timing model and the methodology in Chapter 5. The methodology is based on an iterative subsystem requirements generation process. The algorithms to generate such requirements from function-triggered end-to-end timing constraints is explained in Chapter 6. Table 4.12 summarizes our approach.

Approach	Step	Input	Output	Conditions
TIMEX Model and Methodology	abstract system level scheduling	function- triggered timing constraints	subsystem timing requirements	<ul style="list-style-type: none"> • iterative approach • abstraction from low-level timing properties

Table 4.12. System scheduling approach addressed by our work.

TIMEX - A new Timing Model for Distributed Development

In this chapter we introduce TIMEX and its according development methodology. TIMEX is our new timing model we use for our approach to tackle some of the current development challenges and limitations of recent real-time systems research. Our focus are challenges that are induced by distributed development, which we described in the previous Chapter 4.

5.1 Need for a new Timing Model

As described in Section 2.2.1, automotive systems are so-called distributed systems. This means that functions, which are provided by the car, can be distributed across the system. Several ECUs and communication busses may be involved in the realization of a function. These functions often have to fulfill hard timing constraints, because automotive systems are also real-time systems, as described in Section 2.1.1. Additionally, in Section 3.1 we depicted the distributed development process of the automotive industry. Different roles with different responsibilities even across different companies collaboratively cooperate during development. Subsystems are developed by single teams and integrated to a complete system by the car manufacturer. The integrated system has to fulfill many functional and non-functional requirements. Amongst these also are function-triggered timing constraints, which we explained in Section 3.5.2.

Because of the distributed functions being developed by different teams, function-triggered timing constraints can lead to mutual timing dependencies of the different teams and roles in a distributed development process. In the following we give some examples for such timing dependencies in distributed development.

- In a synchronous FlexRay network the ECU and bus schedules are tightly coupled because they use the same time base.
- In an asynchronous CAN network all sending ECUs may influence the sending behavior of each other dynamically on the shared bus.

- A certain software component’s implementation leads to a concrete execution time when integrated on an ECU.
- The synchronization of events on different ECUs can imply the synchronization of the work of different ECU developers.

In this thesis we focus on the role system integrator. The system integrator has the responsibility to ensure the system’s correct timing behavior. That means he must coordinate the previously mentioned timing dependencies of the various teams in the distributed development process. ECU developers and especially software component developers do not necessarily know the function-triggered timing constraints. Therefore they need a clear specification of the desired subsystem’s timing requirements.

There are many examples for such kinds of collaboration dependencies regarding a system’s timing behavior in a distributed development process. These dependencies of subsystems cause some additional challenges with respect to timing in such a distributed development process. We summarize these challenges later in this section.

Based on our role definitions in Section 3.1.1 we assume the mapping of roles and developed subsystems according to Table 5.1.

Development Role	Developed Subsystem	Development Scope	Timing Properties
SWC developer	software component	software component without execution context	execution time
ECU developer	ECU	ECU without network	task properties, ECU latency, WCRT
system designer	communication bus	complete system with network and ECUs	bus frame properties

Table 5.1. Subsystems developed by the different roles in distributed development of automotive systems.

SWC developers only focus on one SWC, ECU developers only focus on one ECU. These subsystems are delivered to the system integrator and to the ECU developer, respectively. The system integrator develops the communication bus scheduling and additionally integrates the system. The term *development* may technically mean different things depending on the subsystem type. In our context it is only important to point out who is responsible for the resulting timing behavior of which subsystem type.

Note that the system designer plays a special role in our assumed development process. Additionally to developing the communication bus he is the central role that integrated the entire system. From a timing viewpoint, we therefore assume the system designer as the central mediator of timing budgets for the system. The other roles only focus on the subsystem developed by them. This is summarized in Assumption 5.1.

Assumption 5.1 *The scope of SWC developer and ECU developer solely is the subsystem, which is developed by the role (i. e. a single SWC or ECU respectively). These two roles do neither know any development details (e. g. timing behavior) of another party's subsystem nor of the overall function that their subsystem participates in. The system integrator however is the central role that coordinates the entire system's timing.*

Based on Assumption 5.1 and the development process described in this section, which in fact conforms to today's practice in automotive development, some challenges with respect to timing arise.

- The timing behavior of subsystems developed by different teams can depend on each other or influence each other. Some examples for such dependencies are already given in the beginning of this section.
- When the system timing is evaluated, every subsystem's timing behavior must be considered. If the system timing is not correct, i. e. if function-triggered timing constraints are not fulfilled, it is often difficult to point to a specific subsystem that causes the incorrect behavior.
- A subsystem's timing behavior can only be evaluated with respect to a certain context definition, sometimes also called guarantees of the rest system. Due to the fact that we assume subsystems to be developed independently from each other, the context must be specified explicitly, because the rest of the system is not necessarily visible to the subsystem developer.
- Once a given system implementation has successfully been analyzed and the fulfillment of all its function-triggered timing constraints has been ensured, the analysis result can become invalidated if changes of one or more subsystems occur. The changed subsystems furthermore must not even be part of the functions whose behavior is influenced by the change. The effect of such changes with respect to function-triggered timing constraints is not always clearly visible and predictable.
- Often subsystems are reused in other systems with a different context. The reuse is clearly more difficult and costly if the subsystems timing is not known or not formally specified.
- Often subsystems delivered by suppliers are subject to regulations regarding the protection of intellectual property (IP) of the supplier. Such IP of subsystems often is not disclosed to the carmaker, who holds the role of the system integrator in our collaboration scenarios. The system integrator however needs certain information to be able to analyze the system's timing behavior.
- The definition of constraints and later analysis of the timing behavior of a system can be a very challenging task, because typical automotive systems today are very complex. For timing analysis, an abstraction of the system complexity is necessary.
- Following the AUTOSAR approach every delivered subsystem must be accompanied by an according formal specification. This approach should be extended to the subsystem's timing behavior. An additional, abstract, for-

mal, and standardized way to develop and maintain a timing specification is therefore mandatory.

- A collection of such timing description specifications should enable model-based timing analysis of the whole system. Incompatible description formats or semantics can impede successful and correct system timing analysis.
- It is not possible to analyze function-triggered timing constraints of distributed functions on subsystem level. A mediator needs to collect subsystem timing specifications and decide about the fulfillment of such constraints.

To tackle these challenges, we define a special timing model, because none of the existing approaches and models described in Chapter 4 is suitable for all these demands. Additionally we define a methodology for distributed development of automotive real-time systems based on function-triggered timing constraints that makes use of our new timing model. We describe both the timing model and the methodology in this chapter.

In the subsequent Chapter 6 we present our constraint logic programming approach to generate timing requirements for the different subsystems according to the function-triggered timing constraints.

5.2 TIMEX - Timing Extension

5.2.1 Timing Extension Definition

As described in Section 3.3 a system model in our case is used to capture the static structure of an automotive software system. The description or modeling of the system behavior is not in the model's scope. Furthermore the system model includes some implementation details regarding its timing for tasks and frames in the technical architecture, which we call timing properties.

Given such a system model, model-based timing analysis can be performed to determine the system's expected timing behavior. However, to decide whether the expected timing behavior of the system is acceptable, it must be compared with the timing constraints that the system has to fulfill. The constraints are not part of the actual system model itself. Rather they are modeled, or specified, independently of the system model. Therefore we use a concept that we call *timing extension*.

Definition 5.1. *A timing extension is a model or specification dedicated to a real-time system's timing constraints that is provided additively to the actual system model.*

To capture timing constraints, a timing extension must have the following three characteristics.

1. A timing extension must provide an appropriate technical means for timing constraint modeling.
2. A timing extension must precisely define its model semantics.
3. To enable constraint verification after model-based system timing analysis, also the relation – or mapping – of the independent timing extension to the according system model elements must be clear.

We briefly explain how our timing extension model called TIMEX fulfills each of these three characteristic requirements in the subsequent section. TIMEX and its methodology are described in detail in the remainder of this chapter.

5.2.2 TIMEX

The timing extension defined in this thesis is called TIMEX. A TIMEX model is used to attach a timing extension to a given system model. It conforms to the characteristic requirements mentioned before as follows.

TIMEX is used to describe the static timing relations of elements within a given system. It is founded on the basic principles of observable events and event chains, which are described in Section 3.2. Summarized, an event chain is a sequential temporal order of observable events along a data path through the system. TIMEX offers model elements to describe such paths. Of course the system model itself often directly exposes those paths. For example the logical and technical architectures of our system model reveal the data paths across the complete system by channels, which connect components, and by tasks and frames, which describe on which ECU components are executed and which frames are used to exchange messages. The purpose of TIMEX however is a) to abstract from the underlying system model, and b) to describe also such paths that are not completely visible in the system model. This could be for different reasons. Possible reasons are for example IP protection, a (yet) incomplete model, or distributed development where only a subsystem model is available. Three different types of timing constraints can be defined for the data path definitions using TIMEX. Observable events (later called hops), event chains (later called function event chains and segments), and timing constraints thus are the technical means for timing modeling with TIMEX. These concepts are formally described in Section 5.3.

As described in the overview of current real-time systems and timing analysis literature in Section 3.2 and Section 3.5.1, different meanings and semantics of events and timing constraints exist. The semantics of TIMEX thus must be precisely defined to avoid misunderstandings. This is important for all parts of TIMEX. The semantics of observable events was already formally defined in Section 3.4. Function chains and segments, i. e. event chains built up with observable events, are also formally defined later in this section. A clear semantics of such event chains is crucial for the formal definition of timing constraints provided by TIMEX. The semantics of the provided timing constraints is defined in Section 5.3.3.

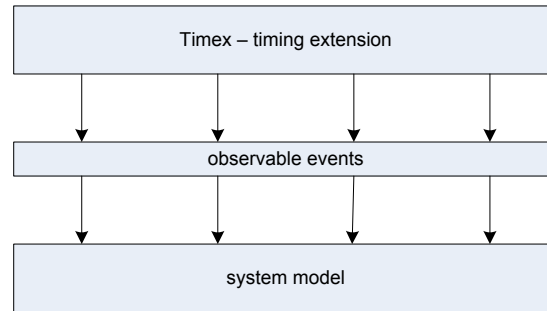


Fig. 5.1. Observable events as our concept to connect a system model with a timing extension like TIMEX.

TIMEX uses the concept of observable events and event chains to capture timing constraints additionally to a given system model. Thereby TIMEX is an abstraction of various system model details, e. g. details of the technical and logical architecture. For a successful round trip of system modeling, timing constraint specification and model-based timing analysis, the connection of TIMEX and the underlying system model must be clear. The concept of observable events represents this connection, or "the glue", between system model and timing extension, as depicted in Figure 5.1.

The special focus of TIMEX is distributed development. As described in Section 3.4 observable events are available on all three architecture layers of our system model. Using these observable events and the provided event chains in TIMEX, timing modeling on different abstraction layers is enabled.

- Observable events of the functional architecture are used to model function event chains and function-triggered timing constraints, which we defined in Section 3.5.
- Observable events of the logical and technical architecture are used to refine function event chains to segments and thus build up event chains from a sensor event to an actuator event of a function.
- Segments indicate the parts of a data path, which are assigned to a specific subsystem in a distributed development process. This enables the mapping of function-triggered timing constraints to requirements for subsystems, see Section 3.6.

The abstraction of the underlying system model together with the formal definition of the mapping between TIMEX and the system model are a great benefit, because reasoning about the system's timing can be performed on the more abstract and more controllable level of TIMEX. Therefore all TIMEX model elements are described in detail in the following.

5.3 TIMEX Model Elements Formalization

TIMEX is a special timing model for a distributed development process, which is addressed by this thesis. As described in the previous section TIMEX is a timing extension that extends a system model by additional timing information. Summarizing, TIMEX is able to capture the following two very essential kinds of information that are important for our approach to overcome the challenges described in Section 5.1 of the distributed development process outlined in Section 3.1.

1. TIMEX enables the modeling of function-triggered timing constraints. According to Definition 3.5 such constraints are independent of a concrete implementation. According to our approach, the system model must not be completely known to define timing constraints for functions. In a pure top-down development process this is a mandatory feature, because function-triggered timing constraints are known prior to their implementation, which is expressed by the system model. The TIMEX model particularly supports this, as we will explain later.
2. Once a system model – i. e. the implementation details of the functions – is complete, the already modeled function-triggered timing constraints can be refined to requirements for the subsystems, as we described in Section 3.6. In the TIMEX model this is realized by the *segmentation* (see Definition 5.5) of the system function event chains to subsystem segments, which will be detailed later. This refined timing model enables a clear responsibility assignment (see Section 3.1.3).

The first part is covered by what we call the *function timing* of TIMEX. The second part is covered by what we call the *system timing*. Function timing and system timing together are the TIMEX timing extension for a given system model.

In the next section we present an example that is used to explain TIMEX and its model elements. The single elements of TIMEX that we use in the example as well as their relations, semantics and formal definition are described in detail afterwards.

5.3.1 TIMEX Example

In Section 3.3.6 we already formally specified an example of a system model that we also visualized in Figure 3.12. The system model offers several observable events, which we defined in Section 3.4.4. All these events belong to one data path. They can therefore be ordered temporarily, as depicted in Figure 3.13.

Based on this example and its observable events we now define a TIMEX model, which makes use of the given observable events. This way we define a timing extension for the system model example. The timing extension relates certain observable events to each other to create event chains and to specify timing constraints for the system's function *damperControl*.

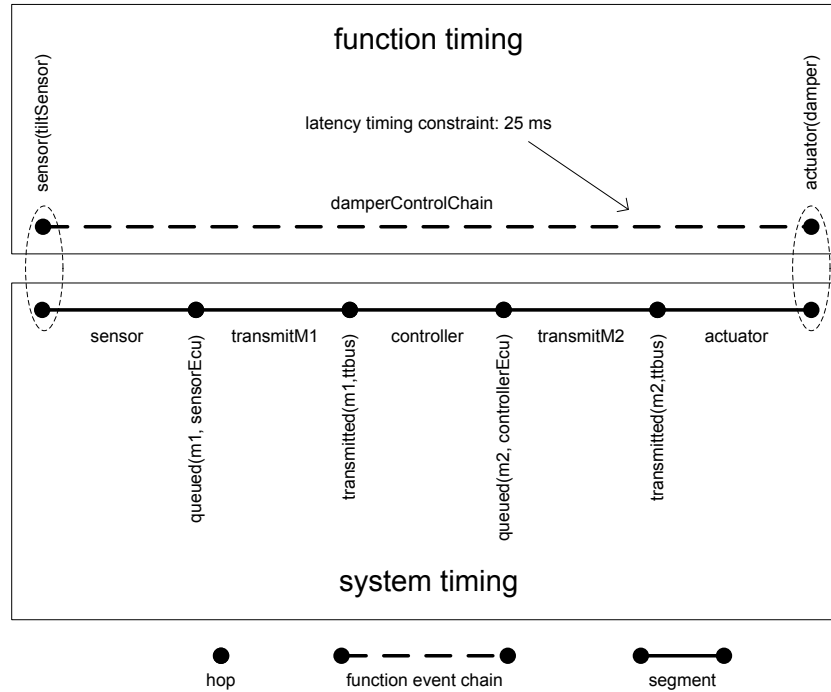


Fig. 5.2. Timing extension of the example system model using TIMEX on both function level and implementation level.

Figure 5.2 depicts the TIMEX model example. The upper part shows the before mentioned function timing. It contains one so-called *function event chain*, which is an event chain according to Definition 3.3. It also contains a latency timing constraint for the function event chain. The lower part of the figure shows the system timing of the TIMEX model. It is used to refine the function event chain by so-called *segments*. Segments are also event chains according to Definition 3.3.

Some of the observable events offered by the example are used in this timing extension. These observable events thus are called hops. By the definition of hops, observable events are made available to the TIMEX model and can be used for function event chain and segment definitions. The system timing reuses the two hops of the function timing here (indicated by the dotted ellipse).

All these TIMEX elements are explained in the two subsequent Sections 5.3.3 and 5.3.4.

5.3.2 Hand Over Points

As one can see in Section 3.4, a complex system model can reveal many observable events on all three layers. For our methodology (see Section 5.5), some of the many observable events are of special interest, namely the ones that are needed for function event chains and segments in the timing exten-

sion and, after all, to express timing constraints. Not necessarily all available events must belong to this group. It depends on the decomposition of the system to subsystems and the collaboration workflows that are realized in the development process (see Section 3.1.2).

To achieve an identification of important observable events, every observable event can be characterized as a *hand over point* in TIMEX, or short as a *hop*.

Definition 5.2. *A hand over point, or hop, is an observable event that is provided by the system model and that is made available to the timing extension. A hop is an observable event that can be observed at the interface of a system or at the border of two subsystems.*

Given an observable event e , the corresponding hop h in TIMEX that references the event is denoted by

$$\text{hop } h = \langle e \rangle$$

The referenced event can be accessed with the operator *.event*. The set H is the set of all hops h_i of the TIMEX model.

$$H = \{h_i\}$$

Every hop $h_i \in H$ is of a certain type that depends on the referenced observable event. According to the six different types of observable events on all three layers of the system model, which are described in Section 3.4, we define the type of a hop as follows. Let \mathbb{T} be the data type of all available hop types. \mathbb{T} contains the following elements.

$$\mathbb{T} = \{\text{sensor}, \text{actuator}, \text{send}, \text{receive}, \text{queued}, \text{transmitted}, \perp\}$$

The function *hoptype* determines the type of hop $h = \langle e \rangle$ as follows, based on the different sets of observable events of Section 3.4 on the three layers of the system model FA , LA and TA .

$$\text{hoptype} : H \rightarrow \mathbb{T}$$

$$\text{hoptype}(h) = \begin{cases} \text{sensor} & \text{if } h.\text{event} \in \text{Sensor}(FA) \\ \text{actuator} & \text{if } h.\text{event} \in \text{Actuator}(FA) \\ \text{send} & \text{if } h.\text{event} \in \text{Send}(LA) \\ \text{receive} & \text{if } h.\text{event} \in \text{Receive}(LA) \\ \text{queued} & \text{if } h.\text{event} \in \text{Queued}(TA) \\ \text{transmitted} & \text{if } h.\text{event} \in \text{Transmitted}(TA) \\ \perp & \text{else} \end{cases}$$

5.3.3 Function Timing

Hand Over Points in Function Timing

For the function timing part possible hops are all observable events of function interfaces, i. e. sensor and actuator events of the set FA , because they are used to model function event chains later. For all interfaces of available functions of a functional architecture a hop can be defined.

The set $H_{function} \subseteq H$ is the set of all hops of the function timing. $H_{Interface}$ is defined as follows.

$$H_{function} = \{\langle e \rangle \mid e \in Interface(FA)\}$$

The set $H_{sensor} \subseteq H_{function}$ contains all hops that reference sensor events. For these hops the following condition holds.

$$\forall h \in H_{sensor} : hoptype(h) = sensor$$

The set $H_{actuator} \subseteq H_{function}$ contains all hops that reference actuator events. Again, for all actuator hops the following condition holds.

$$\forall h \in H_{actuator} : hoptype(h) = actuator$$

Sensor and actuator hops together represent all available hops of the function timing in TIMEX .

$$H_{function} = H_{sensor} \cup H_{actuator}$$

Function Chains

Function timing is used to capture all function-triggered timing constraints as explained in Section 3.5. Therefore, the function timing part of TIMEX contains a set of so-called *function event chains*, which are the basis for timing constraint specification.

A function event chain, which we also briefly call *chain*, is an event chain as defined in Definition 3.3 with a special semantics. It references a stimulus hop and a response hop and thus models an end-to-end timing dependency of a function. According to Definition 3.3, the semantics of a function event chain is that the event of its response hop can be observed as a causal consequence after every observation of the event referenced by its stimulus hop.

A function event chain f with stimulus hop s and response hop r is denoted by the tuple

chain $f = \langle s, r \rangle$, with $s \in H_{sensor}$ and $r \in H_{actuator}$

The stimulus and response hops can be accessed with the operators *.stimulus* and *.response*. The set F contains all function event chains f_i of the function timing.

$$F = \{f_i\}$$

When function event chains are specified it is not necessary to already know exactly what happens in between these two hops of a function event chain. However, according to the semantics definition of an event chain there must be a data path through the system from the stimulus to the response in the final implementation. Otherwise timing analysis is not possible, because no relation between certain stimulus and response occurrences can be determined. Broy et al. [11] call such dependencies between actions that belong to different components *interactions*. Because our observable event definitions are also based on the action concept, a function event chain is similar to such an interaction.

Function-triggered Timing Constraints

Based on all function event chains, which were introduced in the previous section, function timing also contains the set of function-triggered timing constraints. These can be used to constrain the timing behavior of the function event chains. Function-triggered timing constraints are independent of a concrete implementation, i. e. independent of a concrete system with already known logical and technical architecture. Therefore, function timing only uses those observable events, which are provided by the functional architecture, namely sensor and actuator events. Function event chains, whose timing behavior is constrained by function-triggered timing constraints, reference these events. The specification thus is what we call *implementation-independent*.

Note that for our methodology (Section 5.5) as well as for our algorithms to determine subsystem timing requirements based on function-triggered timing constraints (Chapter 6) it would make no difference, if the set of hops in function timing would be extended by hops for events of the logical or technical architecture. However, this restriction is necessary to ensure the before mentioned independence of implementation details, which is a basic assumption for function-triggered timing constraints (Section 3.5.2).

In Section 3.5.3 we already defined the three main types of function-triggered timing constraints. These types can also be specified within the function timing of TIMEX.

A function-triggered latency timing constraint, or short *latency constraint*, references one function event chain and constrains the minimum and maximum latency between each of this chain's stimulus and response occurrences. That means that after every occurrence of the stimulus an occurrence of the

response must follow within the specified time. A latency constraint l that constrains the latency of function event chain f to a minimum min and a maximum max is denoted by the tuple

$$\textit{latency } l = \langle min, max, f \rangle, \text{ with } min, max \in \mathbb{N}$$

The minimum and maximum attributes can be accessed using the operators *.minimum* and *.maximum*. The referenced target chain is accessed with operator *.target*.

The set LC contains all latency constraints l_i of the function timing.

$$LC = \{l_i\}$$

A function-triggered synchronization timing constraint, or short *synchronization constraint*, references two or more function event chains and constrains the synchronicity of either all stimulus or all response hop occurrences of these function event chains. A synchronization constraint s that constrains the synchronicity of several function event chains $f_i \in T$ – the target set – within a tolerance t is denoted by the tuple

$$\textit{synch } s = \langle T, t, scope \rangle, \text{ with } t \in \mathbb{N} \text{ and } scope \in \{stimulus, response\}$$

The *scope* attribute indicates whether the stimulus or the response of the set of function event chains shall be synchronized within tolerance interval t . All attributes can be accessed with the operators *.target*, *.tolerance* and *.scope* for each synchronization constraint.

The set SC contains all synchronization constraints s_i of the function timing.

$$SC = \{s_i\}$$

A function-triggered triggering timing constraint, or short *triggering constraint*, references one function event chain and constrains the occurrence behavior of the stimulus of that function event chain, also called its *triggering*.

For simplicity reasons we only assume strict periodic triggering constraints in our TIMEX model. We assume that for example jitter and other special event models that we discussed in Section 3.2.1 are not specified as triggering timing constraints for a function. Such triggerings however may of course occur as a result of a certain system implementation. We do not provide timing analysis semantics that define in which cases a strict periodic triggering constraint is fulfilled by which non-strict periodic timing behavior. The scope of our work is not to define a new analysis method but a mapping between subsystem timing requirements and function-triggered timing constraints.

A triggering constraint t that references function event chain f and constrains a triggering period p is denoted by the following tuple.

triggering $t = \langle p, f \rangle$, with $p \in \mathbb{N}$

The period can be accessed with the operator *.period*, the operator *.target* can be used to access the target function event chain.

The set TC contains all triggering constraints t_i of the function timing.

$$TC = \{t_i\}$$

Finally, we can define the set C , which contains all function-triggered timing constraints of a TIMEX timing extension for a system model.

$$C = LC \cup SC \cup TC$$

5.3.4 System Timing

Based on a function timing model of TIMEX a certain *system timing* can be defined. If a functional architecture is further used in a system model and refined by a logical and technical architecture model, as described in 3.3, a lot of additional implementation details appear that are not visible in a pure functional architecture. Examples of such details are the component structure of functions, component mapping to ECUs or of course the mapping of messages, or signals, to frames. Another important implementation detail is the type of the communication bus, i. e. time-triggered or event-triggered communication. This is a basic information when it comes to the mapping procedure of function-triggered timing constraints to subsystem timing requirements, which is explained in Section 6.3.

Definition 5.3. *We call all facts that lead to additional implementation details in a system model design decisions. A TIMEX system timing reflects all timing relevant design decisions.*

Design decisions influence the refinement of function event chains in a system timing model using additional hops and segments, which are described in this section. The design decisions influence what types of segments are needed for function event chain segmentation (see Definition 5.5). In section Section 6.2.1 we summarize important design decisions and their influence on timing requirements. Because function timing, as we defined it, is not influenced by such kinds of design decisions, a system timing model is a means to capture the effects of all design decisions in TIMEX.

Hand Over Points in System Timing

According to our methodology and our understanding of distributed development (see Section 3.1) the system designer is able to determine which subsystems of the overall system are developed by which team. According to

Definition 5.2 hops are used to represent observable events in TIMEX and characterize them with the special semantics, which is an observation at the border of two subsystems. Thus the system designer can identify system timing hops additionally to the ones already known from the function timing. These are observable events on the border of two responsibilities of different teams, or roles. They mark the places in the system model where data is handed over from one subsystem to another, for example from an ECU to the communication bus.

All these additional hops that stem from observable events of the logical and technical architecture are collected in the system timing, because they are no more implementation-independent as the ones of the function timing. All these hops must reference send and receive events of the set $Port_{LA}$, or queued and transmitted events of the set $Signal_{TA}$.

The set $H_{system} \subseteq H$ is the set of all hops of the system timing. For all these hops the following condition holds.

$$H_{system} : \{h \mid h.event \in Port(LA) \vee h.event \in Signal(TA)\}$$

The set $H_{send} \subseteq H_{system}$ contains all hops that reference send events. For these hops the following condition holds.

$$\forall h \in H_{send} : hoptype(h) = send$$

The set $H_{receive} \subseteq H_{system}$ contains all hops that reference receive events. For all these hops the following condition holds.

$$\forall h \in H_{receive} : hoptype(h) = receive$$

The set $H_{queued} \subseteq H_{system}$ contains all hops that reference queued events. The following condition holds for them.

$$\forall h \in H_{queued} : hoptype(h) = queued$$

The set $H_{transmitted} \subseteq H_{system}$ contains all hops that reference transmitted events. For all these hops the following condition holds.

$$\forall h \in H_{transmitted} : hoptype(h) = transmitted$$

These four sets of hops together represent all available hops of the system timing.

$$H_{system} = H_{send} \cup H_{receive} \cup H_{queued} \cup H_{transmitted}$$

Segments

All hops of both function timing and system timing, which together we denoted by set H , can be used to model a set of so-called *segments*. Similar to function event chains, a segment is also a special event chain according to Definition 3.3 with a stimulus and a response hop. In a consistent TIMEX model every function event chain is refined by an arbitrary number of segments without gaps and also without overlaps. One segment however can be used for the refinement of many function event chains. Furthermore, one segment can be part of several function event chains. We precisely describe and formalize TIMEX model consistency in Section 5.4.

The semantics of a segment is that it is an event chain according to Definition 3.3, which represents a sub-path of a function event chain's data path. The sub-path spans the entire subsystem, from on border as data input to the other border as data output. Note that there must be a causal dependency between input and output. Otherwise whatever timing constraints are specified in TIMEX, a later timing analysis cannot relate the analyzed timing behavior to the constraints. In other words, a segment belongs to the responsibility of the one team that develops the according subsystem. Similar to function event chains, segments again could also be called interactions, according to Broy et al. [11], because they represent causal dependencies between actions that belong to different components.

A segment s with stimulus hop s and response hop r is denoted by the tuple

$$\text{segment } s = \langle s, r, res \rangle, \text{ with } s, r \in H$$

The hops can be accessed with the operators *.stimulus* and *.response*. The attribute *res* references the subsystem that the segment belongs to. It can be accessed with the operator *.res*. The set S contains all segments s_i of the system timing.

$$S = \{s_i\}$$

System timing consists of all additional system timing hops H_{system} and all segments S that use system timing hops as well as function timing hops.

We will assign some timing attributes to segments and hops later, namely the already mentioned requirements and guarantees. However, these timing attributes within the TIMEX model are not meant to be modeled by an engineer, but generated during the process of mapping function-triggered timing constraints to requirements. Manual timing modeling is done only on the level of function timing by specifying timing constraints.

A detailed description of our methodology is presented in Section 5.5. Our approach to automated segment and hop requirement generation is described in Chapter 6.

5.3.5 TIMEX Example Formalization

Based on the formalism introduced in Section 5.3.3 and Section 5.3.4, we now provide a formalization of the TIMEX model example of Section 5.3.1. The example uses some of the observable events provided by the example system model, which we already introduced in Section 3.4.4 and thereby specified a function-triggered timing constraint for the example system model.

Example Function Timing

There is one function event chain in the example. It needs two hops, which are the chain's stimulus and response hops.

$$\begin{aligned} \text{hop } chainStimulus &= \langle sensor(tiltSensor) \rangle \\ \text{hop } chainResponse &= \langle actuator(damper) \rangle \end{aligned}$$

The sets of hops of the function timing thus simply are as follows.

$$\begin{aligned} H_{sensor} &= \{chainStimulus\} \\ H_{actuator} &= \{chainResponse\} \end{aligned}$$

Based on these hops the function event chain of the example is formalized by

$$\text{chain } damperControlChain = \langle chainStimulus, chainResponse \rangle$$

The set of function event chains only contains one element.

$$F = \{damperControlChain\}$$

According to the example there is one latency constraint for the chain *damperControlChain* with a minimum of 1 and a maximum of 25 milliseconds.

$$\text{latency } damperLatency = \langle 1, 25, damperControlChain \rangle$$

The set of latency constraints contains latencyDamperControl.

$$LC = \{damperLatency\}$$

Example System Timing

Based on the function timing model, we now formalize the system timing model of the example.

As one can see in Figure 5.2 there are four additional hops. These are formalized as follows.

$$\begin{aligned}
 \text{hop } \text{queuedM1} &= \langle \text{queued}(m1, \text{sensorEcu}) \rangle \\
 \text{hop } \text{transmittedM1} &= \langle \text{transmitted}(m1, \text{tbus}) \rangle \\
 \text{hop } \text{queuedM2} &= \langle \text{queued}(m2, \text{controllerEcu}) \rangle \\
 \text{hop } \text{transmittedM2} &= \langle \text{transmitted}(m2, \text{tbus}) \rangle
 \end{aligned}$$

The following hop sets of the example system timing exist.

$$\begin{aligned}
 H_{\text{send}} &= \emptyset \\
 H_{\text{receive}} &= \emptyset \\
 H_{\text{queued}} &= \{\text{queuedM1}, \text{queuedM2}\} \\
 H_{\text{transmitted}} &= \{\text{transmittedM1}, \text{transmittedM2}\}
 \end{aligned}$$

Note that H_{Send} and H_{Receive} are empty sets, because this TIMEX model does not use any observable events of the logical architecture.

Based on the set of all hops H of function timing and system timing we now formalize the set of segments as follows.

$$\begin{aligned}
 \text{segment } \text{sensor} &= \langle \text{chainStimulus}, \text{queuedM1}, \text{ecu1} \rangle \\
 \text{segment } \text{transmitM1} &= \langle \text{queuedM1}, \text{transmittedM1}, \text{bus} \rangle \\
 \text{segment } \text{controller} &= \langle \text{transmittedM1}, \text{queuedM2}, \text{ecu2} \rangle \\
 \text{segment } \text{transmitM2} &= \langle \text{queuedM2}, \text{transmittedM2}, \text{bus} \rangle \\
 \text{segment } \text{actuator} &= \langle \text{transmittedM2}, \text{chainResponse}, \text{ecu3} \rangle
 \end{aligned}$$

In the example, we assume that the system consists of three ECUs (sensor ECU, controller ECU, actuator ECU) that are connected to one bus. This is modeled by the *res* attributes of each segment.

The set S contains all segments.

$$S = \{\text{sensor}, \text{transmitM1}, \text{controller}, \text{transmitM2}, \text{actuator}\}$$

5.4 Consistency of a TIMEX Model

The TIMEX model example of the previous section is structurally correct. While we introduced TIMEX and its formalism we already mentioned a few conditions that every TIMEX model must meet. In fact, there are several such rules that can be applied to check such a model's structural correctness.

Definition 5.4. *The structural correctness of a TIMEX model is called consistency.*

The consistency of each TIMEX model is very important for its later use in the automated generation of timing requirements for hops and segments. The rather generic TIMEX model allows for the specification of inconsistent models as well. A set of consistency rules shall be used to obviate model inconsistency. In this section we highlight and formalize all such consistency rules.

A complete TIMEX model as timing extension of a system model consists of function-triggered timing constraints C and function event chains F modeled as function timing, and the implementation-specific segments S modeled as system timing. Additionally, it of course contains the sets of hops for both function timing $H_{function}$ and system timing H_{system} . We define all consistency rules based on these element sets of a TIMEX model.

As one can see in the example TIMEX model in Figure 5.2, a function event chain is refined by a set of segments.

Definition 5.5. *The complete representation of a function event chain by means of segments is called the function event chain's segmentation.*

Many of the consistency rules we present in the following refer to the segmentation of function event chains. Other rules refer to the type of hops that are used by the model or to the usage of constraints. The TIMEX example of Section 5.3.1 meets all these consistency rules.

5.4.1 Reasonable Hops

As already indicated in Section 3.4 the set of possible observable events of a system model can be very large. Virtually every action of the system, whose occurrences can clearly be temporarily exposed by an analysis method, qualifies as an observable event and thus as a potential hop for TIMEX. However for the intended purpose of TIMEX only certain observable events make sense. This purpose is distributed development and, more precisely, a support for the collaboration workflows of the three main roles described in Section 3.1.2.

For this reason we already restricted the set of observable events to the six types mentioned in Section 3.4, although there were more events possible (e. g. the observation of starting or stopping runnables). There are two types for each level of the system model. Two of which regard functions, two others

components, and the last two regard ECU communication. These are exactly the events needed to reflect the two main collaboration workflows, namely ECU integration and SWC integration.

In Section 5.3.2 we introduced a function *type* that assigns a certain type to a hop, depending on the event the hop references. For the purpose of TIMEX only the six above-mentioned events make sense in the timing extension. For this reason, for every hop in TIMEX one of the six types must assignable using function *type*. This rule is formalized by consistency Rule (5.1)

$$\forall h \in H : \text{hoptype}(h) \neq \perp. \quad (5.1)$$

These are the *reasonable hops* for a consistent TIMEX model. This is a rule for the validity of possible types of hops.

5.4.2 Loose Hops

Another rule to apply to the set of hops regards so-called *loose hops*.

Definition 5.6. *A loose hop is a hop, which is not used as stimulus or as response in any segment.*

Definition 5.6 only caters to the usage of every hop in at least one segment, not in chains. However, every hop, which is used in a function event chain is also used in at least one segment, given that the TIMEX model is consistent, especially according to Rule (5.6). Thus Definition 5.6 is sufficient.

For a consistent TIMEX model we require that no hop is a loose hop, because a hop only shall be declared, if the referenced observable event is needed for the definition of timing constraints and thus needed for segments and chains. A loose hop does not serve any purpose in TIMEX.

The loose hop rule is formalized by Rule (5.2).

$$\forall h \in H : \exists s \in S : s.\text{stimulus} = h \vee s.\text{response} = h. \quad (5.2)$$

5.4.3 Outer Hops

The set of hops H can be divided into the two disjoint sets of so-called *inner hops* and *outer hops*, according to Definition 5.7 and Definition 5.8.

Definition 5.7. *An outer hop is a hop, which is only used as stimulus or as response of any segment.*

Definition 5.8. *An inner hop is a hop, which is used as stimulus of a segment and as response of another segment.*

As another consistency rule, we require that every outer hop is used in a function event chain, either as a stimulus or as a response hop. The reason for this rule is that an outer hop marks the end of a row of segment, i. e. the end of a chain segmentation. Therefore there must be a chain that is segmented by this row of segments.

The set $H_{out} \subseteq H$ contains all outer hops of a TIMEX model. It is defined as follows.

$$H_{out} = \{ h \in H \mid \begin{aligned} &(\exists s \in S : s.stimulus = h \wedge \neg \exists s \in S : s.response = h) \\ &\vee \\ &(\exists s \in S : s.response = h \wedge \neg \exists s \in S : s.stimulus = h) \} \end{aligned}$$

Based on this definition, the formalization of the rule for outer hops is given by Rule (5.3).

$$\forall h \in H_{out} : \exists f \in F : \begin{aligned} &f.stimulus = h \vee \\ &f.response = h. \end{aligned} \quad (5.3)$$

The usage of inner hops is not further restricted by rules.

5.4.4 Segment Types

We already motivated the six available hop types of TIMEX in Section 5.4.1. These are the six observable events of the system model that are relevant with respect to our considered collaboration workflows in Section 3.1.2. Based on these six hop types all segments of the TIMEX model are constructed. However, not every hop of a certain type qualifies for the role of a stimulus hop given a hop of another certain hop types in a response role. The possible combinations of stimulus and response hop types used in segments is restricted, as we will point out in this section. This restriction thus can be formulated as another appropriate TIMEX consistency rule.

The semantics of a segment is that a response hop occurrence can be observed as a causal consequence every time after a stimulus hop occurrence has been observed. Therefore, of course, there must in fact be a causal dependency between the two hops. This can be ruled out for a couple of hop type combinations. We will now identify possible hop type combinations for the different possible subsystem types.

First, consider solely the communication bus as a subsystem. The bus always belongs to the responsibility of the system designer independently of the two

collaboration workflows and represents an own subsystem. An own segment type represents paths through this subsystem. For communication over a bus only one combination of stimulus and response hop types makes sense. A queued event can be observed, when data is handed over from an ECU to the bus, i. e. when the data "enters" the bus subsystem. A transmitted event can be observed, when data is handed over from the bus to an ECU, i. e. when data "leaves" the bus subsystem. So, a segment that covers data transmission over a bus can only have these two types as stimulus and response respectively. We call such a segment *transmission segment*.

Let \mathbb{T}^* be the data type of all segment types. Given function

$$segtype : S \rightarrow \mathbb{T}^*$$

The transmission type for a segment $s = \langle stimulus, response \rangle$ is defined by the function *segtype* as follows.

$$segtype(s) = transmission \text{ if } \begin{aligned} &hoptype(stimulus) = queued \wedge \\ &hoptype(response) = transmitted. \end{aligned}$$

Second, consider the collaboration workflow "ECU Integration". In this case an ECU is an integrated subsystem. Reasonable observable events for this use case are the ones, which can be observed at the border of an ECU. These are sensor and actuator events, where the border of the system, i. e. its interface, also is the border of an ECU. Furthermore, queued and transmitted events belong to the border between an ECU and the bus. So there are four hop types to consider for segments over an ECU as subsystem. With these four hop types there exist three reasonable segment types for ECUs as subsystems.

$$\begin{aligned} segtype(s) = sensortobus \text{ if } & \begin{aligned} &hoptype(stimulus) = sensor \wedge \\ &hoptype(response) = queued. \end{aligned} \\ segtype(s) = overecu \text{ if } & \begin{aligned} &hoptype(stimulus) = transmitted \wedge \\ &hoptype(response) = queued. \end{aligned} \\ segtype(s) = bustoactuator \text{ if } & \begin{aligned} &hoptype(stimulus) = transmitted \wedge \\ &hoptype(response) = actuator. \end{aligned} \end{aligned}$$

A *sensortobus segment* is used to cover the first possible data path over an ECU as subsystem. At a sensor interface data is handed over from the environment to the ECU, i. e. the data "enters" the ECU subsystem. When the ECU software has computed the sensor data internally, it is handed over to the bus and "leaves" the ECU subsystem. The corresponding observable event is a queued event. An *overecu segment* is used for the case, when an ECU receives data from a bus, calculates a response to that data and send the response again over the bus. The data is handed over to the ECU when it is transmitted. The response is handed over to the bus when it is queued for transmission. The

third reasonable segment type for an ECU as subsystem is a *bustoactuator segment*. Such a segment covers a data path from the reception of data on the bus to the action of an actuator that is controlled based on the received data. Again, the data is handed over to the ECU at a transmission event and handed over from the ECU to the system's environment in form of mechanical work when the actuator action is observed.

Third, consider the collaboration workflow "SWC Integration", where a software component is integrated in an ECU. Basically in this case some of the already introduced segments can be refined with more precise segments, because the system timing is able to reveal more details of the corresponding data path than before. A *transmission segment* of course cannot be further refined. The other three segment types introduced so far include the execution of software components between their stimulus and response hops. They can therefore be refined with special segment types, if that SWC are an own subsystem and belong to the responsibility of another team, which then is in the role of a SWC developer. Five more reasonable segment types appear, when send and receive events come into play.

$$\begin{aligned}
 \text{segtype}(s) = \text{sensortoswc} & \text{ if } \text{hoptype}(\text{stimulus}) = \text{sensor} \wedge \\
 & \text{hoptype}(\text{response}) = \text{receive}. \\
 \text{segtype}(s) = \text{overswc} & \text{ if } \text{hoptype}(\text{stimulus}) = \text{receive} \wedge \\
 & \text{hoptype}(\text{response}) = \text{send}. \\
 \text{segtype}(s) = \text{swctobus} & \text{ if } \text{hoptype}(\text{stimulus}) = \text{send} \wedge \\
 & \text{hoptype}(\text{response}) = \text{queued}. \\
 \text{segtype}(s) = \text{bustoswc} & \text{ if } \text{hoptype}(\text{stimulus}) = \text{transmitted} \wedge \\
 & \text{hoptype}(\text{response}) = \text{received}. \\
 \text{segtype}(s) = \text{swctoactuator} & \text{ if } \text{hoptype}(\text{stimulus}) = \text{send} \wedge \\
 & \text{hoptype}(\text{response}) = \text{actuator}.
 \end{aligned}$$

A *sensortobus segment* can be refined into three segments, when the SWC on the data path shall be reflected in TIMEX. This can be done by introducing a hop, where data is handed over to the SWC, which is a receive event, and another hop when data is handed over from the SWC, which is a send event. This procedure results in the three segment types *sensortoswc*, *overswc* and *swctobus*, as formalized above. The refinement of a *bustoactuator segment* by making the involved actuator SWC visible in TIMEX also results in three segments. Again the segments use the receive and send events of the SWC port. The result are the segment types *bustoswc*, *overswc* and *swctoactuator*, as formalized above.

In all other cases function *segtype* returns \perp , which indicates an undefined segment type.

$$\text{segtype}(s) = \perp \text{ else}$$

Summarizing, \mathbb{T}^* contains the following elements.

$$\mathbb{T}^* = \{ \textit{transmission}, \textit{sensortobus}, \textit{overecu}, \\ \textit{bustoactuator}, \textit{sensortoswc}, \textit{overswc}, \\ \textit{swctobus}, \textit{bustoswc}, \textit{swctoactuator}, \perp \}$$

All segments of a system timing model must be of one of these nine types. Otherwise the TIMEX model is not consistent. This rule is formalized by Rule (5.4).

$$\forall s \in S : \textit{segtype}(s) \neq \perp. \quad (5.4)$$

5.4.5 Segment Sequences

Based on the defined set of possible segment types another consistency rule comes up when chains of segments are considered. That is, after a segment of a certain type only certain other types make sense. For example, after data transmission on a bus, modeled as a *transmission segment* with a hop of type *transmitted* as response in TIMEX, there must follow a segment that uses that hop as stimulus. Of course this induces a certain type of the following segment.

First we define the functions *successors* and *predecessors*, which determine the next and previous segments of a given segment in a system timing. Note that segments are collected in a set by a system timing model. Chaining of segments is modeled implicitly by the relations that unfold by the common use of hops as stimuli and responses in several segments.

For a segment $s = \langle \textit{stimulus}, \textit{response} \rangle$ function *successors* is defined as follows.

$$\begin{aligned} \textit{successors}(s) : s &\rightarrow \{s\} \\ \textit{successors}(s) &= \{ \textit{succ} \in S \mid \\ &\quad \textit{s.response} = \textit{succ.stimulus} \} \end{aligned}$$

Function *predecessors* for a segment $s = \langle \textit{stimulus}, \textit{response} \rangle$ is defined as follows.

$$\begin{aligned} \textit{predecessors}(s) : s &\rightarrow \{s\} \\ \textit{predecessors}(s) &= \{ \textit{pred} \in S \mid \\ &\quad \textit{pred.response} = \textit{s.stimulus} \} \end{aligned}$$

Using the functions *segtype*, *successors* and *predecessors* we can formulate a couple of conditions for the types of consecutive segments, which are collected in Rule (5.5).

$$\begin{aligned}
\text{segtype}(s) = \text{sensortobus} &\Rightarrow \forall \text{succ} \in \text{successors}(s) : & (5.5) \\
&\text{segtype}(\text{succ}) = \text{transmission}. \\
\text{segtype}(s) = \text{sensortobus} &\Rightarrow \text{predecessors}(s) = \emptyset. \\
\text{segtype}(s) = \text{bustoactuator} &\Rightarrow \text{successors}(s) = \emptyset. \\
\text{segtype}(s) = \text{bustoactuator} &\Rightarrow \forall \text{pred} \in \text{predecessors}(s) : \\
&\text{segtype}(\text{pred}) = \text{transmission}. \\
\text{segtype}(s) = \text{transmission} &\Rightarrow \forall \text{succ} \in \text{successors}(s) : \\
&\text{segtype}(\text{succ}) = \text{bustoactuator} \vee \\
&\text{segtype}(\text{succ}) = \text{overecu} \vee \\
&\text{segtype}(\text{succ}) = \text{bustoswc}. \\
\text{segtype}(s) = \text{transmission} &\Rightarrow \forall \text{pred} \in \text{predecessors}(s) : \\
&\text{segtype}(\text{pred}) = \text{sensortobus} \vee \\
&\text{segtype}(\text{pred}) = \text{overecu} \vee \\
&\text{segtype}(\text{pred}) = \text{swctobus}. \\
\text{segtype}(s) = \text{overecu} &\Rightarrow \forall \text{succ} \in \text{successors}(s) : \\
&\text{segtype}(\text{succ}) = \text{transmission}. \\
\text{segtype}(s) = \text{overecu} &\Rightarrow \forall \text{pred} \in \text{predecessors}(s) : \\
&\text{segtype}(\text{pred}) = \text{transmission}. \\
\text{segtype}(s) = \text{sensortoswc} &\Rightarrow \forall \text{succ} \in \text{successors}(s) : \\
&\text{segtype}(\text{succ}) = \text{overswc}. \\
\text{segtype}(s) = \text{sensortoswc} &\Rightarrow \text{predecessors}(s) = \emptyset. \\
\text{segtype}(s) = \text{swctoactuator} &\Rightarrow \text{successors}(s) = \emptyset. \\
\text{segtype}(s) = \text{swctoactuator} &\Rightarrow \forall \text{pred} \in \text{predecessors}(s) : \\
&\text{segtype}(\text{pred}) = \text{overswc}. \\
\text{segtype}(s) = \text{swctobus} &\Rightarrow \forall \text{succ} \in \text{successors}(s) : \\
&\text{segtype}(\text{succ}) = \text{transmission}. \\
\text{segtype}(s) = \text{swctobus} &\Rightarrow \forall \text{pred} \in \text{predecessors}(s) : \\
&\text{segtype}(\text{pred}) = \text{overswc}. \\
\text{segtype}(s) = \text{bustoswc} &\Rightarrow \forall \text{succ} \in \text{successors}(s) : \\
&\text{segtype}(\text{succ}) = \text{overswc}. \\
\text{segtype}(s) = \text{bustoswc} &\Rightarrow \forall \text{pred} \in \text{predecessors}(s) : \\
&\text{segtype}(\text{pred}) = \text{transmission}. \\
\text{segtype}(s) = \text{overswc} &\Rightarrow \forall \text{succ} \in \text{successors}(s) : \\
&\text{segtype}(\text{succ}) = \text{swctobus}. \\
\text{segtype}(s) = \text{overswc} &\Rightarrow \forall \text{pred} \in \text{predecessors}(s) : \\
&\text{segtype}(\text{pred}) = \text{bustoswc} \vee \\
&\text{segtype}(\text{pred}) = \text{sensortoswc}.
\end{aligned}$$

Note that for simplicity reasons we do not allow the following two types of segments in TIMEX, although they would exist according to our collaboration workflows of Section 3.1.2:

- It is possible that sensor and actuator interfaces are exposed to the environment technically by the same ECU, i.e. the function is completely provided by one ECU. If that function has a function-triggered timing constraint there must be an according function event chain and a constraint for that chain in the function timing model of TIMEX. The segmentation in that case would be simple. It would just be one segment with the same stimulus and response hops as the chain itself, because the function is provided by only one subsystem. Such kind of segments would have an own type, *sensor-to-actuator segment*, that is not in the segment types T^* defined above. Additionally Rule (5.5) would have to be extended for that case.
- Another example that is not supported by the TIMEX model is the sending of data from one SWC to another on the same ECU. The segment types above assume that a SWC always sends its data either to the bus or to an actuator. Accordingly a SWC receives its data either from a sensor or the bus. Again, it would be possible to introduce a new segment type *swc-to-swc* for that case and to extend Rule (5.5).

We focus on function event chains, which involve several subsystems to show our methodology. This is not the case for the *sensor-to-actuator segment* example above. The other *swc-to-swc segment* type does involve several subsystems, because each of these SWC is an own subsystem if they are represented in TIMEX. However, in our approach of Chapter 6 we especially focus on the collaboration between an ECU developer and a system designer. Therefore the communication between two SWC is not relevant.

5.4.6 Chain Segmentation

As already mentioned the actual chain segmentation is not explicitly modeled in TIMEX. Rather the model holds segments and function event chains both as separate lists. Segments reference hops as their stimulus and response hops. Hops can be referenced multiple times by several segments. Furthermore hops can be referenced in the role of a stimulus hop and in the role of a response hop at the same time. Thereby the segmentation of all chains is built up implicitly. Consider the example TIMEX model of Figure 5.2 and its formalization in Section 5.3.5. The segmentation of the chain *damperControlChain* by the five segments is obvious although it is not explicitly modeled. It solely unfolds by the relation of the segments through their hop usage.

Our approach for modeling segments and chains is kind of a loose binding between these two element types. The connecting model element between a) several segments and b) segments and chains are hops and their multiple usage. There is no explicit "linking element". This approach has the advantage of modeling freedom and model simplicity. An n-to-m-relation of chains and segments can easily be assembled. The disadvantage however is that the model can become structurally inconsistent and incorrect. For this reason another consistency rule is set up.

We declare that in a consistent TIMEX model every function event chain must be refined by a path of segments (also called a *chain of segments*), such that

1. the stimulus hop of every chain is used as stimulus hop in a segment;
2. starting from this segment there is a chain of segments, connected by the usage of a common hop as one segment's response hop and the other segment's stimulus hop;
3. and the chain of segments ends when one segment's response hop is equal to the chain's response hop;
4. so that there are no gaps and overlaps in the segmentation of the chain.

To formalize the requirement of a consistent segmentation of all chains we first declare the TIMEX structure as a graph and then make use of well-proven graph analysis techniques known from graph theory. Similar to a directed acyclic graph (DAG), the set of hops H is a set of vertices and the set of segments S is a set of edges. Every chain f is a path through the graph, from one vertex (the chain's stimulus hop) to another (the chain's response hop), passing several edges along the way. So, a chain is consistently modeled and refined with a set of segments, if a path through the graph from the stimulus hop to the response hop exists.

The idea to figure out whether each chain is segmented consistently is to calculate the *transitive closure* of the TIMEX graph and to find every chain in that transitive closure. Given a directed acyclic graph $G = (H, S)$ with a set of vertices H (in our case all hops) and a set of edges S (in our case all segments). The transitive closure is a graph $G^* = (H, S^*)$, whose set of edges S^* contains all edges of S and additionally an edge $\langle s, r \rangle$, if there is path from s to r in G . The graph G^* is called transitive closure of graph G . If every chain is an element of S^* then the segmentation is consistent.

The algorithm to find the transitive closure of a TIMEX graph is based on the Floyd-Warshall algorithm [97], which is used for finding the shortest paths between every pair of vertices in a weighted and potentially directed graph. Given a TIMEX graph $G = (H, S)$ the following pseudocode algorithm 5.4.1 calculates the transitive closure $G^* = (H, S^*)$.

Algorithm 5.4.1: CALCTRANSITIVECLOSURE(H, S)

```

 $S^* := S$ 

for each  $i \in H$ 
  for each  $j \in H$ 
    do  $\left\{ \begin{array}{l} \text{for each } k \in H \\ \text{do } \left\{ \begin{array}{l} \text{if } \langle i, k \rangle \in S^* \text{ and } \langle k, j \rangle \in S^* \text{ and not } \langle i, j \rangle \in S^* \\ \text{then } \text{add}(\langle i, j \rangle, S^*) \end{array} \right. \end{array} \right.$ 

return ( $S^*$ )

```

Based on the transitive closure $G^* = (H, S^*)$ of a given TIMEX model the consistency rule for chain segmentation is given by Rule (5.6).

$$\forall \langle s, r \rangle \in F : \exists \langle s, r \rangle \in S^* \quad (5.6)$$

5.4.7 Chain Constraints

The purpose of TIMEX finally is to capture function-triggered timing constraints for a given system model. The target model elements to do this are function event chains. These basically are modeled, because constraints shall be captured. In other words in a TIMEX model it does not make sense, to model function event chains, which are not used by any timing constraint. A model, which contains such unconstrained function event chains, actually is incomplete or incorrect. Therefore the final consistency rule we suggest is that every function event chain must be the target of at least on timing constraint.

For every chain f there must be one constraint of any of the three constraint type specific sets that targets f . The consistency rule for constrained function event chains is formalized by Rule (5.7).

$$\begin{aligned} \forall f \in F : & \quad (5.7) \\ & \exists l \in LC : l.target = f \vee \\ & \exists t \in TC : t.target = f \vee \\ & \exists s \in SC : f \in s.target. \end{aligned}$$

5.5 Development Methodology based on TIMEX

In this section we describe our methodology for automotive system development based on the TIMEX model presented before. The methodology focuses on the role system designer as described in Section 3.1.1. This means that the system designer performs all mentioned steps. The methodology is mainly following an idealized top-down approach for system development, which was already outlined in Section 2.2. In addition to the rather general design flow overview of that section, the methodology description now details the development process and most notably adds a distributed development and timing focus to it. In practice of course system development is more complex and is subject to many restrictions and prerequisites. Here, the idealized methodology is sufficient to reflect the basic problem, embed our proposed approach using TIMEX, and apply our constraint logic programming solution, which is presented in the subsequent Chapter 6.

Figure 5.3 depicts our methodology. It consists of six steps. The first three steps are performed manually by engineering work and basically cover the development of the main parts of the system model (functional architecture, logical architecture, hardware topology model, and deployment model) and the definition of an according TIMEX function timing model and system timing model. The technical founding of the manual part is covered by the system model description in Section 3.3 and the TIMEX model described in this Chapter 5. Steps 4, 5 and 6 are performed automatically, i.e. using algorithms. The

result of these last three steps is an appropriate mapping of the previously, manually defined function-triggered timing constraints to subsystem timing requirements and, based on the requirements, the rest of the system model, namely the communication model and the execution model. Steps 5 and 6 are performed iteratively, because it is expected that a set of requirements, which is fulfilled by all subsystems, is not found instantly after one iteration in practice. Another reason for the iteration is that changes during the system design process can force a re-design of the system's timing configuration, which is driven by a new requirement value generation in our approach. Steps 5 and 6 represent our formalization approach of today's informal negotiation process between a system designer and the suppliers. The technical details of the automated and iterative part of our methodology is described in detail in Chapter 6.

The single steps in Figure 5.3 are visualized in a way that emphasizes the dependency between the steps. The temporal order of the methodology steps is somewhat implicit along with the dependency information. Temporal relations like ordering or concurrency of the steps are not further investigated here. In the following each step is described in detail.

5.5.1 Define Function Timing

The first step is the definition of the function timing model as described in Section 5.3.3 where all function-triggered timing constraints for a given functional architecture are collected. Function-triggered timing constraints are implementation-independent, so no logical architecture and no technical architecture is necessary in this step. A functional architecture however is needed to define its according function timing.

Function timing usually is defined early in the system development process. Thus we assume that also the functional architecture is developed in an early stage and before all following implementation details. Therefore the functional architecture offers its interface events, which are available for the function timing model. Each such event can be characterized as a hop for the function timing model. These can be used for the definition of function event chains, which in turn are target of function-triggered timing constraints. In function timing these hops typically are "end-events" of signal paths like external events at sensors and actuator, i. e. the system's interface. For simplicity reasons we do not support other than interface events as hops for function timing. Our functional architecture model however could easily be extended in a way that more than only sensor and actuator interfaces are available for functions. In the methodology as described here, in the first step 1 no other hops than sensor and actuator hops are available, thus the function timing model also does not support other hops.

A function timing model always defines timing constraints for the functional architecture model to which it is attached, or which it references. However, the functional architecture model can be used for a refinement in several system models and the according function timing model thus is re-used. This concept makes function timing independent from an implementation, i. e. from a concrete completed system model.

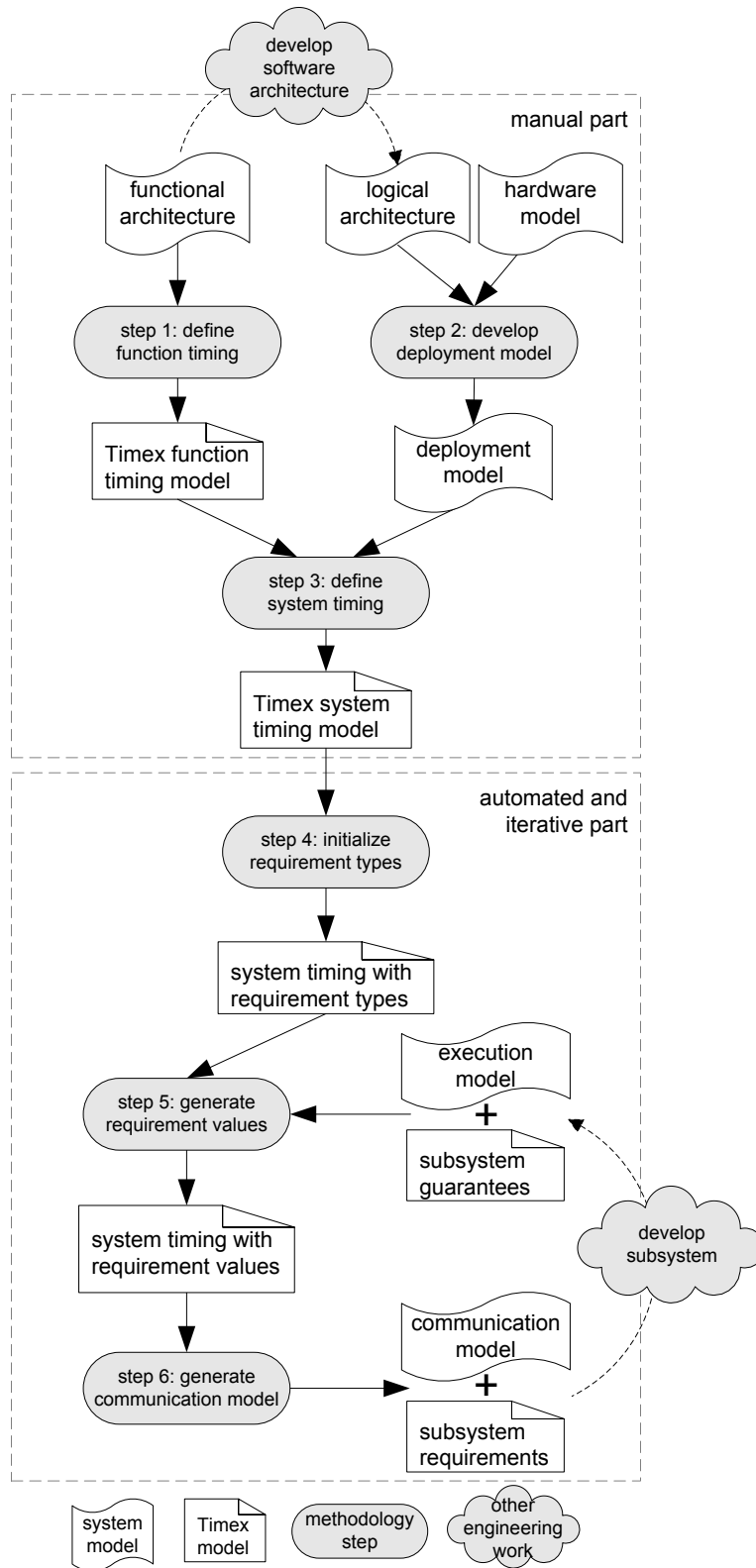


Fig. 5.3. Methodology for distributed development of automotive real-time systems based on TIMEX and our system model.

5.5.2 Develop Deployment Model

The second step is to develop the deployment model according to Section 3.3.5. The input of this step is the logical architecture and the hardware topology model. As indicated by the cloud symbol in Figure 5.3 and already brought up in Section 3.3.4 there is rather complex engineering work necessary to develop the logical software architecture for a given functional architecture. We neglect this step and simply assume an appropriate logical architecture model as well as the hardware topology model as given input for step 2.

The development of the deployment model however is explicitly mentioned as a step in our methodology although at a glance it is not directly related to timing. The reason however is that it indirectly influences the timing behavior and this step's output – the deployment model – is an essential artifact of the methodology and influences the subsequent timing related steps. In Definition 5.3 we call such kind of engineering work a *design decision*, which influences the system timing.

This step also reveals which message must be represented as a signal to be sent over a remote connector. The output of this step is the deployment model. Signals do not explicitly exist in the model. However the deployment model implicitly defines the information, which message must be represented as a signal. Therefore we defined the set of remotely exchanged messages L_{remote} in Section 3.4.3.

5.5.3 Define System Timing

A deployment model and the according function timing model are input for the definition of the system timing as described in Section 5.3.4. A logical architecture and the most important parts of the technical architecture (namely the hardware topology model and the deployment model) are now available for TIMEX. These reveal additional observable events at software component ports and regarding the communication of signals. These observable events can be used as hops by the system timing.

All additional hops of the system timing must be on the signal path between the interface hops of the function timing. We assume that the system designer as central role knows the responsibilities and subsystems within the entire system and thus can identify which observable events separate two teams. These qualify as hops and finally the function event chain segmentation can be performed. Each segment created that way belongs to one responsibility, i.e. to one development team, and covers this team's subsystem.

The output of step 3 is the TIMEX system timing that makes up the TIMEX model together with the function timing developed in step 1.

5.5.4 Initialize Requirement Types

The first automated yet not iterative step is what we call the *initialization of requirement types*.

Definition 5.9. *All function-triggered timing constraints of a TIMEX function timing model are implementation-independent. A TIMEX system timing provides an implementation-dependent, i. e. system-specific, segmentation of function event chains. The goal of our approach is to get subsystem requirements that can be distributed independently from the function-triggered timing constraints. Therefore the needed types of requirements for selected hops and segments must be determined once. It depends on the function-triggered timing constraints and the system details, which hops and segments must have which types of requirements. This determination process is called initialization of requirement types.*

The initialization of requirement types must be performed once for each use of a function timing in a specific system context. The initialization follows clear rules and can therefore be perfectly automated. The process and the different types of available requirements are described in Section 6.2.

Technically, every hop and every segment can be target of a requirement and its according guarantee. The assignment of these types is formally described in Section 6.2. This way a system timing model carries the requirement and guarantee values, which are necessary later in the methodology. The initialization solely determines which of all hops and segments must get which requirement type, so that a mapping of function-triggered timing constraints to requirements is technically possible. The goal is to be able to express all function-triggered timing constraints by system timing requirements. The mapping process also enables a tracing from function-triggered timing constraints to requirements later.

5.5.5 Generate Requirement Values

In the previous step 4 the needed system-specific requirement types have already been initialized, which is important for deriving subsystem specific requirements. These requirements are derived, we also say *generated*, in this methodology step 5. In other words, the hops and segments, which have been selected to carry requirement and guarantee values now get these values assigned automatically, i. e. using an algorithm.

So far during our methodology, all relevant design decisions have been captured in steps 1 to 3 and the TIMEX model has been prepared in step 4. Because it is expected that a kind of negotiation process is necessary until all subsystem requirements are fulfilled by according guarantees, steps 5 and 6 are performed iteratively.

The goal of this methodology step is to generate the values for all timing requirements of segments and hops of the TIMEX model that have been assigned a requirement type in the previous step 4. The requirement values must be chosen in a way that the fulfillment of all function-triggered timing constraints is ensured if all requirements are fulfilled by their guarantees.

There are many formal rules that can be applied in the value generation process. For this step we decided to use a constraint logic programming approach. Chapter 6 is dedicated to this generation process.

5.5.6 Generate Communication Model

In the last step 6 of the methodology the communication model, as described in Section 3.3.5, is generated. We assume that the system designer is responsible for the development of the system's communication model. For that reason this is an explicit step in the methodology, which is described from the perspective of the system designer. We declared the bus as an own subsystem. Because it is developed by the system designer, this step it is not part of the work done during the cloud called "develop subsystem" in Figure 5.3. The input of step 6 is the TIMEX model, which already contains the requirement values of the current iteration. The output of step 6 is a communication model according to the requirements that target the bus subsystem. Another output are the guarantees of the bus subsystem that, eventually, fulfill the requirements. The consequences of fulfillment and non-fulfillment of requirements is discussed in Chapter 6.

The communication model, or more precisely the requirements of the communication model, then is the basis for the work of all other teams, because their own requirements have been aligned to the rest system's requirements in step 5. They assume that the bus subsystem fulfills its requirements and concentrate on the fulfillment of their own subsystem-specific requirements. The communication model usually is used as common exchange specification between the system designer, who develops the bus network configuration, and the other subsystem developers, who are dependent from the communication model. In our methodology, this dependency is taken from the pure communication model to the requirements level.

Primarily, three tasks have to be performed in this step. First, all signals have to be grouped in frames. Second, a communication matrix must be generated using the communication information gathered from the logical software architecture and the deployment model. A communication matrix specifies which frame is sent and received by which ECU. Third, a scheduling of the frames, i. e. a communication model, has to be generated according to the communication type (FlexRay or CAN) and with respect to the requirement values generated before. The resulting communication model must fulfill all communication-related requirements.

5.5.7 Iterative Steps

After the last methodology step, each involved team develops its subsystem. This is indicated by the cloud in Figure 5.3, which takes subsystem requirements as input and has according guarantees as output. A subsystem is either an ECU, if the according segment belongs to a team in the role of an ECU developer, or a single software component, if the according segment belongs to a team in the role of a software component supplier. For each subsystem there are segments or hops that carry timing requirements that must be fulfilled. The teams must implement and configure their subsystem according to these requirements, if possible. Otherwise the system's correct timing behavior cannot be ensured. Again, the consequences of non-fulfillment are discussed in Chapter 6.

When the teams have implemented the subsystems, timing guarantees for each subsystem are handed back to the system designer. The guarantees are evaluated with respect to their requirements. If all requirements are met by the guarantees no action is required and the system is expected to fulfill all its function-triggered timing constraints. Otherwise all guarantees are input for the next iteration of steps 5 and 6 of the methodology and a new set of requirement values must be generated.

Today, this negotiation process takes place in an informal way. The TIMEX model and our proposed methodology formalize it. This is explained in the subsequent chapter.

Generating Subsystem Requirements in TIMEX with Constraint Logic Programming

This chapter is dedicated to the automated part of our methodology depicted in Figure 5.3. First we introduce three basic requirement types for the TIMEX model in Section 6.1. In Section 6.2 we explain the initialization of the necessary requirement types according to Definition 5.9, which is performed in methodology step 4. Furthermore our *constraint logic programming* solution to step 5 of the methodology is presented in Sections 6.3, 6.4 and 6.5, which is used for iterative requirement generation based on function-triggered timing constraints. The generation of a communication model as postulated by methodology step 6 is outlined in Section 6.6.

6.1 Requirements and Guarantees in a TIMEX Model

6.1.1 Possible Types of Requirements and Guarantees

The goal of the automated part of our methodology, i.e. steps 4 to 6, is to derive subsystem-specific requirements from the function-triggered timing constraints. The TIMEX model, as introduced and formalized in Section 5.3, does not provide any means to conveniently capture or express such requirements. Hops and segments are the TIMEX elements that shall carry requirement and guarantee information in the model. The reason for this is that hops and segments, in contrast to function event chains, are elements that refer to subsystems rather than the entire system. According to our methodology, system-wide timing constraints are specified *manually* by engineers. In this section we develop a formal concept to capture the requirement and guarantee information on a subsystem level in TIMEX, which is *automatically* generated in step 5.

First, we define the three possible types that can be used as requirements and guarantees for segments or hops. As we will explain in the subsequent Section 6.2 these types are adequate to map all three types of function-triggered timing constraints to subsystem requirements.

Latency Type

The first type is *latency*. Latency is a tuple of two integers, which are called *min* and *max*.

$$latency = \langle \mathbb{N}, \mathbb{N} \rangle$$

Given a variable of type latency, the two values can be accessed with the two operators *.min* and *.max*.

$$\begin{aligned} latency \ l &= \langle 1, 5 \rangle \\ minimum &= l.min \\ maximum &= l.max \end{aligned}$$

Offset Type

The second type is *offset*. An offset is a tuple of two integers, which are called *min* and *max*.

$$offset = \langle \mathbb{N}, \mathbb{N} \rangle$$

Given a variable of type offset, its min and max values can be accessed with the two operators *.min* and *.max*.

$$\begin{aligned} offset \ o &= \langle 0, 1 \rangle \\ minimum &= o.min \\ maximum &= o.max \end{aligned}$$

Triggering Type

The third type is *triggering*. A triggering is a tuple of one integer, which is called *period*.

$$triggering = \langle \mathbb{N} \rangle$$

Given a variable of type triggering, the period can be accessed with the operator *.period*.

$$\begin{aligned} triggering \ t &= \langle 10 \rangle \\ period &= t.period \end{aligned}$$

6.1.2 Assigning the Types to Segments and Hops

The TIMEX model as defined in the previous chapter must be extended in a way that segments and hops can carry requirement and guarantee information, if this is necessary. Therefore the three possible types of requirements and guarantees defined before are assigned to the according elements, using an appropriate formalism.

Assigning a Latency to a Segment

Each segment of a system timing can potentially have a latency requirement and an accompanying latency guarantee. Therefore two additional attributes of type *latency* are added to the definition of a segment. The semantics of a segment's latency requirement is the same as the semantics of a function event chain's latency constraint. It means that an occurrence of the response hop of the segment must be observed every time after a stimulus hop occurrence has been observed, within the minimum and maximum time duration given by the latency. Given a segment s , its latency requirement and latency guarantee can be accessed using the operators $s.lat_r$ and $s.lat_g$.

$$\begin{aligned} \text{segment } s &= \langle \text{stimulus}, \text{response} \rangle \\ \text{latency } \text{latencyRequirement} &= s.lat_r \\ \text{latency } \text{latencyGuarantee} &= s.lat_g \end{aligned}$$

Assigning an Offset to a Hop

Each hop – except sensor hops – of a TIMEX model, i.e. function timing and system timing hops, can potentially have an offset requirement and an accompanying offset guarantee. Sensor hops cannot have an offset because of Assumption 6.1.

Assumption 6.1 *We assume that sensor events occur nondeterministically at the system's border and are triggered by the environment, not by the system. The system can take notice of sensor events using a sensor interface. It cannot influence the occurrence of sensor events.*

Two additional attributes of type *offset* are added to the definition of a hop, namely an offset requirement and an offset guarantee. The semantics of a hop's offset requirement is that if an occurrence of the hop is observed, then it must be within the time duration specified by the minimum and maximum values of the offset requirement after a certain reference point.

In all our examples the reference point is always the cycle start of the time-triggered network. Thus, all offsets are expressed relatively to the cycle start event. Therefore we omit the explicit specification of the offset reference event for simplicity reasons.

Given a hop h , its offset requirement and offset guarantee can be accessed using the operators $h.off_r$ and $h.off_g$.

$$\begin{aligned} \text{hop } h &= \langle \text{someEvent} \rangle \\ \text{offset } \text{offsetRequirement} &= h.off_r \\ \text{offset } \text{offsetGuarantee} &= h.off_g \end{aligned}$$

Assigning a Triggering to a Hop

Each hop of a TIMEX model, i. e. in function timing and system timing, can potentially have a triggering requirement and an accompanying triggering guarantee. Therefore two additional attributes of type *triggering* are added to the definition of a hop. The semantics of a hop's triggering requirement is the same as the semantics of a function event chain's triggering constraint for that chain's stimulus. It means that the occurrences of the hop must be observed with the period specified by the triggering. Given a hop h , its triggering requirement and triggering guarantee can be accessed using the operators $h.trig_r$ and $h.trig_g$.

$$\begin{aligned} \text{hop } h &= \langle \text{someEvent} \rangle \\ \text{triggering } \text{triggeringRequirement} &= h.trig_r \\ \text{triggering } \text{triggeringGuarantee} &= h.trig_g \end{aligned}$$

Note that there is no synchronization requirement possible on system timing level for segments and hops, although a synchronization constraint in function timing exists. Synchronization always requires several function event chains to be referenced. This is not possible for segments and hops, because their requirements shall be independent from other segments and thus only affect the corresponding subsystem. That is, synchronization constraints for function event chains must be mapped to the three requirement types given above, similar to the other two constraint types, latency and triggering constraint.

6.1.3 Fulfillment of Requirements with Guarantees

In methodology step 5 a new set of requirement values is generated for the whole system timing model. As we will explain later in Section 6.3, these new values in each iteration of the process do not only depend on the timing constraints of the function timing, which they must fulfill. They also depend on the set of guarantees that is delivered to the system integrator by the subsystem developers. The whole re-generation process however is only initiated, if any requirement is not fulfilled by its guarantee, because otherwise the entire system timing is expected to be correct and no action is required.

It is thus necessary to formalize a fulfillment relation of a requirement and its guarantee, i. e. a definition when a guarantee fulfills its requirement. In the following we formalize this relation for all three requirement types.

Fulfillment of a Latency Requirement

A latency requirement specifies a minimum and a maximum latency for a segment. These values represent a required time interval for the valid segment latency. The guaranteed time interval must be within the required interval. Thus, the guaranteed minimum must be greater than or equal to the required minimum. The guaranteed maximum must be lower than or equal to the required maximum.

The fulfillment condition for a latency requirement and its guarantee is formalized by a predicate over a segment. Given a segment s Predicate (6.1) formalizes the fulfillment condition.

$$\begin{aligned} \text{Fulfills_Latency}(s) \equiv & s.\text{lat}_r.\text{min} \leq s.\text{lat}_g.\text{min} \wedge \\ & s.\text{lat}_r.\text{max} \geq s.\text{lat}_g.\text{max} \end{aligned} \quad (6.1)$$

Fulfillment of an Offset Requirement

An offset requirement specifies a minimum and a maximum offset for a hop. Again these values are a time interval. The offset guarantee time interval must be within the offset requirement time interval. That again means the guaranteed minimum must be greater than or equal to the required minimum. The guaranteed maximum must be lower than or equal to the required maximum.

The fulfillment condition for an offset requirement and its guarantee is formalized by a predicate over a hop. Given a hop h Predicate (6.2) formalizes the fulfillment condition.

$$\begin{aligned} \text{Fulfills_Offset}(h) \equiv & h.\text{off}_r.\text{min} \leq h.\text{off}_g.\text{min} \wedge \\ & h.\text{off}_r.\text{max} \geq h.\text{off}_g.\text{max} \end{aligned} \quad (6.2)$$

Fulfillment of a Triggering Requirement

An event triggering requirement for a hop specifies a period. The guaranteed period must be exactly the same as the required period. A special case for triggering requirements and guarantees occurs when the guaranteed period is an integer multiple or an integer divisor of the required period. If the guaranteed period is a multiple, i. e. n -times the required period, then the requirement is obviously not fulfilled, because only every n -th required hop occurrence can actually be observed according to the guarantee. If the guaranteed period is an integer divisor though, every required occurrence can indeed be observed. However there are superfluous and not explicitly required occurrences "in between". We also define this case as non-fulfillment, because superfluous event occurrences potentially cause a waste of processor or bus resources. Further we neglect a jitter of the period and assume that only the basic period is target of timing requirements.

The fulfillment condition for a triggering requirement and its guarantee is formalized by a predicate over a hop. Given a hop h Predicate (6.3) formalizes the fulfillment condition.

$$\text{Fulfills_Triggering}(h) \equiv h.\text{trig}_r.\text{period} = h.\text{trig}_g.\text{period} \quad (6.3)$$

6.2 Initializing Requirement Types

6.2.1 Initialization Basics

First we summarize the basics of the requirement type initialization process. After that we formalize its goal. As defined earlier there are three possible timing constraint types in TIMEX that have to be mapped to a set of timing requirements, which also are available in three types. The initialization of necessary types is mandatory for further methodology steps. From an exclusive function-timing perspective it is not clear, which segments and hops of the system timing must have which type of requirement if at all.

Several design decisions, which are captured by a system timing model, influence the necessary requirement types.

- *The logical architecture* provides a refinement of functions to software components and thus reveals more data path details from the functions' sensor interfaces to their actuator interfaces.
- *The deployment model* of the technical architecture again offers more details of that data path, because now local and remote communication can be identified. So parts of the data path potentially include bus communication.
- *The collaboration workflows* influence which part of the data path is provided by which subsystem, what is reflected by the segment structure in the TIMEX system timing.
- Finally and most notably, the bus type of the hardware topology model influences which segments and hops of the system timing must be initialized with which type of requirement.

The entire tracing from function view to system view is possible with TIMEX. The details of the last point are discussed in this section. In Section 6.2.2 we show the initialization for an event-triggered system and in Section 6.2.3 for a time-triggered system.

For each of the three requirement types we assume a predicate that is true, if segment s or hop h has the requirement assigned to it.

$$Has_Latency(s) \equiv \begin{cases} \text{segment } s \text{ has a latency} \\ \text{requirement and guarantee.} \end{cases} \quad (6.4)$$

$$Has_Offset(h) \equiv \begin{cases} \text{hop } h \text{ has an offset} \\ \text{requirement and guarantee.} \end{cases} \quad (6.5)$$

$$Has_Triggering(h) \equiv \begin{cases} \text{hop } h \text{ has a triggering} \\ \text{requirement and guarantee.} \end{cases} \quad (6.6)$$

Definition 6.1. *The task of the initialization process is to define the predicates $Has_Latency$, Has_Offset and $Has_Triggering$, i. e. to assign a true/false value for each $s \in S$ and each $h \in H$ for the according predicates.*

The initialization process defined in Definition 6.1 is mandatory for the overall goal of the TIMEX methodology:

Definition 6.2. *The goal of the TIMEX methodology is to ensure the system function's correct timing behavior on subsystem level instead of system level. In other words, if every single subsystem requirement is fulfilled, the fulfillment of all function-triggered timing constraints shall be assured.*

Definition 6.2 can be formalized as follows.

Given a TIMEX model with a set of hops H and a set of segments S . We define a predicate $All_Requ_Fulfilled$ that is true, if the requirements of all segments in S and all hops in H are fulfilled. Predicate (6.7) formalizes this.

$$\begin{aligned}
 All_Requ_Fulfilled(S, H) \equiv \forall s \in S : Has_Latency(s) \Rightarrow & \quad (6.7) \\
 & Fulfills_Latency(s) \\
 \wedge & \\
 \forall h \in H : Has_Offset(h) \Rightarrow & \\
 & Fulfills_Offset(h) \\
 \wedge & \\
 \forall h \in H : Has_Triggering(h) \Rightarrow & \\
 & Fulfills_Triggering(h).
 \end{aligned}$$

Further the TIMEX methodology shall ensure the fulfillment of all timing constraints C of the function timing model. Consider Predicate (6.8), which is informally described.

$$All_Constraints_Fulfilled(C) \equiv \text{all constraints in } C \text{ are fulfilled.} \quad (6.8)$$

We do not provide a formal description of Predicate (6.8) because of Definition 6.2. Rather the following implication shall be assured.

$$All_Requ_Fulfilled(S, H) \Rightarrow All_Constraints_Fulfilled(C). \quad (6.9)$$

Following our approach, the fulfillment of a single timing constraint is not clearly visible later. Rather timing constraints are mapped to timing requirements once. A timing requirement can influence several timing constraints, because segments can be used in several function event chains. The expressiveness of all constraints and all requirements of course shall be the same, because this is the key prerequisite to enable the goal of Definition 6.2.

For the formalization of the initialization process we introduce some constructs and helper functions.

We already mentioned that an entire TIMEX model actually represents a directed acyclic graph. A consistent segmentation of a function event chain (see Definition 5.5) leads to one or more paths of segments from the event chains's stimulus hop to its response hop. A path formally is a set of segments with the property that the contained segments represent a continuous sequence from the chain's stimulus to its response. The set of all possible paths for a set of segments S is the power set of S , denoted by:

$$\mathcal{P}(S)$$

To gather all paths of a function event chain we assume the following helper function *paths*, which returns the set of paths of a function event chain. The returned type thus is the power set of $\mathcal{P}(S)$, for a given function event chain.

$$paths : F \rightarrow \mathcal{P}(\mathcal{P}(S)). \quad (6.10)$$

Based on this background information for the initialization process we now explain the details for each of the three timing constraint types for both event-triggered and time-triggered systems.

6.2.2 Initialization for Event-triggered Systems

For the mapping of timing constraints to timing requirements in an event-triggered system we make the following very basic assumption.

Assumption 6.2 *For an event-triggered system we assume that the control flow is always the same as the data flow.*

Assumption 6.2 means that along a data path, which is modeled as a chain of hops in TIMEX, data *and* control is handed over from one hop to another. Within the semantics of TIMEX the consequence of Assumption 6.2 is that the triggering of a segment's stimulus hop always triggers the segment's response hop. If the response hop is at the same time the stimulus of another segment, this in turn triggers the next according response hop. This way the triggering of a function event chain's stimulus hop reaches the function event chain's response hop transitively.

Latency Constraint Mapping

A latency constraint references a function event chain as its target and constrains the minimum and maximum latency of that chain. Due to the fact that every chain is segmented to one or more paths using segments, the latency constraint actually must be fulfilled by each of these potentially more than

one paths of the chain. Because of Assumption 6.2 we can assume that the latency of a path is equal to the accumulated latency of its segments. Therefore the initialization in this case assigns a latency requirement to every segment of every path of the chain of every latency constraint.

Given a set of latency constraints LC of a TIMEX model. Based on Predicate (6.4) the following condition holds in an event-triggered system and thereby initializes the mapping of all latency constraints to requirements.

$$\forall l \in LC : \forall P \in paths(l.target) : \forall s \in P : Has_Latency(s). \quad (6.11)$$

Synchronization Constraint Mapping

According to Assumption 6.2 – and as already utilized for the latency constraint mapping – the latencies of the single segments of a path accumulate to a path latency. Latency is a tuple of two integers, the minimum and maximum latency.

Definition 6.3. *Minimum and maximum latencies of segments accumulate in a path of segments in an event-triggered system because of Assumption 6.2. So a path itself actually has an accumulated minimum and maximum latency, which we call the **jitter** of the path.*

A synchronization constraint references a set of function event chains and a tolerance. According to its scope either all stimuli or all responses of the chains must occur within the specified tolerance. According to Definition 6.3 every path has a jitter. The jitter of every path must be within the specified tolerance. Therefore every segment that is affected by a synchronization constraint must have a latency requirement, similar to Equation (6.11).

Given a set of synchronization constraints SC of a TIMEX model. Based on Predicate (6.4) the following condition holds in an event-triggered system and thereby initializes the mapping of all synchronization constraints to requirements.

$$\forall s \in SC : \forall f \in s.target : \forall P \in paths(f) : \forall s \in P : Has_Latency(s). \quad (6.12)$$

Note that the initialization of latency and synchronization constraints potentially can lead to a redundant latency requirement assignment to a segment. This is the case, when a chain path is affected by both a latency and a synchronization constraint.

Triggering Constraint Mapping

Due to Assumption 6.2 the triggering of all hops of a path of segments implicitly is the same. In each segment of the path the stimulus hop triggers the

response hop and thus implies the same triggering to the response. Note that in practice the triggering along such a data path basically remains the same and keeps the period. However in event-triggered systems a deviation from the strict period often occurs due to path jitters. Richter et al. [73, 74, 29] developed a timing analysis approach for event-triggered systems, which explicitly respects such jitter effects along data paths. We neglect this deviation for our triggering constraints and just focus on the basic strict period. As a consequence, for requirements initialization of triggering constraints it is sufficient, to assign a triggering requirement to the stimulus hop of the function event chain, which is the target of the constraint.

TC is the set of triggering constraints. Based on Predicate (6.6) the following condition holds in an event-triggered system and thereby initializes the mapping of all triggering constraints to requirements.

$$\forall t \in TC : Has_Triggering(t.target.stimulus). \quad (6.13)$$

6.2.3 Initialization for Time-triggered Systems

Note that the initialization for an event-triggered system does not make use of the offset type. Offsets can only be applied in time-triggered systems, because they need a reference hop, which in our case is the cycle start in a time-triggered system. Basically, an offset is used like a latency in a time-triggered system, because there the control flow is not equal to the data flow. Actions like data transmission or software execution are triggered by the progression of time and not by the data itself.

Assumption 6.3 *In a time-triggered system the control flow is not directly given by the data flow because data transmission is triggered by the progress of time.*

Because of Assumption 6.3 it must be exactly specified when data is expected to be queued for transmission and when it is transmitted on the bus using appropriate requirements. Therefore an offset requirement can perfectly be used. Besides the minimum and maximum values, a general offset requirement could also specify a reference hop. As already mentioned we omit this reference, because in a time-triggered network, the reference hop typically is the cycle start of the network. This is sufficient for our approach, as expressed in Assumption 6.4.

Assumption 6.4 *Offset requirements to the cycle start of a time-triggered network are sufficient for our approach. We neglect general hop offsets, which relate a hop to another arbitrary hop other than the cycle start.*

Concluding, the main difference of requirement initialization for a time-triggered in contrast to an event-triggered system is the following. The transmission latency of each segment of type transmission in a time-triggered system is expressed by means of offset requirements, instead of latency requirements. Because of the cyclic repetition of time-triggered communication, also

a triggering must be specified for each queued and transmitted hop. Therefore an additional triggering requirement is necessary to map the initial constraints to a time-triggered network.

Latency Constraint Mapping

In a time-triggered system the data path of a chain, i. e. the chain of segments, can contain transmission segments. According to Assumption 6.3, these segments cannot be initialized with a latency requirement, because it does not fit to the control flow concept of such a network. The response does in this case not occur as a causal consequence of each occurrence of the stimulus hop, but because of the progression of time and the underlying static schedule. Instead those segment's stimulus and response hops are initialized with an offset and a triggering requirement. These require both hops' events to occur always at a certain time after the cycle start (i. e. at a certain so-called slot) and with a certain period. The time distance between stimulus and response hop of a transmission segment is expressed with offset requirements instead of a latency requirement.

Given a set of latency constraints LC of a TIMEX model. Based on Predicate (6.4), Predicate (6.5) and Predicate (6.6) the following condition holds in a time-triggered system and thereby initializes the mapping of all latency constraints to requirements.

$$\forall l \in LC : \forall P \in paths(l.target) : \forall s \in P : \tag{6.14}$$

$$segtype(s) = transmission \Rightarrow \begin{cases} Has_Offset(s.stimulus) \wedge \\ Has_Offset(s.response) \wedge \\ Has_Triggering(s.stimulus) \wedge \\ Has_Triggering(s.response) \end{cases}$$

$$segtype(s) \neq transmission \Rightarrow Has_Latency(s).$$

Synchronization Constraint Mapping

As mentioned in Section 5.5.1 and already formalized in Section 5.3.3 we only support sensor and actuator hops in the function timing model. These two hop types thus are the only possible types as stimuli and responses, respectively, that can be synchronized. Further there are the two collaboration workflows to be considered for requirements initialization in a time-triggered network. Table 6.1 summarizes the four possible cases and the segment types that must be synchronized.

The difference to synchronization constraint mapping in an event-triggered network is that there exists no continuous path of latency requirements that accumulate. In the time-triggered case the path is interrupted by the first transmission segment, which is not getting initialized with a latency but with two offset requirements, as described above. As a consequence, the offset requirement can already be used as a reference to calculate the path jitters

	ECU Integration	SWC Integration
Stimulus Synchronization:	sensortobus	sensortoswc
Response Synchronization:	bustoactuator	swctoactuator

Table 6.1. Possible types of the first and last segment of a function event chain in the two possible collaboration scenarios.

later. We detail the necessary requirements and the according initialization formalization for each of the four cases in the following.

Case 1: Stimulus Synchronization in ECU Integration

The first segment of a chain in this case is a *sensortobus* segment. Thus the segment itself needs a latency requirement, its response, which is a hop of type *queued*, needs an offset requirement.

Case 2: Response Synchronization in ECU Integration

The last segment of a chain in this case is a *bustoactuator* segment. Thus the segment itself needs a latency requirement, its stimulus, which is a hop of type *transmitted*, needs an offset requirement.

Case 3: Stimulus Synchronization in SWC Integration

The first segment of a chain in this case is a *sensortoswc* segment. Thus the segment itself needs a latency requirement. The next segment is of type *overswc* and also needs a latency requirement. The next segment is of type *swctobus* and also needs a latency requirement, its response, which is a hop of type *queued*, needs an offset requirement.

Case 4: Response Synchronization in SWC Integration

The last segment of a chain in this case is a *swctoactuator* segment. Thus the segment needs a latency requirement. The previous segment is of type *overswc* and also needs a latency requirement. Again the previous segment is of type *bustoswc* and also needs a latency requirement, its stimulus, which is a hop of type *transmitted*, needs an offset requirement.

Equation (6.17) formalizes the four cases described above as a condition that holds for time-triggered systems and thus initializes the requirement types needed for synchronization constraints.

For the formalization of Equation (6.17) we assume the two helper functions $next(s)$, which determines the next segment of $s \in S$ in a path of segments, and $prev(s)$, which determines the previous segment.

$$next : S \rightarrow S. \quad (6.15)$$

$$prev : S \rightarrow S. \quad (6.16)$$

$$\begin{aligned}
\forall s \in SC : \forall f \in s.target : \forall P \in paths(f) : \forall s \in P : & \quad (6.17) \\
segtype(s) = sensortobus \Rightarrow & \begin{cases} Has_Latency(s) \wedge \\ Has_Offset(s.response) \end{cases} \\
\wedge & \\
segtype(s) = bustoactuator \Rightarrow & \begin{cases} Has_Latency(s) \wedge \\ Has_Offset(s.stimulus) \end{cases} \\
\wedge & \\
segtype(s) = sensortoswc \Rightarrow & \begin{cases} Has_Latency(s) \wedge \\ Has_Latency(next(s)) \wedge \\ Has_Latency(next(next(s))) \wedge \\ Has_Offset(next(next(s)).response) \end{cases} \\
\wedge & \\
segtype(s) = swctoactuator \Rightarrow & \begin{cases} Has_Latency(s) \wedge \\ Has_Latency(prev(s)) \wedge \\ Has_Latency(prev(prev(s))) \wedge \\ Has_Offset(s.stimulus) \end{cases}
\end{aligned}$$

Triggering Constraint Mapping

For a triggering constraint in an event-triggered network it was sufficient to assign a triggering requirement only to the first hop of the data path, because of Assumption 6.2. In a time-triggered network however the hops do not get triggered one after another transitively through the chain. It is thus necessary to initiate some hops with additional triggering requirements. Because of Assumption 6.3 again stimulus and response hops of transmission segments must be initialized with triggering requirements.

$$\begin{aligned}
\forall t \in TC : \forall P \in paths(t.target) : \forall s \in P : & \quad (6.18) \\
segtype(s) = transmission \Rightarrow & \begin{cases} Has_Triggering(s.stimulus) \wedge \\ Has_Triggering(s.response) \end{cases}
\end{aligned}$$

Note that the initialization can again be redundant for the different constraint types. Some types may be mapped to the same requirements for segments and hops. Or in other words, the same requirement can be a part of several constraints.

6.2.4 Example

Consider the TIMEX example of Figure 5.2. The only function event chain *damperControlChain* is already segmented to five segments, based on the software mapping design decision and the fact that all three ECUs are delivered by individual ECU developers. There is no SWC supplier involved in

this example. The communication bus type as another design decision finally also influences the type of requirements that segments and hops of the system timing have to fulfill.

We initialize the requirement types both for the event-triggered and the time-triggered case according to the rules of this section.

Example Initialization as an Event-triggered System

First we assume an event-triggered bus in the hardware topology model. For the latency constraint in the example we have to ensure that Predicate (6.11) becomes true. This can be achieved by the following formal definition of Predicate (6.4), which we only defined informally before.

$$\begin{aligned} \text{Has_Latency}(\text{sensor}) &:= \text{true}. \\ \text{Has_Latency}(\text{transmitM1}) &:= \text{true}. \\ \text{Has_Latency}(\text{controller}) &:= \text{true}. \\ \text{Has_Latency}(\text{transmitM2}) &:= \text{true}. \\ \text{Has_Latency}(\text{actuator}) &:= \text{true}. \end{aligned}$$

For all hops of the set of TIMEX hops H of the example Predicate (6.5) and Predicate (6.6) simply are defined as follows.

$$\begin{aligned} \forall h \in H : \text{Has_Offset}(h) &:= \text{false}. \\ \forall h \in H : \text{Has_Triggering}(h) &:= \text{false}. \end{aligned}$$

Predicate (6.12) and Predicate (6.13) must not be fulfilled, because neither synchronization nor triggering constraints exist in the example and need to be mapped.

Example Initialization as a Time-triggered System

If we assume a time-triggered network in the hardware topology model, other requirement types are necessary to map the same timing constraint to the same elements of the system timing. To fulfill Predicate (6.14) the following initialization of Predicate (6.4), Predicate (6.5), and Predicate (6.6) must be applied.

$$\begin{aligned}
& \textit{Has_Latency}(\textit{sensor}) := \textit{true}. \\
& \textit{Has_Latency}(\textit{transmitM1}) := \textit{false}. \\
& \textit{Has_Latency}(\textit{controller}) := \textit{true}. \\
& \textit{Has_Latency}(\textit{transmitM2}) := \textit{false}. \\
& \textit{Has_Latency}(\textit{actuator}) := \textit{true}.
\end{aligned}$$

$$\begin{aligned}
& \textit{Has_Offset}(\textit{sensor}(\textit{tiltSensor})) := \textit{false}. \\
& \textit{Has_Offset}(\textit{queued}(m1, \textit{sensorEcu})) := \textit{true}. \\
& \textit{Has_Offset}(\textit{transmitted}(m1, \textit{tbus})) := \textit{true}. \\
& \textit{Has_Offset}(\textit{queued}(m2, \textit{controllerEcu})) := \textit{true}. \\
& \textit{Has_Offset}(\textit{transmitted}(m2, \textit{tbus})) := \textit{true}. \\
& \textit{Has_Offset}(\textit{actuator}(\textit{damper})) := \textit{false}.
\end{aligned}$$

$$\begin{aligned}
& \textit{Has_Triggering}(\textit{sensor}(\textit{tiltSensor})) := \textit{false}. \\
& \textit{Has_Triggering}(\textit{queued}(m1, \textit{sensorEcu})) := \textit{true}. \\
& \textit{Has_Triggering}(\textit{transmitted}(m1, \textit{tbus})) := \textit{true}. \\
& \textit{Has_Triggering}(\textit{queued}(m2, \textit{controllerEcu})) := \textit{true}. \\
& \textit{Has_Triggering}(\textit{transmitted}(m2, \textit{tbus})) := \textit{true}. \\
& \textit{Has_Triggering}(\textit{actuator}(\textit{damper})) := \textit{false}.
\end{aligned}$$

Predicate (6.17) and Predicate (6.18) again must not be fulfilled because neither synchronization nor triggering constraints exist and thus the predicate initializations above are sufficient.

6.3 Generating Requirement Values - General Approach

6.3.1 Problem Analysis

In this section we analyze the problem of step 5 of our methodology depicted in Figure 5.3. Thus, we assume that function timing (step 1) as well as system timing (step 3) have been specified using TIMEX and that the necessary requirement types have been initialized (step 4) using the predicates of Section 6.2.

First we reemphasize our terminology, which is fundamental for this section. Timing constraints are system-wide, function-triggered and invariant. They must be fulfilled by an implementation. In contrast, timing requirements always refer to a single segment or a hop. Requirement values are generated and derived from the constraints. They are valid for one methodology iteration of steps 5 and 6. They are accompanied by an according guarantee that must fulfill the requirement. The suppliers of the according subsystem provide the guarantee values.

The problem of methodology step 5 is to transform the function-triggered timing constraints into single timing requirements for those segments and hops, which have been selected to have such requirements before in step 4, such that the fulfillment of all requirements by their according guarantees implies the fulfillment of all function-triggered timing constraints. If the transformation is done right, then the great advantage is that the timing correctness can be ensured on subsystem requirement level, rather than on system constraint level. In the so-called requirement generation also the provided guarantees shall be considered. The reason is that the output of each iteration shall not be an arbitrary or random new setup of requirement values. The new set of requirements shall be optimized with respect to the latest guarantees. The optimization yields to a fair distribution of time budgets across all involved subsystems and a termination of the iterative negotiation process.

6.3.2 General Algorithm Description

We propose a constraint logic programming approach to find the new requirement values as output of each iteration of methodology step 5. The algorithm differs for event-triggered and time-triggered systems, because different technical assumptions are made and also different requirement types are necessary, as described earlier in this chapter. The general algorithm overview that applies for both system types is depicted in Figure 6.1.

The general, system type-independent algorithm works as follows. For the first time when methodology step 5 is entered, all requirements have not yet been initialized with a value. Only the necessary types have been initialized in step 4. That means at the very first time of the iterative process, we call it *iteration 0*, an initial set of requirement values must be determined for all subsystems. All subsystem developers and suppliers now have to analyze their subsystem with respect to the given requirements and determine the according guarantees. This is indicated by the manual step called *determine subsystem guarantees* in Figure 6.1. These guarantees might or might not fulfill the requirements, or they might even be "better" than the requirement. Note, in Section 6.1.3 we defined fulfillment predicates in a way that both exact and better fulfillment are treated the same way and just mean "fulfilled". This is just sufficient for the algorithm step called *all requirements fulfilled*. In the next step called *generate new requirements* the distinction between exact and better fulfillment becomes important. In the detailed algorithm description we will clarify the meaning of *fulfilling a requirement better*, because the system types differ in that meaning. In any case, the guarantees are handed back to the system designer, who uses them as input for the next iteration. So in every further iteration, which we all call *iteration n*, the procedure is the same. The given guarantees are checked against the according requirements, which we already formalized in Section 6.1.3. If all requirements are fulfilled by their guarantees the entire timing setup of all subsystems is correct and thus all function-triggered timing constraints are fulfilled. Otherwise at least one subsystem cannot fulfill its requirement. So a new iteration is mandatory to fix that particular problem with the generation of a new set of requirement values that is more likely to be fulfilled by every subsystem. This is done by the following three steps, which are also illustrated in the algorithm overview in Figure 6.1.

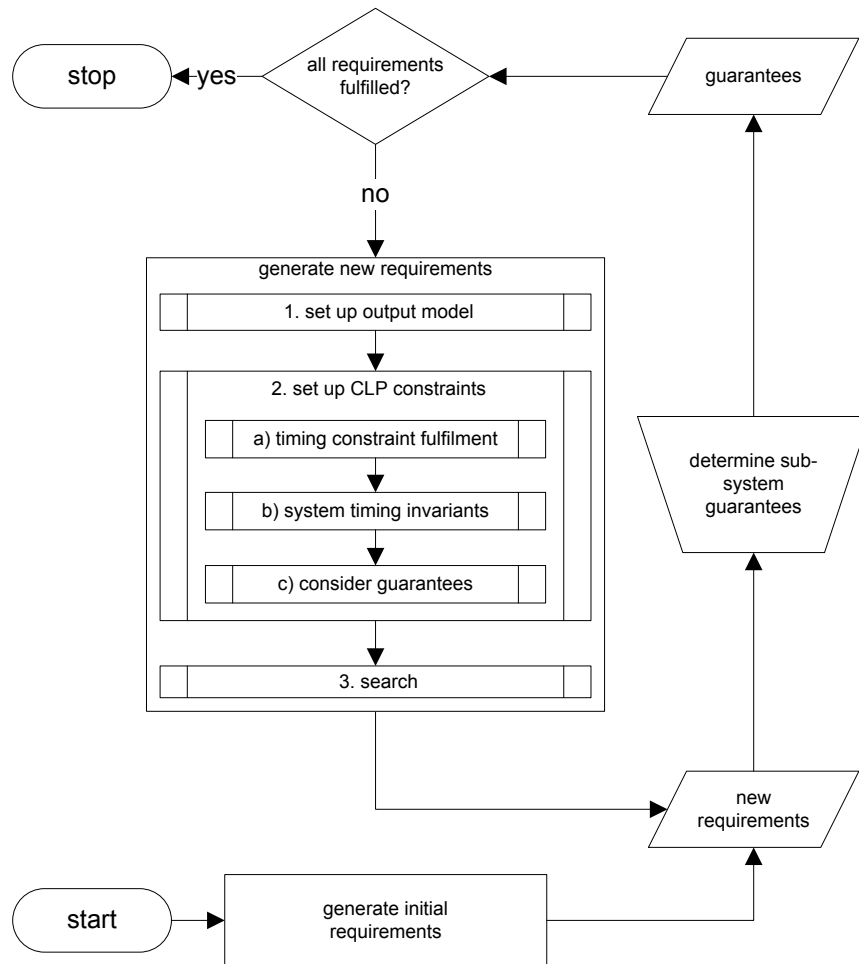


Fig. 6.1. Flowchart representing the algorithm to iteratively generate and modify subsystem requirements according to function-triggered timing constraints.

1. The output model, which is described in Section 6.3.3, is generated from the TIMEX system timing and the available segments and hops with requirements. The output model thus is the collection of all search variables and spans the whole solution space. A single solution is one instantiation of all segment and hop requirements with values for all their different requirement types. Which segments and hops need requirements at all was already initialized in methodology step 4, and formulated in Section 6.2.
2. All constraints (in the sense of CLP constraints) are set up over the search variables to narrow down the solution space reasonably. Some variable value instantiations do not make sense in practice for three reasons:
 - a) The requirement values must imply the fulfillment of all function-triggered timing constraints.
 - b) The system supports the concept of so-called *system timing invariants*, which represent fixed guarantees and thus variables, whose values are static, i. e. invariant.

- c) The new requirement values must be reasonable with respect to the current guarantees.

Especially the third case, relating the new requirement values reasonably with the current guarantees of all subsystems, is the most interesting part of the algorithms with respect to constraint logic programming.

- Standard tree search methods are used to search within the solution space for solutions that fulfill all CLP constraints. Thereby two strategies are possible and discussed in Chapter 7. Either one is interested in only one suitable solution or in the best of all possible solutions. In the first case the first solution that is found is returned. In the latter case the whole search tree or at least whole sub-trees are searched by the search algorithm to be able to find more than one possible solutions and to apply additional optimization metrics. We outline such metrics and optimization strategies in Chapter 8.2.

The processing step called *consider guarantees* plays an important role in the overall algorithm. Later we present two essential CLP approaches that realize this step. We developed one approach for event-triggered systems, called *shifting approach*, and another one especially for time-triggered systems, called *windowing approach*. Because of the importance of this step in general we emphasize some terminology at this point. The step *consider guarantees* of *iteration n* takes as input the old requirement values generated in *iteration n-1* and the according guarantee values. The output for *iteration n+1* are the new requirement values, which are instantiated in the output model by our search algorithm. Figure 6.2 visualizes the input and output relation.

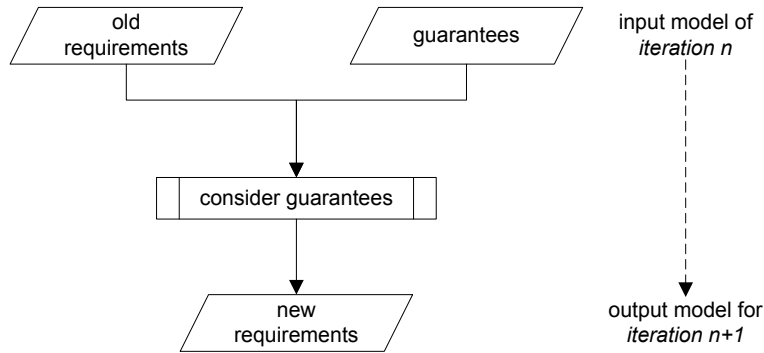


Fig. 6.2. The input and output of the processing step *consider guarantees*.

Technically, the difference between the two processing steps called *generate initial requirements* and *generate new requirements* is just the CLP constraint setup that is done before search. Such constraints that consider the current guarantees do not make sense in *iteration 0*. The fulfillment of timing constraints must be ensured by the output of both processing step variants, as well as the consideration of system timing invariants. Because of this difference Figure 6.1 depicts the variants explicitly as two different processing steps.

The implementation of the algorithm behind the processing step *generate new requirements* in Figure 6.1 differs for the two main system types. Especially the sub-steps called *timing constraints* and *consider guarantees* differ for both system types. Setting up the output model for our CLP approach and the definition of system timing invariants practically is the same for both event-triggered and time-triggered systems. These two common sub-steps are therefore explained in this section. The CLP constraint setup though completely differs for event-triggered and time-triggered systems. We explain both CLP implementations in the subsequent sections for each of the system types. The sub-step of triggering the search over all variables and finding a solution or an optimal solution is also discussed per system type.

6.3.3 Set up Output Model

The output model is the set of all variables that shall be instantiated in the search phase of the algorithm. For every requirement value of a segment or hop one integer variable is added to the output model. We denote all variables of the output model similar to the requirement and guarantee values of the usual TIMEX model with an additional asterisk (*) to indicate that the new requirement value of the output model is meant, not the old requirement value of the previous iteration $n - 1$ and not the guaranteed value. The entire output model thus again comprises the sets of segments S and hops H and represents a "copy" of the original model. The suffixes $_r$ and $_g$ indicate the old requirement value and the guarantee value in the normal TIMEX model as already introduced in Section 6.1.2. Summarizing, the following triple exists for each requirement value:

- suffix $_r$: old requirement value
- suffix $_g$: old guarantee value
- asterisk *: new requirement value within the output model

In constraint logic programming, an initial domain must be assigned to each such variable. Furthermore, some general problem-specific but system-independent constraints can be defined for all variables.

As initial domain for each variable we define the entire domain of positive integer values. That is, every latency, offset and triggering requirement can have an arbitrary positive integer value as its minimum and maximum or period value, respectively.

As first general problem-specific constraint we define that the minimum value of each latency requirement must be lower than or equal to the corresponding maximum value of that requirement. This is formalized by Constraint (6.19) over the set of all segments S .

$$\begin{aligned} & \text{Output_Model_Latencies}(S) \equiv \\ & \forall s \in S : s.lat*.min \leq s.lat*.max. \end{aligned} \tag{6.19}$$

Further we define a constraint that ensures that only a certain set of possible triggering requirement values $a, b, c, \dots \in \mathbb{N}$ is allowed. The background for this constraint is that the problem complexity is drastically reduced by it. In practice it is also common only to allow a certain standard set of possible periods, especially in time-triggered systems. Constraint (6.20) formalizes this over the set of hops H .

$$\begin{aligned} & \text{Output_Model_Triggerings}(H) \equiv \\ & \forall h \in H : h.\text{trig}^*.\text{period} \in \{a, b, c, \dots\}. \end{aligned} \quad (6.20)$$

As offset requirements cannot be used in event-triggered systems, Constraint (6.19) and Constraint (6.20) are sufficient output model constraints for those systems. The two constraints are also valid for the output model in case of a time-triggered system. For those additionally a constraint for offset requirements must be defined.

Similar to latency requirements, the minimum value of every offset requirement must be lower than or equal to the according maximum value of that requirement. Constraint (6.21) formalizes this over the set of hops H .

$$\begin{aligned} & \text{Output_Model_Offsets}(H) \equiv \\ & \forall h \in H : h.\text{off}^*.\text{min} \leq h.\text{off}^*.\text{max}. \end{aligned} \quad (6.21)$$

The usage of Constraint (6.19), Constraint (6.20), and Constraint (6.21) as CLP constraints narrows the solution space reasonably for time-triggered systems.

6.3.4 System Timing Invariants

Often some parts of the system are reused parts of previous developments. If such parts are used as black-boxes (for example an ECU) there is no configuration freedom and thus their timing guarantees are static and known in advance. Thus it is pointless to generate other requirement values than the ones known from the guarantees. Such requirements are called *invariant*. The system designer can be sure not to receive other guarantee values than the predefined ones. The subsystem developer can be sure not to get other requirement values than the ones known from the invariant definition. Our timing model as well as our requirement generation algorithm is capable of the concept of invariants.

System timing invariants can be defined to set certain requirements to a fixed value. This narrows the solution space, because the variable cannot take any other values. We assume a predicate *Invariant* which determines if invariant requirement values exists for the *min* and *max* latency of a certain segment s , the *min* and *max* offset of a certain hop h , or the triggering *period* of a certain hop h of the output model.

The predicate *Invariant* is used in the following three CLP constraints *Invariant_Latencies*(S) over all segments, *Invariant_Triggerings*(H) over all hops and *Invariant_Offsets*(H) over all hops.

$$\begin{aligned}
& \text{Invariant_Latencies}(S) \equiv & (6.22) \\
& \forall s \in S : \text{Invariant}(s, \text{minlatency}, \text{maxlatency}) \Rightarrow \\
& s.\text{lat}^*.\text{min} = \text{minlatency} \wedge s.\text{lat}^*.\text{max} = \text{maxlatency}.
\end{aligned}$$

$$\begin{aligned}
& \text{Invariant_Triggerings}(H) \equiv & (6.23) \\
& \forall h \in H : \text{Invariant}(h, \text{period}) \Rightarrow \\
& h.\text{trig}^*.\text{period} = \text{period}.
\end{aligned}$$

$$\begin{aligned}
& \text{Invariant_Offsets}(H) \equiv & (6.24) \\
& \forall h \in H : \text{Invariant}(h, \text{minoffset}, \text{maxoffset}) \Rightarrow \\
& h.\text{off}^*.\text{min} = \text{minoffset} \wedge h.\text{off}^*.\text{max} = \text{maxoffset}.
\end{aligned}$$

Constraint (6.24) can only be used for time-triggered systems, because we do not support offsets in event-triggered systems. Constraint (6.22) and Constraint (6.23) can be used for both system types.

6.4 Generating Requirement Values for Event-Triggered Systems

In the description of the requirement generation algorithm we suppose the step *generate new requirements* to be performed iteratively. We assume that all current subsystem guarantees are available before each iteration. The initiatory check whether all requirements are fulfilled by their guarantees is performed using Predicate (6.7), which is called *All_Requ_Fulfilled(S, H)*.

If Predicate (6.7) is *false* then the processing step *generate new requirements* is triggered. We describe this step using predicate logic. We denote those predicates, which act as a CLP constraint, by "*Constraint XY*" in our algorithm description.

6.4.1 Timing Constraint Fulfillment

According to Equation (6.11) every segment of a function event chain with latency constraint has got a latency requirement. So, for an event-triggered network we define the following Constraint (6.25) over the set of latency constraints *LC*, which is true, if the requirement values fulfill all latency constraint maximum and minimum values. As a CLP constraint, Constraint (6.25) narrows the solution space reasonably, because only requirement value variable instantiations are allowed, which fulfill all latency constraints.

$$\begin{aligned}
\text{Requirements_Fulfill_Latency_Constraints_ET}(LC) \equiv & \quad (6.25) \\
\forall l \in LC : \forall P \in \text{paths}(l.\text{target}) : \sum_{s \in P} s.\text{lat}^*.\text{max} \leq l.\text{max} & \\
& \wedge \\
\forall l \in LC : \forall P \in \text{paths}(l.\text{target}) : \sum_{s \in P} s.\text{lat}^*.\text{min} \geq l.\text{min}. &
\end{aligned}$$

By defining Equation (6.12) we already formalized that every segment of a function event chain with synchronization constraint needs a latency requirement. The segment latencies of a segment path accumulate. The resulting jitter of several paths must meet the tolerance of the synchronization constraint. We define Constraint (6.26) over the set of synchronization constraints SC to formalize the fulfillment of all such timing constraints.

$$\begin{aligned}
\text{Requirements_Fulfill_Synch_Constraints_ET}(SC) \equiv & \quad (6.26) \\
\forall sc \in SC : \text{maxpath}(sc) - \text{minpath}(sc) \leq sc.\text{tolerance} &
\end{aligned}$$

The functions $\text{maxpath}(sc)$ and $\text{minpath}(sc)$ determine the maximum and minimum path latency of all paths of all function event chains in the target set of a synchronization constraint sc . They are defined as follows.

$$\begin{aligned}
\text{maxof}(\text{max} : \forall f \in sc.\text{target} : \forall P \in \text{paths}(f) : \text{max} = \sum_{s \in P} s.\text{lat}^*.\text{max}) & \\
\text{minof}(\text{min} : \forall f \in sc.\text{target} : \forall P \in \text{paths}(f) : \text{min} = \sum_{s \in P} s.\text{lat}^*.\text{min}) &
\end{aligned}$$

Also triggering constraints shall be fulfilled if all requirements are fulfilled by their guarantees. This can only be the case, if the triggering requirements for the according stimulus hops get the right period value. According to the requirement type instantiation condition given by Equation (6.13) every stimulus hop in a chain of segments has got a triggering requirement. The correct triggering requirement value instantiation is formalized by Constraint (6.27) over the set of triggering constraints T .

$$\begin{aligned}
\text{Requirements_Fulfill_Triggering_Constraints_ET}(TC) \equiv & \quad (6.27) \\
\forall t \in TC : \forall P \in \text{paths}(t.\text{target}) : \forall s \in P : & \\
s.\text{stimulus}.\text{trig}^*.\text{period} = t.\text{period}. &
\end{aligned}$$

6.4.2 Consider Guarantees - The Shifting Approach

In this sub-procedure of our requirement generation algorithm the given guarantees are included. Before we explain our approach to express this inclusion by CLP constraints, we first summarize how requirement and guarantee values can relate to each other.

Note that we focus on the maximum values of latency requirements in the shifting approach, because we see them as the crucial negotiation target. Offset requirements are not used in event-triggered networks. Triggering requirements are usually not negotiated because they are not as flexible in the system configuration as latencies. They are rather statically defined beforehand. Our algorithm checks the correctness of triggering guarantees, but it does not negotiate them.

For a maximum latency requirement/guarantee pair of a single segment there are three possible cases:

1. The guarantee value is equal to the requirement value. This means the guarantee exactly fulfills the requirement for that segment.
2. The guarantee value is greater than the requirement value. This means the requirement for that segment is not fulfilled (violated), because it takes longer than required.
3. The guarantee value is smaller than the requirement value. This means the requirement is "over-fulfilled", because the guaranteed latency is even better, i. e. faster, than required.

Especially the third case is based on the fundamental Assumption 6.5 for our approach.

Assumption 6.5 *If the latency for a segment of a subsystem is smaller than required, we assume that the according subsystem developer does report this "over-fulfillment" in the segment's latency guarantee.*

Description of the Shifting Approach

Assumption 6.5 is fundamental because in our approach we try to redistribute spare latency to equalize not fulfilled latency requirements that might happen elsewhere in the system.

To formalize the problem of relating the new latency requirement values with the current latency guarantee values we introduce the concept of *shifting*, which is an abstraction of the underlying requirement values that actually are affected by the shifting procedure.

Case I: Horizontal Shifting

Consider a chain of segments, each of which having a latency requirement and an according guarantee. If one segment of the chain does not fulfill the latency requirement with its guarantee, this can be solved by using the spare latency of another segment in the same chain, which over-fulfills its requirement. Such an example is depicted in Figure 6.3. The latency constraint for the chain in this example is 20. The numbers above the segments are the old latency requirement value (first number) and the according guarantee value (second number). Segment A exactly fulfills its requirement. Segment B over-fulfills its requirement with some spare latency, whereas segment C violates

its requirement. The idea of horizontal shifting is to use over-fulfillment "horizontally" in the same chain. A solution of the timing problem for the next iteration in the example is to use the spare latency of segment B and provide it to segment C in the next iteration's latency requirement, because it is expected that segment C then can also fulfill its requirement and thus in the next iteration the timing problem most probably is solved. We call this procedure *horizontal shifting*.

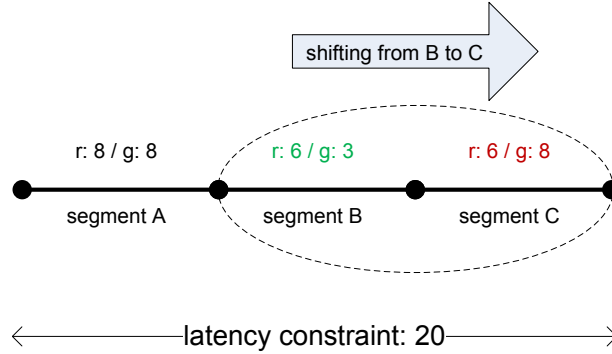


Fig. 6.3. Example for horizontal shifting.

The horizontal shifting example is summarized in Table 6.2. The table additionally shows the sum of requirement and guarantee values. Note that the guarantee sum of *iteration n* actually does not violate the latency constraint, which in this example is 20. The timing problem was locally at segment C, but it could be solved using the horizontal shifting approach.

segment	iteration n		iteration n+1	
	requirement	guarantee	requirement	guarantee
A	8	8	8	8
B	6	3	3	3
C	6	8	8	8
sum	20	19	19	19

Table 6.2. Complete example for horizontal shifting for an iteration i and the subsequent iteration $i+1$.

Case II: Vertical Shifting

Consider two chains of segments, again each with a latency requirement and an according guarantee. If a segment of one of the chains does not fulfill the latency requirement with its guarantee, this can be solved by using the spare latency of a segment of another chain on the same resource, which over-fulfills its requirement, as depicted in Figure 6.4. The latency constraint for both chains in this example is 20. Segments A, B, D and E exactly fulfill their requirements. Segment F over-fulfills its requirement with some spare latency, whereas segment C violates its requirement. The idea of vertical shifting on the same resource is to use over-fulfillment "vertically" on the same resource. In the example a solution of the timing problem for the next iteration is to use the spare latency of segment F and provide it to segment C in the next

iteration's latency requirement, because it is expected that segment C then can also fulfill its requirement and this timing problem is expected to be solved in the next iteration. Note that this vertical shifting is only possible, because the upper chain still fulfills its latency constraint, although segment C violates its requirement. In other words, the old requirement values were too strict, i. e. the overall latency requirement sum of the upper chain was smaller than the latency constraint. We call this procedure *vertical shifting*. It is based on Assumption 6.6.

Assumption 6.6 *If one resource holds several segments with latency requirements then we assume that the overall latency budget on that resource can be redistributed among all segments without causing fulfillment problems on that resource.*

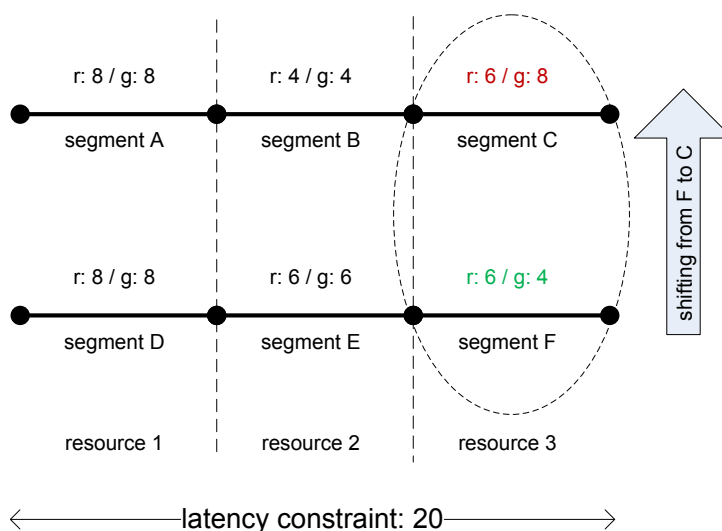


Fig. 6.4. Example for vertical shifting on the same resource for an iteration i and the subsequent iteration $i+1$.

The vertical shifting example is summarized in Table 6.3. The timing problem was locally at segment C, but it could be solved using the vertical shifting approach.

Case III: Diagonal Shifting

Similar to case II consider two chains of segments. If a segment of one of the chains does not fulfill the latency requirement with its guarantee, this can in some cases be solved by using the spare latency of a segment of another chain even on a different resource, which over-fulfills its requirement, as depicted in Figure 6.5. The latency constraint for both chains in this example again is 20. Segments A, B, D and F exactly fulfill their requirements. Segment E over-fulfills its requirement, whereas segment C violates its requirement. The idea of diagonal shifting across different resources is to use over-fulfillment "diagonally" from one resource on another one. In the example a solution of the timing problem for the next iteration is to use the spare latency of segment

segment	iteration n		iteration n+1	
	requirement	guarantee	requirement	guarantee
A	8	8	8	8
B	4	4	4	4
C	6	8	8	8
sum	18	20	20	20
D	8	8	8	8
E	6	6	6	6
F	6	4	4	4
sum	20	18	18	18

Table 6.3. Complete example for vertical shifting on the same resource for an iteration i and the subsequent iteration $i+1$.

E and provide it to segment C in the next iteration’s latency requirement, because it is expected that segment C then can also fulfill its requirement and the timing problem is expected to be solved in the next iteration. However in this case the spare latency cannot be directly given to segment C, because they both neither share the same chain, as in horizontal shifting, nor the same resource, as in vertical shifting. Rather in this case segment B can be used to "transfer" the spare latency, because it connects segments E and C (it shares the same resource as E and the same chain as C). So if the requirement for B is reduced, C can keep its guarantee without causing a chain latency constraint violation. A reduction of the requirement of segment B however is only possible, because the requirement for E is increased on the same resource. We call this procedure *diagonal shifting*. It is based on Assumption 6.5 and Assumption 6.6.

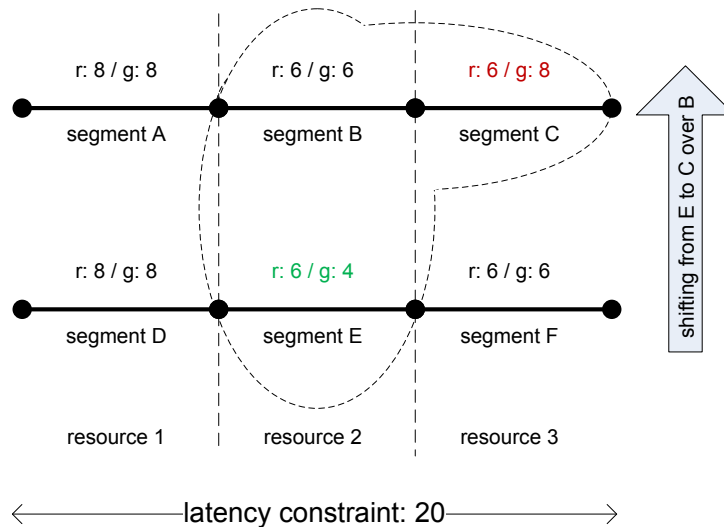


Fig. 6.5. Example for diagonal shifting across different resources for an iteration i and the subsequent iteration $i+1$.

The diagonal shifting example is summarized in Table 6.4. The timing problem was locally at segment C, but it could be solved using the spare latency of segment E over segment B.

segment	iteration n		iteration n+1	
	requirement	guarantee	requirement	guarantee
A	8	8	8	8
B	6	6	4	4
C	6	8	8	8
sum	20	22	20	20
D	8	8	8	8
E	6	4	6	6
F	6	6	6	6
sum	20	18	20	20

Table 6.4. Complete example for vertical shifting on different resources for an iteration i and the subsequent iteration $i+1$.

Note that the shifting approach is used to find a new set of requirement values, which has the potential to solve a timing problem with a high probability. The approach tries to solve timing problems (i. e. non-fulfillments) by redistributing free resources that come from over-fulfillments in a clever way. There is no certainty that the new requirements are fulfilled in the next iteration. However the approach helps in finding a reasonable requirement setup for the next iteration if a problem exists.

CLP Implementation of the Shifting Approach

The three examples of the previous section illustrate the idea behind our shifting approach. The three cases, horizontal, vertical and diagonal shifting, can be combined to redistribute latency budgets within a set of segments in an event-triggered system. The segments of course must be connected by either sharing the same chain or the same resource, which is supposed to be the case for usual automotive networks.

The description of the shifting approach so far is rather informal and example-driven. In this section we present our constraint logic programming implementation. It lines up with the other constraints of this Section 6.4 to accomplish the CLP constraint setup of the processing step *generate new requirements* depicted in Figure 6.1 for event-triggered systems.

The goal of the CLP realization is to express the entire shifting functionality on the level of a single segment. For each segment a set of individual constraints shall be constructed so that in total all three shifting cases are captured by CLP constraints. Note, the examples above still allow for many different solutions and thus many redistribution scenarios. The solutions that follow the rules of shifting shall be the ones that fulfill all constraints over the segments.

To construct the CLP constraint setup for each segment, we define a couple of predicates over a segment s , which are combined to form the overall constraint afterwards. For the constraints we use the following terminology:

- A segment has got a **buffer**, if the guaranteed latency is smaller than the old required latency.
- A segment has got a **backlog**, if the guaranteed latency is greater than the old required latency.
- The new latency requirement of a segment is **relaxed**, if it is greater than the guaranteed latency.
- The new latency requirement of a segment is **tightened**, if it is smaller than the guaranteed latency.

The following four predicates formalize each of these cases.

$$Has_Buffer(s) \equiv s.lat_r.max > s.lat_g.max \quad (6.28)$$

$$Has_Backlog(s) \equiv s.lat_r.max < s.lat_g.max \quad (6.29)$$

$$Relaxed(s) \equiv s.lat_g.max < s.lat^*.max \quad (6.30)$$

$$Tightened(s) \equiv s.lat_g.max > s.lat^*.max \quad (6.31)$$

The backlog and buffer predicates can recursively be extended from a single segment to a chain of segments, i. e. a path of a function event chain. Therefore we assume two helper functions $next(s)$ and $prev(s)$, which determine the next and previous segment of a segment path, as defined in Section 6.2.3. The predicates $Has_Next(s)$ and $Has_Prev(s)$ are *true*, if segment s has got a next and previous segment in a path. Given a path P , with $s, next(s), prev(s) \in P$ the following two conditions hold for the two functions:

$$Has_Next(s) \Rightarrow s.response = next(s).stimulus$$

$$Has_Prev(s) \Rightarrow s.stimulus = prev(s).response$$

Using the two helper functions we formulate the extended versions of the predicates Has_Buffer and $Has_Backlog$ that are applied for paths as follows.

$$\begin{aligned} Path_Buffer(s) \equiv & Has_Buffer(s) \vee \\ & Path_Buffer(next(s)) \vee \\ & Path_Buffer(prev(s)). \end{aligned} \quad (6.32)$$

$$\begin{aligned} Path_Backlog(s) \equiv & Has_Backlog(s) \vee \\ & Path_Backlog(next(s)) \vee \\ & Path_Backlog(prev(s)). \end{aligned} \quad (6.33)$$

We define the following Predicate (6.34) and Predicate (6.35), which are true if another segment exists on the same resource, whose latency will be relaxed or tightened, respectively, in the output model.

$$\begin{aligned} \textit{Relaxed_Res}(s) \equiv \exists s2 : s2.res = s.res \wedge \\ \textit{Relaxed}(s2). \end{aligned} \quad (6.34)$$

$$\begin{aligned} \textit{Tightened_Res}(s) \equiv \exists s2 : s2.res = s.res \wedge \\ \textit{Tightened}(s2). \end{aligned} \quad (6.35)$$

The shifting approach can now be expressed using all the defined predicates over a single segment, and thus form the final predicate that expresses our shifting approach in predicate logic. The predicate *Fulfills_Latency(s)* was already introduced by Predicate (6.1).

The following predicate over the set of all segments S formalizes the shifting approach. We explain all CLP constraints used in the predicate afterwards.

$$\begin{aligned} \textit{Shifting}(S) \equiv \forall s \in S : \\ \textit{Tightened}(s) \Rightarrow \textit{Path_Backlog}(s) \wedge \\ \textit{Relaxed_Res}(s) \end{aligned} \quad (6.36)$$

$$\begin{aligned} \wedge \\ \textit{Relaxed}(s) \Rightarrow \textit{Path_Buffer}(s) \wedge \\ \textit{Tightened_Res}(s) \end{aligned} \quad (6.37)$$

$$\begin{aligned} \wedge \\ \neg \textit{Fulfills_Latency}(s) \Rightarrow s.lat^*.max = s.lat_r.max \vee \\ s.lat^*.max = s.lat_g.max. \end{aligned} \quad (6.38)$$

Constraint (6.36) expresses that if any segment is tightened, then there must be a backlog in a path, which uses the segment. This means requirements are only tightened, if there is a reason. Further another segment on the same resource must be relaxed, because this increases the possibility of successful requirement fulfillment in the next iteration.

Similarly, Constraint (6.37) in turn expresses that if any segment is relaxed, then this can only be done because there is a path buffer. Otherwise there would be available buffer for the relaxing and the path latency would be exceeded. Further there must be another segment, which is tightened instead. Basically this constraint prevents random requirement relaxing.

The basic Constraint (6.38) initiates the entire shifting approach. It says that if a segment does not fulfill its latency requirement – i. e. the guaranteed latency is too long – then an action must be performed. Either the requirement is repeated or the guarantee is taken over. Either of these possibilities potentially leads to other actions and effects on other segments on the same resource or

on the same chain. If the old requirement is repeated although the guarantee wanted to have a greater latency, then another segment on the same resource must be relaxed instead. This can be found by vertical or diagonal shifting, which is implemented by the shifting predicate. If the guaranteed value is accepted, then as a consequence the overall path latency is too long and another segment in the chain must have a buffer. The buffer can be found by horizontal shifting. The triggered effects themselves have to follow the rules implied by the first two constraints.

6.4.3 Search

To search for a solution in the output model for an event-triggered system, all constraints must finally be connected. Given a set of segments S and a set of hops H as well as all function triggered timing constraints LC , SC and TC , which must be fulfilled if all requirements of S and H are fulfilled, a solution is only valid, if all the following constraints are met.

$$\begin{aligned}
 Search_ET(S, H, LC, SC, TC) \equiv & \quad (6.39) \\
 & Output_Model_Latencies(S) \wedge \\
 & Output_Model_Triggerings(H) \wedge \\
 & Invariant_Latencies(S) \wedge \\
 & Invariant_Triggerings(H) \wedge \\
 & Requirements_Fulfill_Latency_Constraints_ET(LC) \wedge \\
 & Requirements_Fulfill_Synch_Constraints_ET(SC) \wedge \\
 & Requirements_Fulfill_Triggering_Constraints_ET(TC) \wedge \\
 & Shifting(S).
 \end{aligned}$$

6.5 Generating Requirement Values for Time-Triggered Systems

The processing steps called *timing constraints*, *consider guarantees* and *search* of the general algorithm depicted in Figure 6.1 are formalized differently for time-triggered systems. Basically this is due to the completely different timing behavior of such systems, which we already mentioned in Assumption 6.2 and Assumption 6.3.

Before we specialize these processing steps for time-triggered systems, we introduce some general rules for requirement values, which arise due to the basic Assumption 6.3 for time-triggering.

6.5.1 General Rules for Requirement Values in Time-triggered Systems

An offset requirement has a minimum and a maximum value. The two values specify a time interval, in which a hop must occur. We call this time interval the *window* of the according hop.

Because offset requirements are only defined for the stimulus and response hops of transmission segments, two types of such windows exist: queuing and transmission windows. These are the offset intervals for queued and transmitted hops, respectively.

The general rules for requirement values in time-triggered systems can be expressed according to the possible segment types.

For an *externaltobus* segment the maximum latency must be greater than the period of its response. The reason for this is that external events occur potentially at any time. This means that in the worst case the sensor event just occurs right at the time the sensor SWC reads its sensor data. Thus it takes one more period to get the updated data. The maximum latency is in any case greater than the period. A smaller requirement can never be fulfilled. Further we define that the maximum latency of such a segment must be lower than twice the response period, because otherwise potentially sensor data could be overwritten. Furthermore the queuing window of *externaltobus* segments shall have a size of 0. In combination the requirements for an *externaltobus* mean that the sensor data shall be queued for transmission at a certain time point (offset) and with a certain maximum age (latency). Given the set of segments S Predicate (6.40) formalizes this.

$$\begin{aligned}
 \text{General_TT_Externaltobus}(S) &\equiv & (6.40) \\
 \forall s \in S : \text{segtype}(s) = \text{externaltobus} &\Rightarrow \\
 s.\text{lat}^*.\text{max} > s.\text{response}.\text{trig}^*.\text{period} \wedge \\
 s.\text{lat}^*.\text{max} < 2 * s.\text{response}.\text{trig}^*.\text{period} \wedge \\
 s.\text{response}.\text{off}^*.\text{min} = s.\text{response}.\text{off}^*.\text{max}
 \end{aligned}$$

For *transmission* segments there are also some rules that the requirement values must apply to. One obvious rule is that it is pointless to require an offset value, which is greater than the period of a hop, because such a requirement could never be fulfilled (see Assumption 2.3). Given the set of segments S Rule (6.41) formalizes this.

$$\begin{aligned}
 \text{General_TT_Transmission}(S) &\equiv & (6.41) \\
 \forall s \in S : \text{segtype}(s) = \text{transmission} &\Rightarrow \\
 s.\text{stimulus}.\text{off}^*.\text{max} < s.\text{stimulus}.\text{trig}^*.\text{period} \wedge \\
 s.\text{response}.\text{off}^*.\text{max} < s.\text{response}.\text{trig}^*.\text{period}
 \end{aligned}$$

For *overecu* segments a useful rule is that the input data must not be overwritten, or queued. This means that the maximum calculation latency after the input data has been read must be lower than the period of input data reception. Similar to *externaltobus* segments the queuing window shall have a size of 0. Again, Rule (6.42) formalizes these rules for a set of segments S .

$$\begin{aligned}
 \text{General_TT_Overecu}(S) &\equiv & (6.42) \\
 \forall s \in S : \text{segtype}(s) = \text{overecu} &\Rightarrow \\
 s.\text{lat}^*.\text{max} < s.\text{stimulus}.\text{trig}^*.\text{period} \wedge \\
 s.\text{response}.\text{off}^*.\text{min} = s.\text{response}.\text{off}^*.\text{max}
 \end{aligned}$$

As last general rule for requirement values in time-triggered systems we define that all triggerings of all transmission segment hops along a function event chain must have the same period. Otherwise so-called over- and under-sampling effects would appear. Such effects are discussed for example by Feiertag et al. [29]. In our requirements generation algorithm we want to exclude such effects. The basic principle of our approach can also be explained without the additional complexity, which is induced by over- and under-sampling effects. Rule (6.43) over a set of chains F formalizes the condition that all triggerings along a path must be equal.

$$\begin{aligned} \text{General_TT_Triggering}(F) \equiv & \quad (6.43) \\ \forall f \in F : \forall P \in \text{paths}(f) : \forall s \in P : & \\ \text{segtype}(s) = \text{transmission} \Rightarrow & \\ s.\text{stimulus.trig}^*.\text{period} = s.\text{response.trig}^*.\text{period} & \end{aligned}$$

6.5.2 Timing Constraint Fulfillment

All three kinds of timing constraints must be fulfilled by the system, which we now assume to be realized according to the time-triggered paradigm.

As described in Section 6.2.3 latency requirements cannot directly be expressed for transmission segments in a time-triggered network. According to Rule (6.14) every transmission segment of a function event chain with latency constraint has got an offset and a triggering requirement for its stimulus and response hop. The other segments are initialized with latency requirements. The calculation of the overall latency of a path of segments thus mixes latency requirements and offset requirements, which indirectly express latencies. The following Constraint (6.44) over the set of latency constraints LC is true, if the requirement values fulfill all maximum latency constraints in a time-triggered system.

$$\begin{aligned} \text{Requirements_Fulfill_Latency_Constraints_TT}(LC) \equiv & \quad (6.44) \\ \forall l \in LC : \forall P \in \text{paths}(l.\text{target}) : \sum_{s \in P} \text{latency}(s) \leq l.\text{max}. & \end{aligned}$$

The function $\text{latency}(s)$ determines the latency of segment s as follows. The "transmission latency" is actually the time interval between the maximum offset of the stimulus, which is a queued hop, and the maximum offset of the response, which is a transmitted hop.

$$\text{latency}(s) = \begin{cases} s.\text{response.off}^*.\text{max} - & \text{if } \text{segtype}(s) = \text{transmission} \\ s.\text{stimulus.off}^*.\text{max} & \\ s.\text{lat}^*.\text{max} & \text{else} \end{cases}$$

With Equation (6.17) we already introduced the four different cases that have to be distinguished for synchronization constraints. They arise from stimulus or response synchronization in either of the collaboration scenarios ECU integration or SWC integration.

Constraint (6.45) over the set of synchronization constraints uses helper functions to check, whether the requirements fulfill all synchronization constraints. The helper functions distinguish the four cases for synchronization constraints in a time-triggered network.

$$\begin{aligned} \text{Requirements_Fulfill_Synch_Constraints_TT}(SC) \equiv & \quad (6.45) \\ \forall sc \in SC : \text{maxdelay}(sc) - \text{mindelay}(sc) \leq & \text{sc.tolerance} \end{aligned}$$

$$\begin{aligned} \text{maxdelay}(sc) = \text{maxof}(\text{max} : \forall f \in \text{sc.target} : \\ \forall P \in \text{paths}(f) : \forall s \in S : \text{max} = \text{maxdelay}(s)). \end{aligned}$$

$$\begin{aligned} \text{mindelay}(sc) = \text{minof}(\text{min} : \forall f \in \text{sc.target} : \\ \forall P \in \text{paths}(f) : \forall s \in S : \text{min} = \text{mindelay}(s)). \end{aligned}$$

$$\text{maxdelay}(s) = \begin{cases} s.\text{response.of}f^*.\text{max} & \text{if } \text{segtype}(s) = \\ \quad -s.\text{lat}^*.\text{min} & \text{sensortobus} \\ s.\text{stimulus.of}f^*.\text{max} & \text{if } \text{segtype}(s) = \\ \quad +s.\text{lat}^*.\text{max} & \text{bustoactuator} \\ \text{next}(\text{next}(s)).\text{response.of}f^*.\text{max} & \text{if } \text{segtype}(s) = \\ \quad -\text{next}(s).\text{lat}^*.\text{min} - s.\text{lat}^*.\text{min} & \text{sensortoswc} \\ \text{prev}(\text{prev}(s)).\text{response.of}f^*.\text{max} & \text{if } \text{segtype}(s) = \\ \quad +\text{prev}(s).\text{lat}^*.\text{max} + s.\text{lat}^*.\text{max} & \text{swctoactuator} \end{cases}$$

$$\text{mindelay}(s) = \begin{cases} s.\text{response.of}f^*.\text{min} & \text{if } \text{segtype}(s) = \\ \quad -s.\text{lat}^*.\text{max} & \text{sensortobus} \\ s.\text{stimulus.of}f^*.\text{min} & \text{if } \text{segtype}(s) = \\ \quad +s.\text{lat}^*.\text{min} & \text{bustoactuator} \\ \text{next}(\text{next}(s)).\text{response.of}f^*.\text{min} & \text{if } \text{segtype}(s) = \\ \quad -\text{next}(s).\text{lat}^*.\text{max} - s.\text{lat}^*.\text{max} & \text{sensortoswc} \\ \text{prev}(\text{prev}(s)).\text{response.of}f^*.\text{min} & \text{if } \text{segtype}(s) = \\ \quad -\text{prev}(s).\text{lat}^*.\text{min} - s.\text{lat}^*.\text{min} & \text{swctoactuator} \end{cases}$$

According to Equation (6.18) every stimulus and response hop of each *transmission* segment of a function event chain, which has to fulfill a triggering constraint, has a triggering requirement. These requirements must be the same as the triggering constraint. Constraint (6.46) formalizes this for the generation of requirements over a set of triggering constraints TC .

$$\begin{aligned} \text{Requirements_Fulfill_Triggering_Constraints_TT}(TC) \equiv & \quad (6.46) \\ \forall t \in TC : \forall P \in \text{paths}(t.\text{target}) : \forall s \in P : \\ \text{segtype}(s) = \text{transmission} \Rightarrow s.\text{stimulus.trig}^*.\text{period} = & t.\text{period}. \end{aligned}$$

6.5.3 Consider Guarantees - The Windowing Approach

The new requirement values that are generated in each iteration must not only ensure the fulfillment of all timing constraints. Again they must also be related to the previous guarantees to not randomly generate the new values. Similar to the event-triggered case with its shifting approach, we developed an abstraction for the time-triggered case, which is used to explain the regeneration process more precisely using predicate logic over segments. We call our approach for time-triggered systems *windowing approach*.

Note that in the following we focus on the collaboration scenario *ECU Integration* to describe the windowing approach. It is thus used to negotiate time budgets between the system designer (bus configuration) and ECU developers.

Similar to Assumption 6.5, also for the windowing approach we make the following basic Assumption 6.7

Assumption 6.7 *We assume that unused window buffer is reported back to the system designer.*

Description of the Windowing Approach

When focusing on the collaboration scenario *ECU Integration*, every path of each function event chain consists of the four possible segment types *sensortobus*, *transmission*, *overecu*, and *bustoactuator*. A sequence of segments in a path follows a certain pattern, which is as follows:

- A *sensortobus* segment is always followed by a *transmission* segment and has no predecessor.
- A *transmission* segment is always followed by either an *overecu* or a *bustoactuator* segment. Its predecessor is always either an *overecu* or a *sensortobus* segment.
- An *overecu* segment only has *transmission* segments as possible successor and predecessor segments.
- A *bustoactuator* segment has no successor segment. Its predecessor is always a *transmission* segment.

All hops of a chain only have one of four possible types, which are *sensor*, *actuator*, *queued*, and *transmitted* hops. Based on these observations it turns out that every hop along a chain path has got an offset requirement except the chain's stimulus and response hops, because those are sensor and actuator hops. In other words, every inner hop of a chain (see Definition 5.8) separates two segments with an offset.

According to the windowing approach we understand every segment of a chain path as a window, which is assigned to the respective resource for their execution/transmission. Every window thereby has three parameters:

- *position* in the time line
- *size* from its start position to its end
- *period* of repetition

The position is absolute, because of the observations summarized above, i. e. the offsets of inner hops define an absolute position. Because time-triggered systems have a common absolute time base, the windowing approach is possible. Otherwise no positions could be assigned to windows. The end of a window results from its start position and size, and thus also marks an absolute position in the time line. The chain of the example TIMEX model of Figure 5.2 thus can be represented as five windows, as depicted in Figure 6.6. The period is assumed to be equal for all windows of a chain.

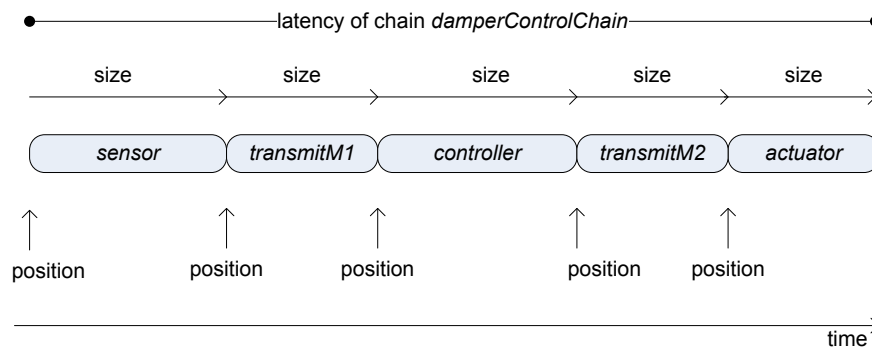


Fig. 6.6. Window representation of the TIMEX example.

The advantage of the windowing approach is that reasoning about requirement and guarantee values can be done solely based on the three attributes size, position and period for all windows. The approach abstracts from the concrete segment types that exist underneath the window and especially from their offset, latency and triggering requirement and guarantee *values*. Size, position and triggering are "coded" into the requirement and guarantee values. Both the demands to new requirement values of the output model as well as the effects of violated requirements can be described more abstract and unique for all segments by the windowing approach. A transformation from the window parameters to the segment and hop requirement and guarantee values and vice versa can easily be performed automatically. We did this in our prototypic implementation of the approach, which we describe in Chapter 7. For the description of the windowing approach we stick to the abstracted view.

The requirement values must be in a way such that the following two conditions hold:

1. There shall be no gaps between subsequent windows, because this is a waste of scheduling flexibility for the subsystem implementers. A gap is potential calculation/transmission time that neither of the two close-by windows can use.

2. There shall be no overlaps of subsequent windows, because this means that in the worst case a complete period is lost until the data is processed. This extends the overall chain latency by the time of one period.

The two unwanted situations are depicted in Figure 6.7.



Fig. 6.7. Unwanted situations of windows: a) gap b) overlap.

The new requirement values of each iteration must ensure two things: perfect window match and the fulfillment of the underlying function event chain's timing constraints. The example of Figure 6.6 fulfills these two demands, as indicated by the length of the row of windows and the according length of the latency constraint depicted above. Also the CLP constraints over the output model variables of our CLP approach already ensure these two conditions. The perfect window match of the generated requirement values is ensured by the several constraints of Section 6.5.1. The constraints presented there prevent the values from causing gaps and overlaps. The fulfillment of all timing constraints is ensured by the constraints presented in Section 6.5.2.

The size and position of each window is handed to the subsystem developers by means of offset and latency (and triggering) requirements. For all these requirements a response window size and position in terms of guarantees is given back to the system designer. By collecting all guarantees, the system designer can create a comprehensive picture of the entire window situation. The guarantee values again can lead to unwanted cases, which are the same as already discussed for the requirement values before. The guarantees must not produce gaps and overlaps, as depicted in Figure 6.7. They also must not cause non-fulfillment of timing constraints.

If all guarantees exactly fulfill their requirement, no unwanted guarantee situations occur and the system fulfills all timing constraints, because the requirements were already generated accordingly. Otherwise, if some guarantees over- or under-fulfill their requirement, then a new requirement generation is initiated to "repair" the window situations that cause gaps, overlaps, or non-fulfillment of timing constraints in general.

When focusing on one window, four non-fulfillment situations can occur between old requirement values and guarantee values, which we describe in the following. In general two terms are important here: backlog and buffer. The concepts can be used for both the beginning – called stimulus – and the end – called response – of a window. The four possible situations of a required window against its guaranteed window are visualized in Figure 6.8.

1. A *response backlog* occurs when the guaranteed end of the window exceeds the required end, i. e. the guaranteed end is after the required end.

2. A *response buffer* occurs when the guaranteed end of the window under-exceeds the required end, i.e. the guaranteed end is before the required end.
3. A *stimulus backlog* occurs when the guaranteed beginning of the window exceeds the required beginning, i.e. the guaranteed beginning is before the required beginning.
4. A *stimulus buffer* occurs when the guaranteed beginning of the window under-exceeds the required beginning, i.e. the guaranteed beginning is after the required beginning.

One stimulus situation can occur simultaneously with one response situation. For example, a guaranteed window can have a stimulus backlog *and* a response buffer. If either the window stimulus or the window response exactly fulfills its requirement, none of the above-mentioned situations for that particular window stimulus or window response exists.

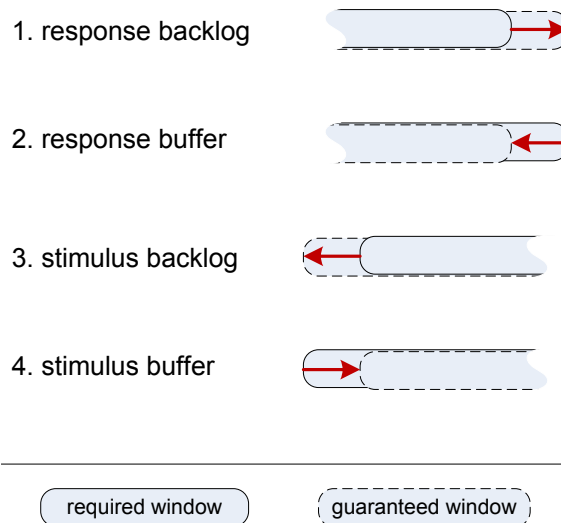


Fig. 6.8. Four possible situations how the old required window can relate to its guaranteed window.

Based on these four cases for one guaranteed window, the above-mentioned unwanted gap and overlap situations between two guaranteed windows can occur. A guarantee overlap for example can occur, when the left window has a response backlog and the right window has neither a stimulus buffer nor a stimulus backlog, but exactly fulfills its requirement window.

If any window does not fulfill its required size and position and thus can cause either a gap or an overlap, a next iteration of the processing step *generate new requirements* is initiated to repair the situation. Similar to the description of the relation between old requirements and guarantees for a window presented above, also the guarantees relate to the new requirements for a window accordingly. However we use a different terminology to clearly distinguish the

two relations. The following four situations can occur with respect to the guarantees when new window requirements are generated.

1. The *response is relaxed* if the new requirement shifts the end of the window "rightward", i.e. the new required end is after the guaranteed end.
2. The *response is tightened* if the new requirement shifts the end of the window "leftward", i.e. the new required end is before the guaranteed end.
3. The *stimulus is relaxed* if the new requirement shifts the beginning of the window "leftward", i.e. the new required beginning is before the guaranteed beginning.
4. The *stimulus is tightened* if the new requirement shifts the beginning of the window "rightward", i.e. the new required beginning is after the guaranteed beginning.

The four situations between guaranteed window and new required window are visualized in Figure 6.9.

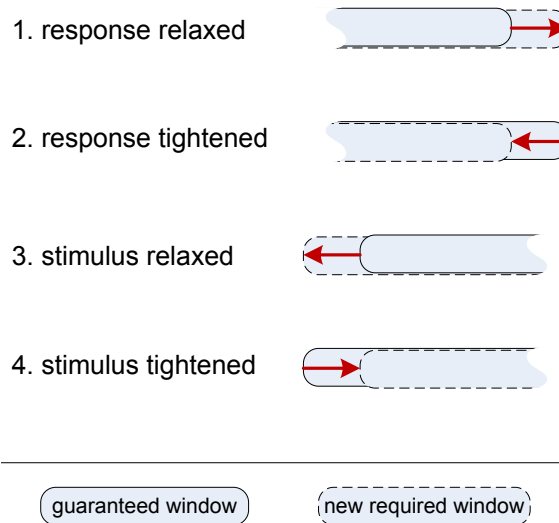


Fig. 6.9. Four possible situations how the guaranteed window can relate to its new required window.

So far we neglected one specific detail of time-triggered systems in our approach. All actions, such as transmission of data and execution of software, are performed periodically. Therefore each window has another property besides position and size, namely its *window period*. We assume that all windows of a segment path have the same period. This is ensured by Constraint (6.43).

As in a time-triggered system all actions repeat with a certain period, the relaxing and tightening actions described above are subject to several conditions. So far we assumed that time is continuously advancing. In fact, from the viewpoint of a single window, time is only advancing until the window

period P . So time is relative to the period start time of a window. When a new instance of a window's period starts we call that point in time a *period switch*.

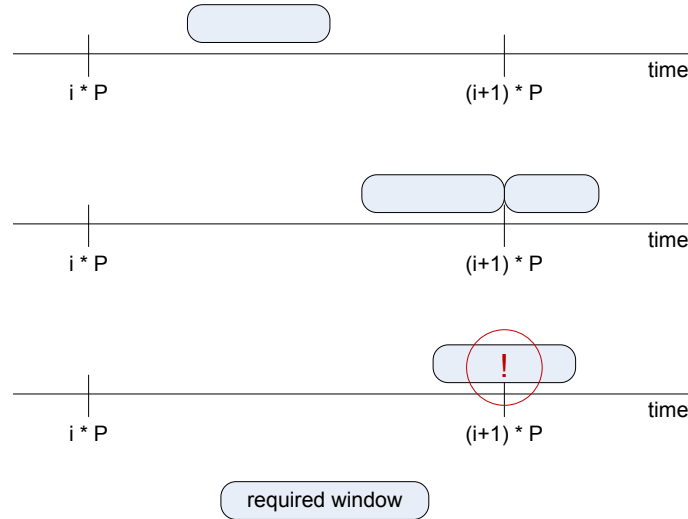


Fig. 6.10. All window requirements must be in a way that a period switch does not occur during a window.

With respect to the period, we can formulate additional demands for required windows. To simplify matters all requirement values must be in a way that every window does not overlap its period switch. Because of the other condition that no gaps are allowed for requirement values, the period switch of several windows that belong to one segment path can only be exactly between two successive windows. Basically this condition stems from Assumption 2.3. It is already incorporated into the TIMEX model by Predicate (6.41). Figure 6.10 visualizes the described situations for a single window at an allowed position, for two successive windows with allowed period switch, and for a single window with not allowed period switch. Note that this condition is not required for windows of type *externaltobus* due to Predicate (6.40). The size of such windows is always greater than one period and thus inevitably overlaps a period switch.

CLP Implementation of the Windowing Approach

A variable instantiation within the whole solution space of the output model must fulfill all basic windowing conditions described above (no gaps and overlaps, period switch) and must repair potential problems with the guarantees. Therefore we now describe our CLP solution of the windowing approach with predicate logic.

First we set up some basic predicates over old and new requirements and guarantees (see Figure 6.2) based on the three window attributes position, size and period.

We use these basic predicates to formulate composite predicates. All predicates are then used to formulate the constraints that represent the CLP realization of the windowing approach. We use the suffix $_r$ for the old requirement, $_g$ for the guarantee and $*$ for the new requirement of the according window parameter position, size or period in our constraint formalization.

Relating Old Requirements and Guarantees:

First we define the following predicates for the four situations depicted in Figure 6.8. Let $s \in S$ be a segment with appropriate offset and/or latency requirements. The following predicates are true if the according situation occurs for the window guarantees and the old requirement.

$$Response_Backlog(s) \equiv position_r + size_r < position_g + size_g. \quad (6.47)$$

$$Response_Buffer(s) \equiv position_r + size_r > position_g + size_g.$$

$$Stimulus_Backlog(s) \equiv position_r > position_g.$$

$$Stimulus_Buffer(s) \equiv position_r < position_g.$$

Relating Guarantees and New Requirements:

The following predicates represent the four situations depicted in Figure 6.9 for a segment $s \in S$ and relate the guaranteed window to the new required window.

$$Response_Relaxed(s) \equiv position_g + size_g < position^* + size^*. \quad (6.48)$$

$$Response_Tightened(s) \equiv position_g + size_g > position^* + size^*.$$

$$Stimulus_Relaxed(s) \equiv position_g > position^*.$$

$$Stimulus_Tightened(s) \equiv position_g < position^*.$$

Relating Old Requirements and New Requirements:

The new requirement values can also be related to the old requirement values without taking the guarantees into account. In this case we say the requirements are *repeated* for a window, if the new requirement is the same as the old requirement. The window stimulus and response requirements can be repeated. Again, the actual stimulus and response positions are a result of the underlying requirement values.

$$Repeat_Stimulus(s) \equiv position_r = position^*. \quad (6.49)$$

$$Repeat_Response(s) \equiv position_r + size_r = position^* + size^*.$$

Start and End Positions of a Window:

As depicted in Figure 6.10 a window must not overlap its own period switch. It can only end right before or start right after the period switch. Given a segment s that represents a window with its three attributes *position*, *size* and *period*. The following predicate is true if a window is at the end position or start position, respectively.

$$End_Position(s) \equiv position + size = period. \quad (6.50)$$

$$Start_Position(s) \equiv position = 0.$$

Freeing the Resource for a Window:

Each segment has a resource attribute that indicates to which resource (processor or bus) the segment belongs. Thus we also can assign windows to resources. The position and size attributes of a window mark a time span in which the segment may utilize the resource (not necessarily the entire time span). This means either task execution or frame transmission by the resource on the technical architecture. It is more likely that a requirement of a window can be fulfilled, if as less as possible other windows occupy the resource at the same time. Therefore we are interested in knowing whether a resource is freed by the new requirement values compared to the old ones. Predicate (6.51) is true, if during the time span of window s less windows from other chains occupy the resource (i. e. have overlapping time spans) according to the new requirements than according to the old requirements.

$$Resource_Freed(s) \equiv \text{less windows overlap } s. \quad (6.51)$$

Note again that the windowing approach abstracts from the underlying actual requirements and guarantees, i. e. latency, offset and triggering values. The mapping from the predicates over windows and their movements to the real requirements and guarantees of the segment and hops depends on the segment type. Therefore we stick to the pure window-based CLP constraints for the sake of better understandability of the CLP implementation.

Constructing Composite Predicates:

Based on the basic Predicates 6.48 that relate guarantees and new requirements, another composite predicate can be formulated. Given a segment $s \in S$, $Keep_Guarantee_Window(s)$ is true, if the new requirement values just take over the guaranteed values, i. e. they do not change the window with respect to its guarantee at all.

$$\begin{aligned} Keep_Guarantee_Window(s) \equiv & \neg Response_Relaxed(s) \wedge \quad (6.52) \\ & \neg Response_Tightened(s) \wedge \\ & \neg Stimulus_Relaxed(s) \wedge \\ & \neg Stimulus_Tightened(s). \end{aligned}$$

The following composite predicates can be formulated based on the stimulus and response repetition basic Predicates 6.49.

$$\begin{aligned} Repeat_Stimulus_Only(s) \equiv & Repeat_Stimulus(s) \wedge \quad (6.53) \\ & \neg Repeat_Response(s). \\ Repeat_Response_Only(s) \equiv & Repeat_Response(s) \wedge \\ & \neg Repeat_Stimulus(s). \\ Repeat_Complete(s) \equiv & Repeat_Stimulus(s) \wedge \\ & Repeat_Response(s). \end{aligned}$$

Consider Predicates 6.50 and functions $next(s)$ and $prev(s)$. The following predicates are true, if there is a period switch between two windows in a path of segments $prev(s), s, next(s) \in S$.

$$\begin{aligned}
Period_Switch_Right(s) &\equiv End_Position(s) \wedge & (6.54) \\
&Start_Position(next(s)). \\
Period_Switch_Left(s) &\equiv Start_Position(s) \wedge \\
&End_Position(prev(s)).
\end{aligned}$$

A window's stimulus or response can be moved to the left or right, respectively, by the new requirements. This can be either with respect to the guarantee or with respect to the old requirement. In both cases we call such a position change *window shifting*. Based on the already presented predicates the shifting of a window can be expressed as follows.

$$\begin{aligned}
Shift_Right(s) &\equiv Stimulus_Buffer(s) \vee & (6.55) \\
&Stimulus_Tightened(s) \vee \\
&Period_Switch_Right(s). \\
Shift_Left(s) &\equiv Response_Buffer(s) \vee \\
&Response_Tightened(s) \vee \\
&Period_Switch_Left(s).
\end{aligned}$$

Using All Window Predicates to Formulate Constraints:

Let $next, prev \in S$ be the next and previous segment of segment s in a segment path. Given segment s both can be determined by the functions $next(s)$ and $prev(s)$.

Based on all these predicates over single segments, which are understood as windows, our windowing approach to generate new requirements based on the guarantees and old requirements of the previous iteration is formulated by the following predicate. It consists of the Constraints 6.56 to 6.65. We explain the meaning and consequences of each constraint thereafter.

$Windowing(S) \equiv \forall s \in S :$

$$\begin{aligned} Response_Backlog(s) \Rightarrow & Keep_Guarantee_Window(s) \vee (6.56) \\ & Repeat_Response_Only(s) \vee \\ & Repeat_Complete(s) \end{aligned}$$

\wedge

$$\begin{aligned} Stimulus_Backlog(s) \Rightarrow & Keep_Guarantee_Window(s) \vee (6.57) \\ & Repeat_Stimulus_Only(s) \vee \\ & Repeat_Complete(s) \end{aligned}$$

\wedge

$$\begin{aligned} Response_Backlog(s) \wedge & Keep_Guarantee_Window(s) \Rightarrow (6.58) \\ & Shift_Right(next) \vee \\ & Shift_Right(prev) \end{aligned}$$

\wedge

$$\begin{aligned} Response_Backlog(s) \wedge & Repeat_Response_Only(s) \Rightarrow (6.59) \\ & Stimulus_Relaxed(s) \wedge \\ & Shift_Left(prev) \end{aligned}$$

\wedge

$$\begin{aligned} Response_Backlog(s) \wedge & Repeat_Complete(s) \Rightarrow (6.60) \\ & Resource_Freed(s) \end{aligned}$$

\wedge

$$\begin{aligned} Stimulus_Backlog(s) \wedge & Keep_Guarantee_Window(s) \Rightarrow (6.61) \\ & Shift_Left(prev) \vee \\ & Shift_Left(next) \end{aligned}$$

\wedge

$$\begin{aligned} Stimulus_Backlog(s) \wedge & Repeat_Stimulus_Only(s) \Rightarrow (6.62) \\ & Response_Relaxed(s) \wedge \\ & Shift_Right(next) \end{aligned}$$

\wedge

$$\begin{aligned} Stimulus_Backlog(s) \wedge & Repeat_Complete(s) \Rightarrow (6.63) \\ & Resource_Freed(s) \end{aligned}$$

\wedge

$$\begin{aligned} Response_Tightened(s) \Rightarrow & Stimulus_Relaxed(s) \vee (6.64) \\ & Stimulus_Buffer(s) \vee \\ & Resource_Freed(s) \end{aligned}$$

\wedge

$$\begin{aligned} Stimulus_Tightened(s) \Rightarrow & Response_Relaxed(s) \vee (6.65) \\ & Response_Buffer(s) \vee \\ & Resource_Freed(s). \end{aligned}$$

There are two basic constraints that control the entire re-generation process, because there also are two situations that must be repaired by the algorithm.

First, Constraint (6.56) expresses three mutually exclusive reaction possibilities to a response backlog. If such a backlog occurs then either the guaranteed window can be kept by the new requirements, or the response requirement is repeated despite the backlog, or the stimulus and response requirement is repeated for that window.

Second, Constraint (6.57) expresses three mutually exclusive reaction possibilities to a stimulus backlog. Similar to the response backlog described above, the guaranteed window can be kept, i. e. granted to the segment, the stimulus requirement can be repeated despite the backlog, or the window requirement is repeated entirely.

Each of the three *or*-linked reaction possibilities for a response backlog is bound to additional constraints.

If the guarantee window is kept (Constraint (6.58)) there is a potential overlap at the window response because of the response backlog. To fix this overlap the next window must be shifted to the right (recursively until a buffer is found somewhere on the path). If there is no next window the previous window must be shifted to the right (recursively), because otherwise the overall path latency would be exceeded.

If a response backlog exists but the response requirement of the window is repeated (although it is known that there is a problem), then the stimulus of the window must be relaxed and the previous window must be shifted to the left. The according Constraint (6.59) formalizes this.

Finally, if a response backlog exists but the required window is repeated completely despite that, then the resource must be freed at this place in the schedule (Constraint (6.60)). The freeing of the schedule reduces the processor utilization and increases the probability of requirement fulfillment.

Similar, each of the three *or*-linked reaction possibilities for a stimulus backlog is bound to additional constraints.

If the guarantee window is kept (Constraint (6.61)) there is a potential overlap at the window stimulus because of the stimulus backlog. So the previous window must be shifted to the left. If there is no previous window the next window must be shifted to the left (recursively), because otherwise the overall path latency would be exceeded.

If a stimulus backlog exists but the stimulus requirement of the window is repeated despite that (Constraint (6.62)), then the response of the window must be relaxed and the next window must be shifted to the right.

Finally, if a stimulus backlog exists but the required window is repeated completely despite that, then again the resource must be freed at this place in the schedule (Constraint (6.63)).

Some of the implications defined above are only possible, if the stimulus or response of the respective next or previous windows is tightened. To finalize our windowing approach CLP realization, additional constraints are necessary for stimulus and response tightening.

If one end of a window is tightened, then one of three possible conditions must hold to compensate the stricter requirement: either the other window end must be relaxed, or the resource must be freed, or the window must have a buffer at the other end. Constraint (6.64) and Constraint (6.65) formalize this implications.

6.5.4 Search

We connect all constraints in a search predicate for time-triggered systems to find a solution in the output model that fulfills all constraints of the windowing approach. Given a set of segments S , a set of hops H , a set of chains F and all function triggered timing constraints LC , SC and TC , which have to be fulfilled if all windows fulfill their requirements. Predicate (6.66) is the search predicate for time-triggered systems.

$$\begin{aligned}
 \text{Search_TT}(S, H, F, LC, SC, TC) \equiv & \quad (6.66) \\
 & \text{Output_Model_Latencies}(S) \wedge \\
 & \text{Output_Model_Triggerings}(H) \wedge \\
 & \text{Output_Model_Offsets}(H) \wedge \\
 & \text{Invariant_Latencies}(S) \wedge \\
 & \text{Invariant_Triggerings}(H) \wedge \\
 & \text{Invariant_Offsets}(H) \wedge \\
 & \text{General_TT_Externaltobus}(S) \wedge \\
 & \text{General_TT_Transmission}(S) \wedge \\
 & \text{General_TT_Overecu}(S) \wedge \\
 & \text{General_TT_Triggerings}(F) \wedge \\
 & \text{Requirements_Fulfill_Latency_Constraints_TT}(LC) \wedge \\
 & \text{Requirements_Fulfill_Synch_Constraints_TT}(SC) \wedge \\
 & \text{Requirements_Fulfill_Triggering_Constraints_TT}(TC) \wedge \\
 & \text{Windowing}(S).
 \end{aligned}$$

6.6 Generating a Communication Model

After a successful run of our requirement generation algorithm presented in the previous three sections the necessary set of system timing requirements is available. Some of these requirements, namely the ones that affect the *transmission* segments, have to be fulfilled by the system designer. In Section 3.1.1 we defined that this role is responsible for the network design. For that reason we explicitly point out the generation of the communication model as an own step in our methodology.

The communication model generation again depends on the system type. According to Section 3.3.5 the communication model offers different attributes for frames in both types. However there are some observations that apply for both types:

- Multiple signals are mapped to one frame. The timing properties of a frame determine its timing behavior (e. g. worst case response time, transmission offset). All contained signals literally share the same timing behavior and thus automatically have the same timing guarantee. As a consequence, it makes sense to group those signals to the same frame, which also have the same (or at least "compatible") timing requirements, because otherwise not all of them can be fulfilled at once.
- A frame can only be sent by one ECU. Thus only signals that are sent by the same ECU can be mapped to the same frame.
- In most systems, the size of a frame is limited. Especially in time-triggered systems a static slot size is configured, where each slot holds one frame. If the signal size (or length) is assumed equal, a static number of signals fits in one frame.

Evaluation

In Section 4.7 we summarize three basic limitations of related work in the area of timing modeling and timing analysis. We developed the TIMEX model, its methodology and especially the shifting and windowing approach presented in this thesis to overcome these limitations. In this chapter we demonstrate and discuss how the limitations are overcome by our work and how the TIMEX model and methodology are applied in practice.

7.1 Standardization of the TIMEX Model

Most timing models used today consist of system and schedule attributes on the lowest level of schedulable entities like task and frame properties. They do not support appropriate abstraction techniques that enable more abstract modeling and analysis methods. We discussed this limitation of timing models of related work in Section 4.7.2.

We discussed several reasons for a new more abstract timing model in Section 5.1. The TIMEX timing model solves the limitation mentioned above and allows for a) a more abstract modeling of system timing, b) the specification of timing constraints on a functional level, and c) timing analysis on the abstracted level of requirements and guarantees for segments and hops. The goal of the TIMEX model is to support timing modeling and timing analysis in a distributed development process. The other timing models are not suitable for that purpose, because they reveal too much information about the system, what often shall not be shared between several development teams and organizations.

The basic concept of the TIMEX model is our concept of observable events and event chains (Section 3.2). During the work on this thesis, the basic concept of events and event chains has also been discussed in an expert group within the AUTOSAR development partnership. The concept finally has been adopted to the AUTOSAR model and published as timing model for AUTOSAR Release 4.0. The model thus is now part of the automotive industry's specification standard for embedded hardware and software architecture. Figure 7.1 shows the according AUTOSAR meta-model extract for the event and event chain concept.

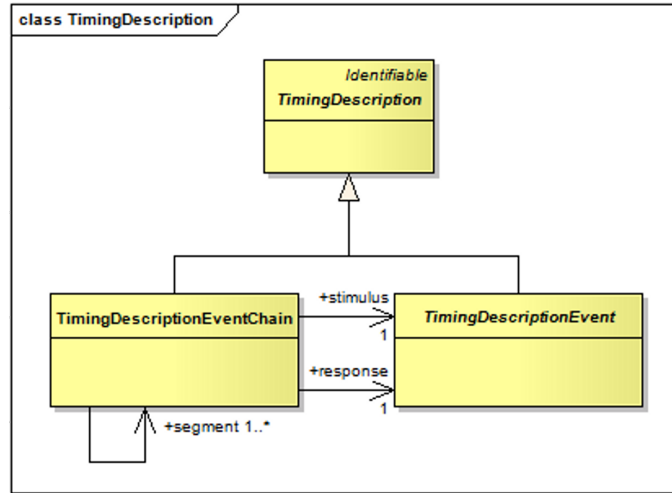


Fig. 7.1. AUTOSAR timing model: events and event chains [4].

Another basic concept of TIMEX is the concept of requirements and guarantees. They represent two different roles, or semantics, of timing constraints in a TIMEX system timing. This concept has also been adopted by the AUTOSAR timing model. An AUTOSAR timing constraint has one of the two mentioned roles. The idea of distributed development model support provided by TIMEX thus found its way to practice by the standardization in the AUTOSAR model. However, TIMEX and AUTOSAR timing constraints have some differences, which we list later in this section. Figure 7.2 depicts the AUTOSAR meta-model extract for the requirement and guarantee roles of a timing constraint.

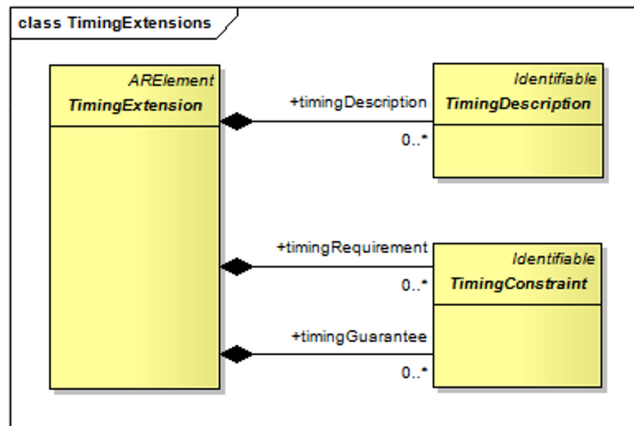


Fig. 7.2. AUTOSAR timing model: timing requirements and timing guarantees [4].

With the increasing acceptance and usage of the AUTOSAR standard, the need for a common timing model arose recently. Moreover AUTOSAR offers an ideal modeling basis for such a timing model, because it consists of a lot of common model elements and terminology used throughout the industry already. For that reason the event and event chain concept was incorporated into the

AUTOSAR model. Before that only company- or tool-specific timing models existed, which complicated the exchange of data between organizations. Now the AUTOSAR timing model is being rolled out in the companies and increasingly used to model and exchange timing specifications, for example between an OEM and its suppliers. To ease the usage of the timing model, a specification tool has been developed based on the textual modeling editor prototype of this thesis, which we show in Appendix A. The specification tool has already been published as part of the AUTOSAR tool platform Artop [48, 77].

As already described, the AUTOSAR timing model and TIMEX share a) the concept of observable events and event chains and b) the two different semantics of timing constraints, namely requirements and guarantees. Above that similarities however there are certain differences in how the two models utilize these concepts. The three most important differences are:

1. The main difference of the two models is that the AUTOSAR timing model is more generic, whereas TIMEX is rather problem specific. The AUTOSAR timing model offers only one type of event chain, which is the general causal connection of two events. This generic event chain is used for every stimulus and response relationship of two events. There are no rules that constrain the usage of certain event types in the stimulus or response roles. Further the AUTOSAR timing model does not restrict or regulate the refinement of event chains with sub-chains. This means that according to the standard basically every timing-based relation can be specified and arbitrarily refined. The TIMEX model however offers two specific event chains, namely a *function event chain* and a *segment*. Segments are the only possible refinement level for event chains. Further TIMEX offers certain types of segments that are identified by their types of stimulus and response events (called hops). The reason for this is that the TIMEX model is designed for the use in a distributed development process and to overcome the limitation of related work that standard timing models are not very well suitable for that usage. Therefore the concept of well-defined segment types as a refinement for function event chains based on subsystem borders fits for such processes. Further TIMEX uses the concept of hops, which are used to characterize observable events as such event that can be observed at the interface of a system or at the border of two subsystems (see Definition 5.2).
2. The AUTOSAR timing model and the TIMEX model (more precisely the system model underneath the TIMEX model) are based on a set of possible event types. The granularity and completeness of the system model basically determines, which event types can be observed in such a system. AUTOSAR has a very large and comprehensive system model and thus offers more different types of observable events than TIMEX. However, our system model is appropriate to demonstrate the usage of the TIMEX model and methodology to improve distributed development of automotive systems.
3. Both timing models basically have the same timing constraint types, which are latency, triggering and synchronization constraints. In the AUTOSAR model, these constraints can be used for every available event or event chain, respectively. The target elements for timing constraints is not lim-

ited or restricted. Similar to the definition and refinement of event chains, also with respect to timing constraints the AUTOSAR timing model is very generic. TIMEX however only has one semantics of a timing constraint, namely the usage of a timing constraint as a function-triggered and implementation independent timing constraint in a so-called TIMEX function timing. Such a function-timing is not possible in AUTOSAR at all, because AUTOSAR does not contain the concept of functions. AUTOSAR software components are already implementation-specific and AUTOSAR does not have a mapping from functions to software components. However, as we discuss in Section 3.5, timing constraints basically should not be specified for a certain system implementation in terms of hardware and software in the first place. Rather they should be defined on the implementation-independent level of functions.

Concluding, the AUTOSAR timing model and the TIMEX model are structurally equal. TIMEX has a problem-specific focus for managing time budgets in distributed development. The AUTOSAR timing model is very generic and comprehensive. TIMEX models can be transformed to AUTOSAR timing models with appropriate generators and vice versa.

We implemented such a generator in a tool prototype for this thesis. The prototype also contains the before mentioned textual modeling editor that helps engineers to create TIMEX models, the CLP implementation and a visualization of TIMEX. The prototype is outlined in Appendix A.

7.2 Application of the TIMEX Methodology

We discuss the application of our TIMEX methodology based on the running example of this thesis. The example is depicted in Figure 5.2. In particular, we focus on methodology step 5, i. e. the iterative generation of new requirement values according to our constraint logic programming approach presented in Chapter 6. Thereby, we first assume an event-triggered realization and then a time-triggered realization of the system model of the running example. Given a set of old requirement values and according guarantee values of an iteration n we show how both the shifting approach (Section 6.4.2) and the windowing approach (Section 6.5.3) work to compute appropriate new requirement values for iteration $n+1$. The new values are the solution to the CLP problem, which we formalized using predicate logic.

7.2.1 Discussion of the Shifting Approach

For the discussion of our shifting approach we assume an event-triggered system that implements the system timing model of the example in Figure 5.2.

Analysis of Horizontal Shifting:

First we demonstrate horizontal shifting. For simplicity reasons we therefore assume that the whole system only consists of the one function chain of the running example. Figure 7.3 depicts the situation that we analyze.

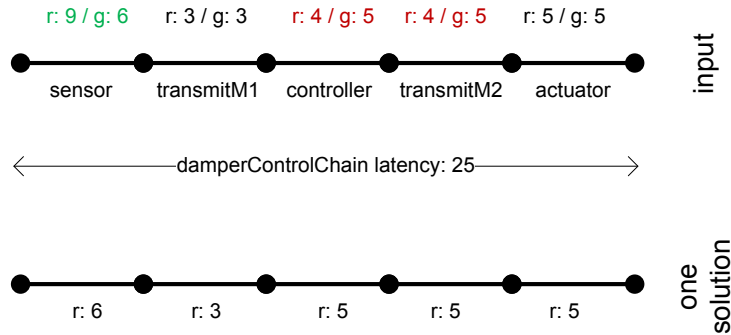


Fig. 7.3. Example with one solution by horizontal shifting.

The input to the CLP system are the old requirements r and the according guarantees g of iteration n , which are displayed above the segments. The segment named *sensor* has an old requirement of 9 and got a guarantee of 6. The requirement therefore is over-fulfilled and the segment has got a buffer. The buffer is indicated by the green color. The segments called *transmitM1* and *actuator* both fulfill their requirement exactly. The segments called *controller* and *transmitM2* both do not fulfill their requirement. They have got a backlog. The backlog is indicated by the red color.

The sum of the old requirement values is 25, what is exactly the latency constraint of the overall function chain. The sum of the guarantee values is 24. That would also fulfill the latency constraint of a maximum of 25. However, there are two local timing conflicts at segments *controller* and *transmitM2*. The latency constraint is considered as not fulfilled, until all local timing conflicts are solved. Such situations can be handled with horizontal shifting.

The search for new requirement values is triggered by Predicate (6.39). The predicate applies the rules for the horizontal shifting approach by *Shifting(S)*.

The application of all rules of the shifting approach *Shifting(S)* over the segments S leads to one solution for the CLP problem here. The solution is shown in the lower part of Figure 7.3 as the new requirements r below all segments. First, the two backlogs are repaired according to Constraint (6.38), which leads to taking over the guarantee values. This is possible, because also the guarantee value of the segment *sensor*, which has got a buffer, is taken over as new requirement. So the new overall latency of the function chain is 24 and thus fulfills the latency constraint of 25. All five segments must neither be tightened nor relaxed, because their guarantees are taken as new requirements. Therefore Constraint (6.36) and Constraint (6.37) are not applied in this example.

Analysis of Vertical and Diagonal Shifting:

For a demonstration of vertical and diagonal shifting we must extend the running example. We need an additional function chain, whose segments share resources with the original function chain, because vertical and diagonal shifting are based on the redistribution of resource utilization.

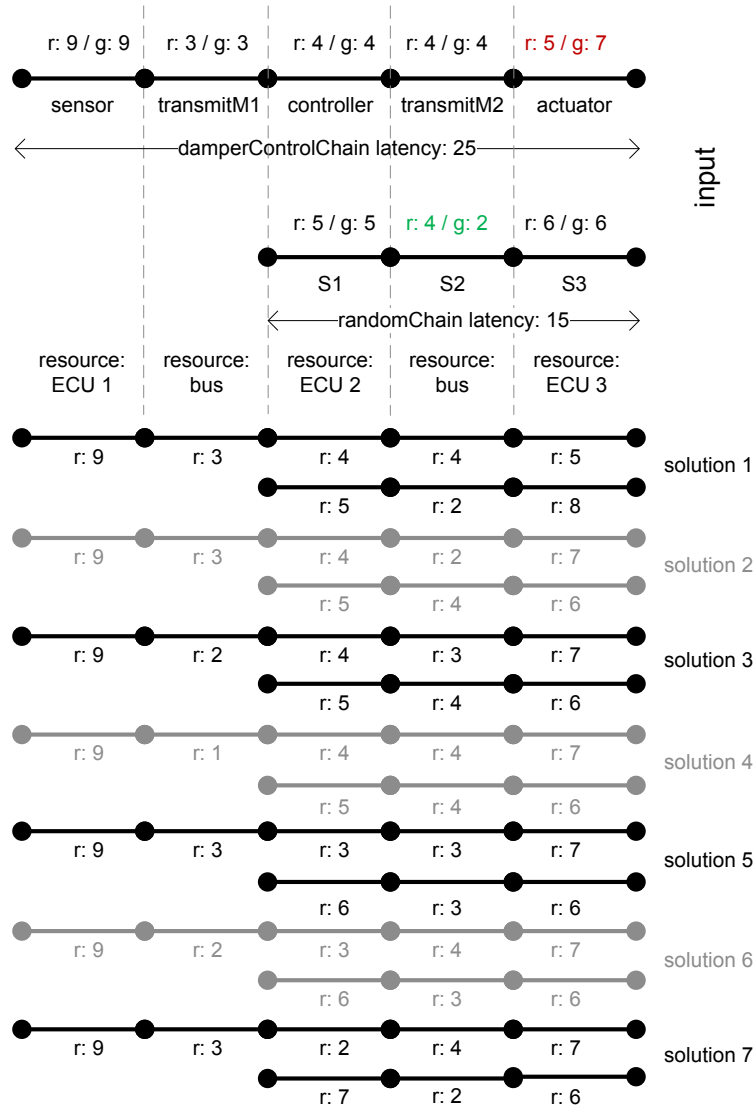


Fig. 7.4. Example with seven solutions by diagonal shifting.

As shown in Figure 7.4, we add a function chain called *randomChain* to the example. The new function chain consists of three segments:

- The first segment *S1* is of type *externaltobus* and it is located on the same ECU resource as the *controller* segment, which is of type *overecu*.
- The second segment *S2* is a *transmission* segment and belongs to the same bus resource as segments *transmitM1* and *transmitM2* (there is only one bus in the example).
- The third segment *S3* is of type *bustoexternal* and it is located on the same ECU resource as segment *actuator*.

The segment groups *controller* plus *S1*, *transmitM1* plus *transmitM2* plus *S2*, as well as *actuator* plus *S3* each compete for their common resource. *transmitM1*, *transmitM2* and *S2* are transmission segments and thus actually represent a frame transmission on the bus. The four segments of the other two segment groups represent competing task executions on their common processor resource.

The sum of the old requirement values of the *damperControlChain* is 25, i. e. it is exactly the latency constraint of that function chain. The sum of the guarantee values of the *damperControlChain* is 27. So according to the guarantees the latency constraint of a maximum of 25 is not fulfilled. There is a local timing conflict at segment *actuator*, because there the guarantee does not fulfill its requirement by 2. On the other hand, the guarantee sum of the *randomChain* is 2 smaller than the old requirement sum. Segment *S2* has got a buffer of 2. Such local timing conflicts at segment *actuator* can be solved with vertical and diagonal shifting by utilizing the buffer of segment *S2* for a fulfillment of the other function chain's latency constraint.

The application of all rules of the shifting approach *Shifting(S)* over the segments *S* of both chains leads to seven solutions for the CLP problem. The solutions are shown in the lower part of Figure 7.4 as the new requirements *r* below all segments. We discuss the solutions in the following.

The search for new requirement values is triggered by Predicate (6.39). The predicate applies all CLP constraints for possible solutions, such as standard output model constraints, invariants, the fulfillment of the timing constraints by the new requirement values, and finally the rules for the shifting approach by *Shifting(S)*.

According to Constraint (6.38) a not fulfilled segment latency requirement either results in taking over the guarantee or in setting the same requirement as before, which is currently not fulfilled. The latter case has been chosen in solution 1, i. e. segment *actuator* again got a new requirement of 5. This choice however means that the requirement of segment *actuator* is tightened with respect to the guarantee, which is 7. According to Constraint (6.36) such a tightening implies a) that the chain path has a backlog (which is true for the *damperControlChain*) and b) that there must be a relaxed segment on the same resource. There is only one other segment on the same resource ECU 3 in this case, which is segment *S3*. The new requirement of *S3* in solution 1 therefore is 8, which is relaxed compared to the guarantee of 6. According to Constraint (6.37) the relaxing of *S3* implies a) that there is a path buffer (which is true for the *randomChain*) and b) that there is a tightened segment on the same resource, which is segment *actuator*, as already discussed. All other segments of the two chains just take over their guarantee as the new requirement and neither Constraint (6.36) nor Constraint (6.37) must be applied for them. In particular segment *S2* thereby releases its buffer to relax segment *S3* by horizontal shifting. The buffer is then used to solve the timing problem vertically on the resource ECU 3 as described above. In combination, the timing problem is solved by diagonal shifting. Solution 1 is the only possible solution for the case that the new requirement of segment *actuator* is repeated.

Solutions 2 to 7 cover the cases, where according to Constraint (6.38) the new requirement of segment *actuator* is 7, i.e. its guarantee is just taken over as new requirement. All these solutions have in common that another segment of the function chain *damperControlChain* must be tightened by 2, or 2 segments must be tightened by 1 each, because otherwise the latency constraint of *damperControlChain* would not be fulfilled.

Therefore solution 2 is the most obvious one. The buffer of segment *S2* is vertically shifted to segment *transmitM2* according to Constraint (6.37) and Constraint (6.36). Practically this means that the priorities of the two frames that are represented by the two segments must be adapted and thus the bus schedule must be redesigned to fulfill these new requirements.

In solutions 3 and 4 segment *S2* also provides its entire buffer to other segments on the bus resource by vertical shifting. In solution 3 the buffer is divided for segment *transmitM1* and *transmitM2*. both are tightened by 1, which is possible because segment *S2* on the same bus resource is relaxed (Constraint (6.36) and Constraint (6.37)). In solution 4 the buffer is used solely to tighten segment *transmitM1* by 2.

Solutions 5, 6 and 7 utilize diagonal shifting. In solutions 5 and 6 a part of the buffer of segment *S2* is shifted horizontally to segment *S1*. From there it is shifted vertically to segment *controller* which is then tightened by 1. The other part of the buffer is still shifted vertically to the segment *transmitM2* (solution 5) and *transmitM1* (solution 6), respectively. In solution 7 the buffer of segment *S2* is completely shifted diagonally to segment *controller* over segment *S1*. Therefore the new requirement of segment *controller* can be tightened by 2 and the overall latency constraints of both function chain *damperControlChain* and function chain *randomChain* are fulfilled, according to all new requirement values.

The application of the shifting approach according to our CLP implementation possibly offers several solutions for a new requirement value assignment. Thus there still exists a certain freedom to choose one of the proposed solutions. A typical approach in constraint programming is to evaluate solutions according to appropriate cost functions. The solution with the lowest cost can be selected as final output. We outline possible cost evaluations for the shifting approach in Section 8.2.

7.2.2 Discussion of the Windowing Approach

For the discussion of our windowing approach we assume a time-triggered system that implements the system timing model of the running example depicted in Figure 5.2.

We assume that the system only consists of the one function chain of the example. The five segments now are understood as time windows according to the windowing approach described in Section 6.5.3. Figure 7.5 displays the five windows and their size and positions in time. The windows are within the function chain's given period $P = 10$, which we also consider as invariant here. That means all triggering variables of the CLP problem have a predefined value of 10 to simplify the demonstration of the windowing approach.

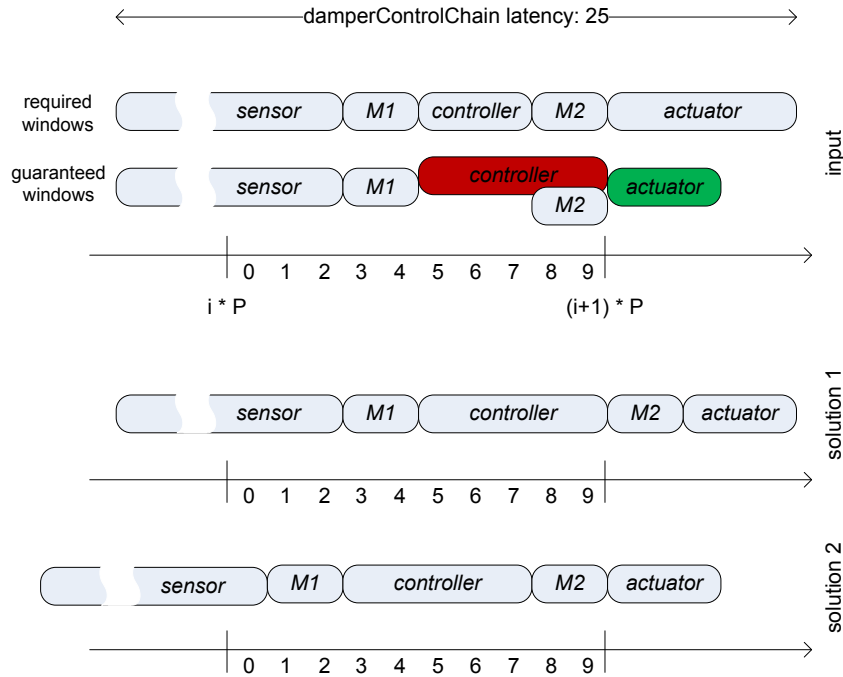


Fig. 7.5. Example with two solutions by the windowing approach.

The input to the CLP system here are all old latency and offset requirements and their according guarantees of iteration n . Additionally to the visualization as windows we also provide a detailed table of all offset and latency values of the example’s segments and hops in the following. As mentioned in Section 6.5.3 window positions and sizes actually represent these low-level TIMEX model details. Table 7.1 holds these low-level requirement and guarantee values for the input of the windowing example depicted in Figure 7.5.

element	type	old requirement		guarantee	
		min	max	min	max
sensor	latency	1	13	1	13
controller	latency	1	3	1	5
actuator	latency	1	5	1	3
M1 queued	offset	2	2	2	2
M1 transmitted	offset	3	4	3	4
M2 queued	offset	5	7	5	9
M2 transmitted	offset	8	9	8	9

Table 7.1. The input values of the windowing approach example.

The input value situation in the example is as follows. The old requirement values are in a way that the overall latency of the *damperControlChain*, i. e. the length of all five windows, is exactly 25. There are neither gaps nor overlaps between windows and the period switch occurs between two windows, namely *M2* and *actuator*. We defined rules in terms of predicates and constraints in Section 6.5.3 to formalize these basic window conditions.

The current guarantees of all windows in the example cause a local timing problem. The controller window has a response backlog of 2. That means the window size has increased by 2 at the right end. The response backlog is also represented by the low-level latency and offset requirement values given in Table 7.1. As shown in the table, the *controller* latency guarantee is 5 instead of 3 and therefore also the *M2 queued* hop max offset is 9 instead of 7. This response backlog leads to an overlap of the two windows *controller* and *M2*. The bus data *M2*, which is queued for transmission at time point 9 is transmitted at time point 8 at the earliest (min) and 9 in the worst case (max). Therefore a whole period of 10 is lost and must be added to the overall function chain latency. The latency constraint of 25 is thus not fulfilled, because according to the current window guarantees the overall latency can be calculated as follows:

$$\underbrace{25}_{\text{original latency}} + \underbrace{10}_{\text{period loss}} - \underbrace{2}_{\text{buffer of window } \textit{actuator}} = 33$$

The *actuator* window in the example has a response buffer of 2. The buffer is indicated by the green color in Figure 7.5. The *actuator* latency guarantee has decreased to 3, compared to the old requirement of 5. All other windows exactly fulfill their requirements with their guarantees. Using our windowing approach, the buffer of the actuator window can be used to solve the local timing problem of the controller window.

As we will demonstrate in the following, two solutions are possible in the windowing approach CLP system initiated by the *Search_TT* Predicate (6.66). The predicate applies all rules of the windowing approach as CLP constraints. The rules cover all general conditions for offset, period and latency requirements, the fulfillment of all timing constraints and some more (see Section 6.5.3). In the following we focus on the functionality of the predicate *Windowing(S)* over the set of all segments, which implements the windowing approach.

The starting point of the windowing approach CLP realization here is Constraint (6.56). The predicate initiates an action, if a response backlog occurs, which is the case for window *controller*. There are three possible reactions to a response backlog according to the predicate. Either the guaranteed window is kept, or only the window response requirement is repeated, or the old window requirement is repeated completely (window stimulus and response). The last possibility would trigger Constraint (6.60) and thus would imply the predicate *Resource_Freed(s)*. As there is no other window available on the resource of the *controller* window in the example, this possibility cannot be applied because the resource cannot be freed. The other two possible reactions in Constraint (6.56) result in the two following solutions to the CLP problem of the example.

The first solution is summarized in Table 7.2 and also depicted in Figure 7.5. The guarantee window is kept and thus Constraint (6.58) is triggered. That means that the next window *M2* must be shifted to the right. According to Predicate (6.55) a right shift in this case means that there is a period switch to the right and in the new requirements the window stimulus must be tightened.

The window $M2$ does not have a stimulus buffer. Finally, Constraint (6.65) transitively shifts the last *actuator* window to the right over the period switch. The $M2$ window response must be relaxed, because it does not have a response buffer. The *actuator* window finally has a buffer of 2 and the constraints are fulfilled. The shifting leads to an overall function chain latency of 25 again, because the overlap of the two windows is solved and the buffer is transferred from the *actuator* window to the *controller* window.

		new requirement	
element	type	min	max
sensor	latency	1	13
controller	latency	1	5
actuator	latency	1	3
M1 queued	offset	2	2
M1 transmitted	offset	3	4
M2 queued	offset	5	9
M2 transmitted	offset	0	1

Table 7.2. Solution 1 of the windowing approach example.

The second solution is summarized in Table 7.3 and depicted in the lower part of Figure 7.5. According to Constraint (6.56) in this case only the response of the *controller* window requirement is repeated and Constraint (6.59) is triggered. As a consequence the *controller* window's stimulus must be relaxed because there is no buffer that could be used. Therefore the previous window must be shifted to the left. Thereby Constraint (6.65) triggers the transitive shifting of also the *sensor* window to the left. Again, the resulting window arrangement leads to an overall latency of 25 and thus fulfills the latency constraint of the function chain.

		new requirement	
element	type	min	max
sensor	latency	1	13
controller	latency	1	5
actuator	latency	1	3
M1 queued	offset	0	0
M1 transmitted	offset	1	2
M2 queued	offset	3	7
M2 transmitted	offset	8	9

Table 7.3. Solution 2 of the windowing approach example.

Note that the window arrangements of the two solutions of the example differ only in the relative positions of the windows in time. The window sizes are equal in both solutions. The overall function chain latency of the new window requirements is also equal in both solutions. This way the example highlights the basic functionality of "moving and resizing time windows" by the windowing approach.

The application of the windowing approach can offer several solutions, as one can see in the discussed example. We outline possible cost functions to identify an optimal solution of the windowing approach in Section 8.2.

7.3 Benefits of the TIMEX Methodology

7.3.1 Automatically Derive Subsystem Timing Requirements

In Section 4.7.1 we explain the limitations of related work regarding the derivation of subsystem constraints from given system-level timing constraints. Many approaches to analyze a system's timing behavior assume given task or frame deadlines. The fulfillment of those deadlines can be guaranteed by various analysis approaches.

According to Definition 6.2, the goal of TIMEX methodology is to ensure the system's correct timing behavior on subsystem level instead of system level. Therefore we distinguish function-triggered timing constraints (system level) and timing guarantees (subsystem level). The fulfillment of all timing requirements by their guarantees must ensure the fulfillment of all timing constraints. We formulated this condition in Predicate (6.9).

In all examples that we discuss in this section, for both the windowing approach and the shifting approach, we demonstrate that the system's incorrect timing behavior is indicated by what we call *local timing problems*. Those simply are not fulfilled requirements of either segments or hops, i. e. timing requirements on subsystem level. It is not necessary to perform a system wide timing analysis to realize the incorrect timing behavior. This is an advantage of TIMEX and its semantics of timing requirements. However, to rely on Predicate (6.9), the generation of all timing requirements must be done correctly beforehand. We use predicate logic to formalize our CLP approach that incorporates several CLP constraints, which ensure the validity of Predicate (6.9) both for event-triggered and time-triggered systems.

In Section 6.4.1 we formulate a CLP constraint for each of the three function-triggered timing constraint types of TIMEX for the case of an event-triggered system. These CLP constraints are used in the overall search Predicate (6.39), which initiates the entire shifting approach. Similar, in Section 6.5.2 we formulate a CLP constraint for each of the three function-triggered timing constraint types for the case of a time-triggered system. The search Predicate (6.66) uses these and thus ensures Predicate (6.9) also for the windowing approach.

In the examples discussed for the shifting and windowing approaches the solutions according to our CLP realization automatically ensured correct timing requirements. In the event-triggered case, the sum of the latency requirement values is lower than the latency constraint of the function chain. In the time-triggered case, the length of the windows in the time line also fulfills the overall latency.

To conclude the discussion about how TIMEX overcomes the related work limitation regarding the derivation of subsystem constraints: If an appropriate

timing model like TIMEX is used and the subsystem requirement generation is done right, then the great advantage is that the timing correctness can be ensured on subsystem requirement level, rather than on system constraint level. Our CLP implementation ensures this and Predicate (6.9) is valid for the TIMEX methodology.

7.3.2 Systematically React to Local Timing Problems

Another related work limitation we discuss in Section 4.7.3 is that the timing properties of typical timing models do not qualify as target of timing requirements and for being exchanged between development teams. Those low-level properties like response times, priorities etc. are very unstable and implementation-specific in practice. The consequences of changes of those low-level timing properties to the entire system are not easy to predict. Thus, timing budget negotiation – as proposed by the TIMEX methodology – should be done on more abstract levels. Timing problems can systematically be solved on these abstract levels, which in our model is the level of segments and hops, instead of the above-mentioned low-level timing properties. We discuss the benefits of TIMEX with respect to systematical reaction to local timing problems based on the shifting and windowing approach examples in the following.

Implementation-specific low-level timing properties of a subsystem can change during development. From a system integration perspective, which typically a system designer has, such changes are only important if they affect the guarantees of that subsystem. In other words, the subsystem developer does not need to reveal his implementation-specific low-level timing properties, which we consider as his intellectual property. The level of information exchange between development teams and the system designer solely is the level of segment and hop requirements and guarantees. In the examples, we solve local timing conflicts only based on the comparison of requirement and guarantee values. We are not interested in the detailed task priorities, worst case response times or other schedule information. Further, the output of the requirement generation process also solely are requirements for segment latencies and hop offsets, not requirements for the low-level schedules themselves, like priorities or task offsets. In today's collaboration processes in the automotive industry, this kind of timing requirements gains much more acceptance by subsystem developers and suppliers, because they leave enough freedom of design and respect the suppliers development competence.

The temporary Assumption 4.6 of Chapter 4 is not valid in practice. The system is developed by many different development teams. Thus typical schedule generation approaches are hard to realize in distributed development. If a timing problem like a not fulfilled timing constraint exists in a system design, it is unrealistic to generate an entire new schedule for the whole system (ECUs and busses) and to give completely new requirements for low-level properties to the subsystem developers. Rather, each subsystem developer is responsible for his part of the overall global schedule. The TIMEX approach solely applies a raw schedule frame for the subsystems, by assigning time budgets for them. These budgets are iteratively negotiated. Timing problems can be identified as local timing problems of certain subsystems and the reaction to such problems yields to a minimal impact to the rest of the system.

In the first event-triggered system example of Section 7.2.1 a timing problem exists only for the segments *controller* and *transmitM2*. The solution calculated by the shifting approach is not a complete new requirement setup. The two segments *transmitM1* and *actuator* are not affected by the new solution. In the second, more complex example with two chains, also segments are affected by the seven new solutions, which were not involved in the local timing problems. However, especially when requirements are tightened for a subsystem, the shifting approach ensures that another requirement is relaxed on the same resource and thus redistributes the budgets in a fair manner.

As described in Section 2.1.4, especially time-triggered ECU and bus schedules are tightly coupled with each other. Changes of the schedule of a sender ECU for example can lead to a period loss, if the receiver ECU schedule is not adapted accordingly. The windowing approach example discussed in Section 7.2.2 shows such a local timing problem. The windowing approach however ensures that depending schedules are adapted and also minimizes the impact on other subsystems. The movement of windows on the time line is performed in a way that fair window position and size changes are ensured. That means that a window is moved to a new location, if the specific local schedule is freed at that position (see Constraint (6.51)).

Summarizing, the TIMEX requirement generation approach helps to systematically react to local timing problems. In a not distributed development process, probably also a random re-generation of the global schedule would be acceptable. However, in distributed development TIMEX helps to minimize the impact of local timing problems to other subsystem's timing requirements. Further, TIMEX is an approach to abstract from low-level timing properties if timing requirements and according guarantees are exchanged between system designers and subsystem developers. The low-level timing properties themselves are considered as intellectual property of the subsystem suppliers.

Conclusion

8.1 Summary

In this thesis we presented our approach to derive and iteratively maintain subsystem timing requirements from given function-triggered – and system-independent – timing constraints. As a formal basis we developed TIMEX, a timing model for distributed development of automotive real-time systems. TIMEX is organized in two sections. A TIMEX *function timing* is used to capture implementation-independent timing constraints of automotive functions. As an example of such a function we use a chassis control function throughout the thesis, which is used to stabilize the car during driving. Such functions typically are end-to-end functions from a certain sensor to an observable actuator output and have strict timing constraints due to safety reasons. A TIMEX *system timing* is used to structure the entire automotive system according to the assignment of subsystems to development teams. The end-to-end paths of functions are thus segmented at the border between two different subsystems, such as ECUs and the communication busses. A system timing carries subsystem timing requirements, which are derived from the function-triggered timing constraints.

The derivation of subsystem timing requirements as well as the iterative refinement of the requirements during development can be expressed by formal rules. In this thesis we presented a predicate logic formalization for these rules for both event-triggered and time-triggered systems. The approach takes the current requirement and guarantee setup of all subsystems as input. If at least one requirement is not fulfilled by its guarantee, the fulfillment of all function-triggered timing constraints cannot be assured and a modification of the requirements is triggered. As a benefit of TIMEX, the abstract timing analysis can be performed by the system designer per subsystem and on the level of timing requirements. This model-based approach eases and automates the system designer's timing validation process of the system. If a timing problem exists, our constraint logic programming system can be used to search for possible modified timing requirement setups, which have the highest probability of fulfillment of all timing requirements.

8.2 Outlook

Our work leaves some room for improvements and further investigation. This section gives an outlook of possible future work.

Regarding our constraint logic programming implementation of the shifting and windowing approaches, we did not discuss optimization strategies for the search for solutions. As we showed in Section 7.2, there can be multiple solutions as a timing requirement modification to fix a local timing problem. Shifting and windowing are designed to allow for as many reaction choices as possible and calculate all such possible reactions according to the rules. We assumed engineering work to analyze the solutions and select one of them. However, formalized optimization or selection metrics could be added to the search to automate this process. As an example, a metric could prioritize those solutions, which involve the least resource or subsystem changes, because all changes must be handed to the respective subsystem developers as new requirements.

The functional architecture of our system model only consists of end-to-end functions. Further, these functions can only have sensor inputs and actuator outputs at the boarder of the system to its environment. The functional architecture thus does not provide a concept of subfunctions, which can be composed to functions. Typical automotive functions are end-to-end functions that are visible at the system boarders. Body functions of a car for example are typically controllable by the driver via buttons and respond by actuators, which are visible to the driver, such as light or mechanical reaction. Examples for such body functions are the electric window lift, where the user presses a button (sensor) and expects window movement (actuator reaction), or the turn indicator, where the user pushes a lever (sensor) and expects the indicator lights to turn on (actuator reaction). Chassis functions also use some sensor data as input and produce mechanical actuator output, like the running example (damper control function) of this thesis. Other chassis functions, like for example steering or braking, are also based on driver input, which is made available to the system by sensors, and control recognizable actuators. As already stated above, all those functions fit to the scheme of sensor-to-actuator functions. Our system model and our TIMEX model fit for such sensor-to-actuator functions. In practice however, such functions are often divided into subfunctions. A typical reason for subfunction structures is to reuse basic functionality in several functions. Consider as example the turn indicator again. A car typically has several parallel functions based on the indicator lights, such as left and right turn indication or warning lights. One common subfunction for all these functions could be a turn light controller that mediates between the different sensors (turn indicator lever, warning light button) and actuators (all available turn lights). Such a subfunction does not have direct actuator output, but an internal interface to the actual light controllers. Therefore it could not be modeled within a TIMEX function timing model as presented in this thesis. Other system models we discussed in Section 3.3 are capable of such subfunctions. The TIMEX model could be extended to support timing constraints for subfunctions in its function timing model by also adding internal interfaces.

Observable events, and thus also hops, in our models represent observable actions in the system. The instances of such events can be observed, every time the action is performed by the system. In our current model and its semantics, the instances cannot be limited – or filtered – to a subset of all possible instances. Additional concepts for a more precise definition of observable events are conceivable. One could be interested in only such instances of an observable event, which are defined by some kind of data-dependency. For a send event at a software component's port for example probably only the instances are of interest, which occur when the sent data element has a specific value. The same "occurrence filter" could be applied for signal events or receive events (see Section 3.4). Consider the indicator function explained above. When the controller subfunction receives the sensor input, it decides which lights should be turned on or off. As a result it could send one shared message to all light controllers that contains the information, which lights should be turned on. Timing constraints for the subfunctions (left, right or warning light) would only be valid depending on the content of this message. Thus, the message occurrences could be filtered to fit the right subfunction.

Another possible enhancement for our event model is the concept of composed events. That is, an abstract observable event is defined to occur every time, when several basic observable events actually occur. Timing requirements, as well as other requirements of automotive functions, often are defined using such types of events. As an example consider a wiper function, which is triggered by the driver who pushes a lever. The activation of the function however actually also depends on other input data, such as rain information of the rain sensor. A timing constraint for the wiper function thus must be defined using the additional rain sensor input context. Therefore a composed event could be used. A concept of composed observable events could easily be added to TIMEX. The semantics of the concept and its influence on the requirement generation by the shifting and windowing approach should be investigated.

More generally, timing constraints could depend on certain system states or modes. Car systems often define some standard modes, such as "stopping", "parking" or "high speed driving". Often timing constraints are only valid in the context of a certain state of the system, typically because the functions are not available in other states or because they have different constraints in other states. The timing extensions of AUTOSAR for example support a simple concept for mode switch timing constraints [4]. The system model of this thesis could be extended by such a state or mode concept.

In this thesis we basically investigated simple function event chains with one sensor stimulus and one actuator response event, such as the damper function example. The TIMEX model however also allows for more complex function event chains. Consider the warning light function, which requires all four turn lights to be activated after the warning light button has been pushed. Such a function could be modeled with four response events, one for each light activation. In TIMEX, each of the four paths from the button to the lights can easily be modeled as an own function event chain. Each of the function event chains must then be assigned an own function-triggered timing constraint. The model so far does not provide a simplification that allows for an aggregated, complex function, which consists of several function event chains that belong to the same car function.

Another improvement of the TIMEX model could be a support for complex data paths of a function's stimulus event to its response event. If several possible paths exist, we call such segments with the same stimulus hop but different response hops a "fork". Segments with different stimulus hops but the same response hop are called a "join". Using forks and joins, especially in combination with data-dependent and composed hops can result in very complex function event chain definitions. The derivation of segment and hop requirements from timing constraints for such complex function event chains is certainly more difficult, compared to the rather simple ones that we investigated in this thesis. Our work however already provides the necessary formal model and the basic derivation algorithm that could be extended by rules for more complex function event chains as well.

A

Tool Prototype

During the work on this thesis we developed a tool prototype to support the application of TIMEX. The tool consists of the following parts:

- an Ecore [89] realization and a generated Java implementation of the TIMEX meta-model
- a textual TIMEX model editor based on Xtext [40]
- an implementation of both the shifting and the windowing approach using the constraint programming system ECLiPSe [19]
- a visualization concept for TIMEX models and requirement/guarantee value pairs

The tool is based on Artop, the AUTOSAR Tool Platform [48, 77]. Artop is an implementation of common base functionality for AUTOSAR development tools, such as model and workspace management, import and export of AUTOSAR XML files, and generic tree-based editors for all AUTOSAR models. Artop is based on Eclipse (not to be confused with ECLiPSe). Eclipse is an open source community that develops an open tool platform that is used for many software engineering and programming tools [34].

The four parts of the functionality of our TIMEX tooling listed above were developed as plug-ins for Artop. In the following, we provide a short description of the tool parts.

A.1 Tool Overview

As shown in Figure A.1, the tooling for TIMEX is built up around three different types of model.

- The standardized AUTOSAR model is used as *exchange model* requirement and guarantee data between development teams. The timing extension structure (i. e. necessary event, event chain and timing constraint elements,

without concrete initial values) is automatically generated based on the TIMEX model.

- As *engineering model* the tool provides TIMEX and the according textual model editor (see Section A.2).
- The TIMEX model is used to generate Prolog facts for the constraint solver, which represent the *computation model*. The facts and the implemented rules of the shifting and windowing approach are used to search for new requirement values in each iteration of the TIMEX methodology.

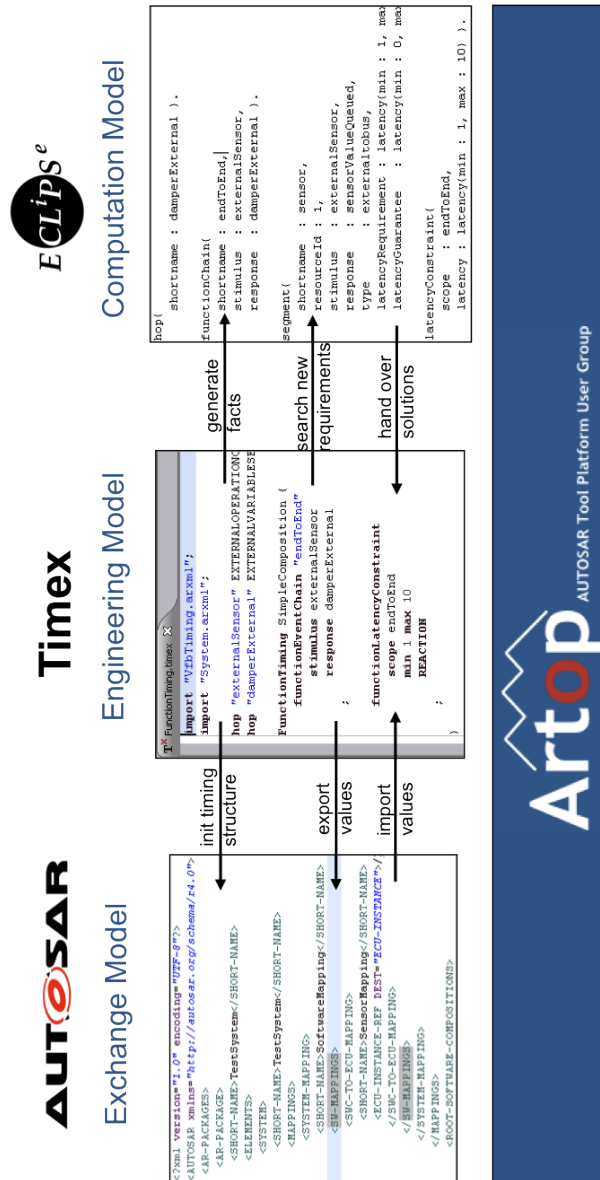
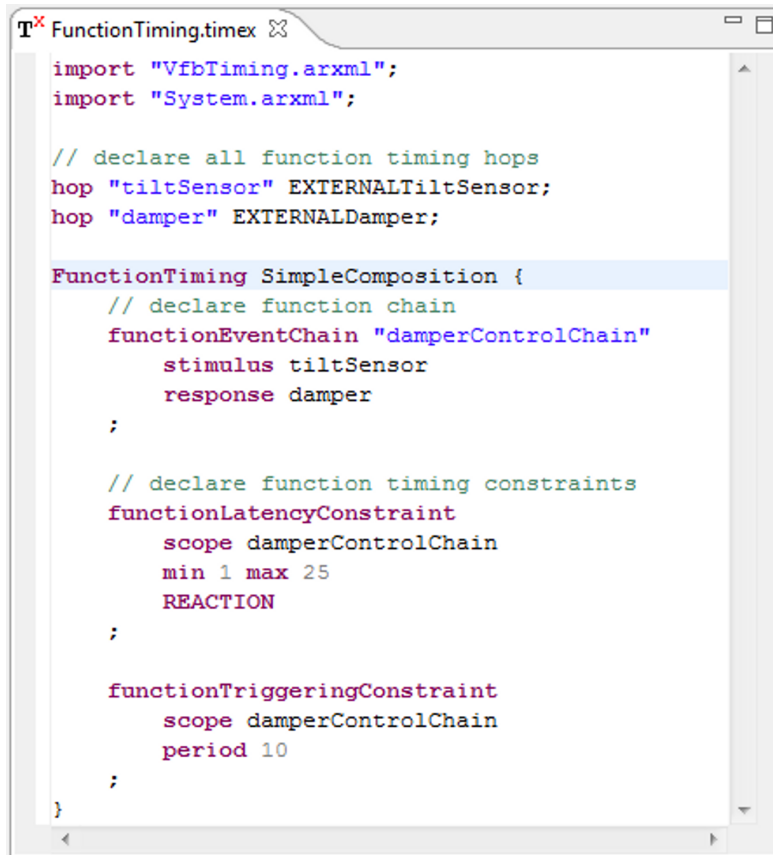


Fig. A.1. Three different model types of the overall TIMEX tooling.

A.2 Textual TIMEX Model Editor

Using Xtext, we created a TIMEX Ecore [89] model and generated a TIMEX textual editor. The editor provides the syntax for both a function timing and a system timing model of TIMEX.



```

Tx FunctionTiming.timex
import "VfbTiming.arxml";
import "System.arxml";

// declare all function timing hops
hop "tiltSensor" EXTERNALTiltSensor;
hop "damper" EXTERNALDamper;

FunctionTiming SimpleComposition {
  // declare function chain
  functionEventChain "damperControlChain"
    stimulus tiltSensor
    response damper
  ;

  // declare function timing constraints
  functionLatencyConstraint
    scope damperControlChain
    min 1 max 25
    REACTION
  ;

  functionTriggeringConstraint
    scope damperControlChain
    period 10
  ;
}

```

Fig. A.2. Example function timing in the textual TIMEX editor.

Figure A.2 displays the function timing of the running example in the textual editor. Hop declarations reference AUTOSAR observable events, which are imported by the editor in the first line. After the hop declaration, a complete function timing is defined, which uses the declared hops. A function timing in our model references a function of our system model.

The industry standard AUTOSAR however does not provide the concept of a "function", because it basically consists of a logical and technical architecture model. Our system model, which in principle is similarly structured, extends the AUTOSAR model with its functional architecture. Nevertheless we want to ensure the possibility to map from one model to the other. As a workaround, it is possible to define a so-called composition component without atomic software components in AUTOSAR as a function and use it to model function-triggered timing constraints. In the example the function timing references a *SimpleComposition* for that purpose.

Figure A.3 displays the system timing of the running example in the textual editor. Similar to the function timing syntax, first all referenced AUTOSAR and TIMEX models are imported. Then all additional hops of the system timing are declared. The system timing in this case is used for the time-triggered realization of the function. Therefore a hop *cycleStart* is defined.



```

T× SystemTiming.timex
import "SignalTiming.xml";
import "../FunctionTiming.timex";
import "CycleTiming.arxml";
import "../System.arxml";
import "../System_Cluster.arxml";

hop "m1Queued" SIGNALSensorValueSensorECUQueued;
hop "m1Transmitted" SIGNALSensorValueTRANSMITTEDONBUS;
hop "m2Queued" SIGNALDamperValueControllerQueued;
hop "m2Transmitted" SIGNALDamperValueTRANSMITTEDONBUS;

hop "cycleStart" CYCLESTARTFr_Cluster;

SystemTiming SimpleSystem_FrSynch {
  segment "sensor" resourceId 1
    stimulus tiltSensor
    response m1Queued
  ;

  segment "transmitM1" resourceId 2
    stimulus m1Queued
    response m1Transmitted
  ;

  segment "controller" resourceId 3
    stimulus m1Transmitted
    response m2Queued
  ;

  segment "transmitM2" resourceId 2
    stimulus m2Queued
    response m2Transmitted
  ;

  segment "actuator" resourceId 4
    stimulus m2Transmitted
    response damper
  ;
}

```

Fig. A.3. Example system timing in the textual TIMEX editor.

A.3 Timex Visualization

Figure A.4 shows our visualization of a TIMEX system model for an event-triggered system. A function chain is displayed as a sequence of segment lines. The segment lines are colored according to their requirement fulfillment. If a segment's requirement is not fulfilled by its guarantee, then it is colored in red. A circle at a hop indicates a triggering requirement for that hop.

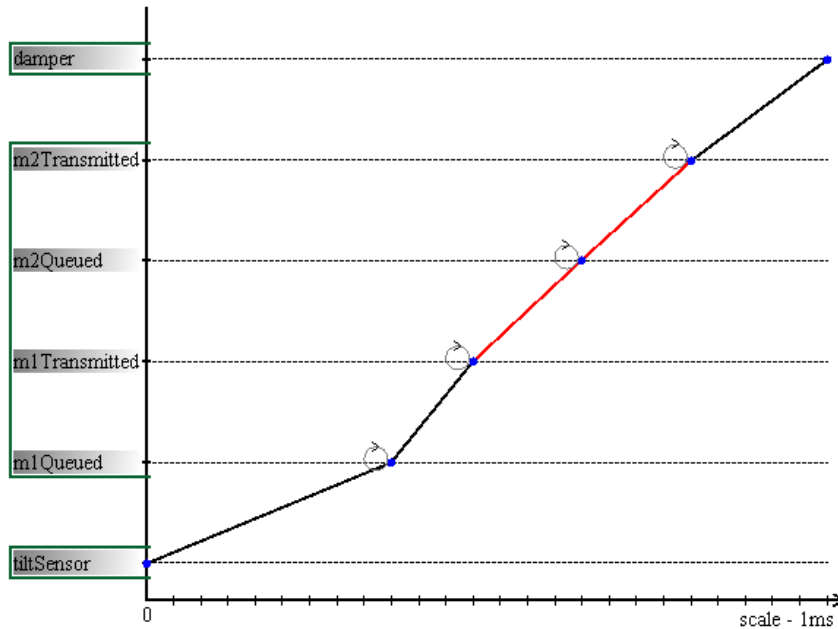


Fig. A.4. Visualization of the example TIMEX model for an event-triggered system.

Figure A.5 shows the system timing for a time-triggered system. Hop offsets are displayed relatively to the period start. For simplification, additional other display types for system timing models are available. Figure A.6 as such a simplification shows the same system timing without requirement values and concentrates solely on the structure.

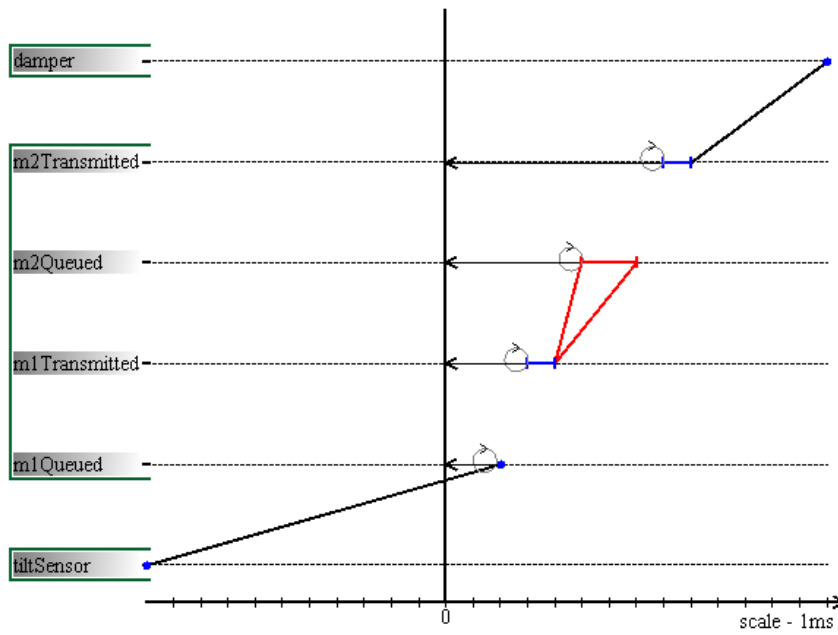


Fig. A.5. Visualization of the example TIMEX model for a time-triggered system.

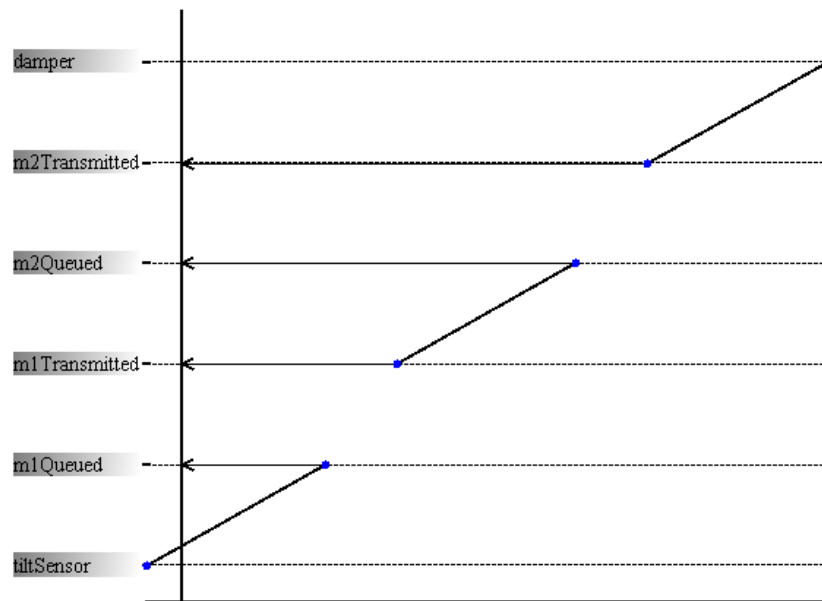


Fig. A.6. Structural visualization of the example TIMEX model for a time-triggered system.

A.4 Requirement and Guarantee Value Table Editor

An easy way of editing requirement and guarantee values is provided by our table editor. Each line in the editor represents one requirement and guarantee pair.

Figure A.7 displays the table editor that has loaded the system timing model for the event-triggered system case. If a requirement is not fulfilled, it is colored in red. An over-fulfilled requirement is colored in darker green.

Requirement Ty...	Element	R: Min	R: Max	G: Min	G: Max
Latency	td_sensor	1	9	1	6
Latency	td_transmitM1	1	3	1	3
Latency	td_controller	1	4	1	5
Latency	td_transmitM2	1	4	1	5
Latency	td_actuator	1	5	1	5
Triggering	td_SensorValue_IPdu_Frame_Queued	Period	10	Period	10
Triggering	td_SensorValue_IPdu_Frame_Transmitted	Period	10	Period	10
Triggering	td_DamperValue_IPdu_Frame_Queued	Period	10	Period	10
Triggering	td_DamperValue_IPdu_Frame_Transmitted	Period	10	Period	10

Fig. A.7. Table editor view of the requirements and guarantees of the event-triggered example.

Figure A.8 displays the table editor that has loaded the system timing model for the time-triggered system implementation. Here, additionally hop offsets are included in the table. Changes in the table can directly be saved to the according AUTOSAR XML file by clicking "Save Changes".

The screenshot shows a window titled "Timing Editor" with a search bar for "Filter elements:" and two buttons: "Refresh Table" and "Save Changes". Below is a table with columns: Requirement Ty..., Element, R: Min, R: Max, G: Min, and G: Max. The table contains 12 rows of data, with alternating green and red background colors for rows.

Requirement Ty...	Element	R: Min	R: Max	G: Min	G: Max
Latency	td_sensor	1	13	1	13
Latency	td_controller	1	3	1	5
Latency	td_actuator	1	5	1	3
Offset	td_SensorValue_IPdu_Frame_Queued	2	2	2	2
Triggering	td_SensorValue_IPdu_Frame_Queued	Period	10	Period	10
Offset	td_SensorValue_IPdu_Frame_Transmitted	3	4	3	4
Triggering	td_SensorValue_IPdu_Frame_Transmitted	Period	10	Period	10
Offset	td_DamperValue_IPdu_Frame_Queued	5	7	5	9
Triggering	td_DamperValue_IPdu_Frame_Queued	Period	10	Period	10
Offset	td_DamperValue_IPdu_Frame_Transmitted	8	9	8	9
Triggering	td_DamperValue_IPdu_Frame_Transmitted	Period	10	Period	10

Fig. A.8. Table editor view of the requirements and guarantees of the time-triggered example.

References

1. N. Audsley. Deadline Monotonic Scheduling. Technical report, Department of Computer Science, University of York, October 1990.
2. N. C. Audsley. Optimal Priority Assignment And Feasibility Of Static Priority Tasks With Arbitrary Start Times. Technical report, Department of Computer Science, University of York, 1991.
3. N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atalanta, 1991.
4. AUTOSAR Development Partnership. Specification of Timing Extensions, Version 1.0.0, Release 4.0, Rev 1.
5. AUTOSAR Development Partnership. Main Requirements, Version 2.1.0, Release 4.0, Rev 1.
6. AUTOSAR Development Partnership. Specification of Operating System, Version 4.0.0, Release 4.0, Rev 1.
7. G. Blache and S. Krishnan. Design of safety critical systems with ASCET. In *Proceedings of ISSRE*, 2009.
8. M. Broy. Challenges in Automotive Software Engineering. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 33–42, New York, NY, USA, 2006. ACM Press.
9. M. Broy. The 'Grand Challenge' in Informatics: Engineering Software-Intensive Systems. *Computer*, 39(10):72–80, 2006.
10. M. Broy. Time and Causality in Interactive Distributed Systems. Summer School, August 2008.
11. M. Broy, F. Dederich, C. Dendorfer, M. Fuchs, T. Gritzner, and R. Weber. The Design of Distributed Systems - An Introduction to FOCUS. Technical Report TUM-I9202, Technische Univerität München, 1992.
12. M. Broy, M. Feilkas, J. Grünbauer, A. Gruler, A. Harhurin, J. Hartmann, B. Penzenstadler, B. Schätz, and D. Wild. Umfassendes Architekturmodell für das Engineering eingebetteter Software-intensiver Systeme. Technical Report TUM-I0816, Technische Universität München, jun 2008.
13. M. Broy, M. Gleirscher, P. Kluge, W. Krenzer, S. Merenda, and D. Wild. Automotive Architecture Framework: Towards a Holistic and Standardised System Architecture Description. . Technical Report TUM-I0915, Technische Universität München, jul 2009.
14. M. Broy, I. H. Krüger, A. Pretschner, and C. Salzmann. Engineering Automotive Software. In *Proceedings of the IEEE*, volume 95, pages 356 – 373, Feb. 2007.
15. M. Broy and K. Stolen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer, 2001.
16. J.-Y. Brunel, M. Di Natale, A. Ferrari, P. Giusto, and L. Lavagno. SoftContract: an Assertion-Based Software Development Process that Enables Design-by-Contract. In *DATE '04: Proceedings of the conference on Design, automation*

- and test in Europe*, page 10358, Washington, DC, USA, 2004. IEEE Computer Society.
17. S. Bunzel. Overview on AUTOSAR Cooperation. 2nd AUTOSAR Open Conference, May 2010.
 18. G. C. Buttazzo. Rate monotonic vs. EDF: judgment day. *Real-Time Syst.*, 29(1):5–26, 2005.
 19. A. M. Cheadle, W. Harvey, A. J. Sadler, J. Schimpf, K. Shen, and M. G. Wallace. ECLiPSe - A Tutorial Introduction. Technical report, Cisco Systems, Inc., April 8 2010.
 20. H. Chetto and M. Chetto. Some Results of the Earliest Deadline Scheduling Algorithm. *IEEE Trans. Softw. Eng.*, 15(10):1261–1269, 1989.
 21. T. Chung and H. Dietz. Adaptive genetic algorithm: Scheduling hard real-time control programs with arbitrary timing constraints, Submitted for publication, January 1995.
 22. W. Damm, A. Votintseva, A. Metzner, B. Josko, T. Peikenkamp, and E. Böde. Boosting Re-use of Embedded Automotive Applications Through Rich Components. In *Foundations of Interface Technologies*, 2005.
 23. R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien. Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Syst.*, 35(3):239–272, 2007.
 24. S. Ding, N. Murakami, H. Tomiyama, and H. Takada. A GA-based scheduling method for FlexRay systems. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 110–113, New York, NY, USA, 2005. ACM.
 25. A. Easwaran, I. Shin, I. Lee, and O. Sokolsky. Bounding preemptions under EDF and RM schedulers. Technical report, University of Pennsylvania, 2006.
 26. C. Ekelin and J. Jonsson. Real-Time System Constraints: Where do They Come From and Where do They Go? In *Proc. of the Int. Workshop on Real-Time Constraints*, pages 53–57, 1999.
 27. C. Ekelin and J. Jonsson. Solving Embedded System Scheduling Problems using Constraint Programming. Technical report, Dept. of Computer Engineering, Chalmers University of Technology, 2000.
 28. W. Elmenreich, G. Bauer, and H. Kopetz. The Time-Triggered Paradigm. In *Proceedings of the Workshop on Time-Triggered and Real-Time Communication*, Manno, Switzerland, Dec. 2003.
 29. N. Feiertag, K. Richter, J. Nordlander, and J. Jonsson. A Compositional Framework for End-to-End Path Delay Calculation of Automotive Systems under Different Path Semantics. In *Proceedings of the IEEE Real-Time System Symposium (RTSS), Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS'08)*, December 2008.
 30. M. Feilkas, A. H. J. Hartmann, D. Ratiu, and W. Schwitzer. Motivation and Introduction of a System of Abstraction Layers for Embedded Systems. Technical Report TUM-I0925, Technische Universität München, 2009.
 31. H. Fennel, S. Bunzel, H. Heinecke, and et al. Achievements and Exploitation of the AUTOSAR Development Partnership. In *Proceedings of Convergence 2006*, 2006.
 32. FlexRay Consortium. FlexRay Protocol Specification Version 2.1 Revision A, 2005.
 33. S. Fürst, J. Mössinger, S. Bunzel, T. Weber, F. Kirschke-Biller, P. Heitkämper, G. Kinkelin, K. Nishikawa, and K. Lange. AUTOSAR - A Worldwide Standard is on the Road. In *Proc. of 14th International VDI Congress Electronic Systems for Vehicles*, 2009.
 34. E. Gamma and K. Beck. *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2003.

35. M. R. Garey and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, January 1979.
36. M. Gehrke, M. Hirsch, W. Schäfer, O. Niggemann, D. Stichling, and U. Nickel. Typisierung und Verifikation zeitlicher Anforderungen automotiver Software Systeme. In *Proc. of the Dagstuhl-Workshop: Model-Based Development of Embedded Systems (MBEES), 15.-18.1.2007, Schloss Dagstuhl, Germany*, 2007.
37. R. Gerber, S. Hong, and M. Saksena. Guaranteeing Real-Time Requirements With Resource-Based Calibration of Periodic Processes. *IEEE Trans. Softw. Eng.*, 21(7):579–592, 1995.
38. M. Grenier, L. Havet, and N. Navet. Configuring the communication on FlexRay: the case of the static segment. *extended version of a paper published at ERTS2008, available at <http://www.realtimeatwork.com>*, 2008.
39. K. Gresser. An Event Model for Deadline Verification of Hard Real-Time Systems. *Proc. of fifth Euromicro Workshop on Real-Time Systems*, 5, 1993.
40. R. C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley, 2009.
41. A. Gruler, A. Harhurin, and J. Hartmann. Modeling the Functionality of Multi-Functional Software Systems. In *14th Annual IEEE International Conference on the Engineering of Computer Based Systems (ECBS)*, pages 349 – 358, Los Alamitos, CA, USA, 2007. IEEE Computer Society. available at <http://doi.ieeecomputersociety.org/10.1109/ECBS.2007.54>.
42. N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Pub., 1993.
43. A. Hamann, M. Jersak, K. Richter, and R. Ernst. Design Space Exploration and System Optimization with SymTA/S " Symbolic Timing Analysis for Systems. In *RTSS '04: Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS'04)*, pages 469–478, Washington, DC, USA, 2004. IEEE Computer Society.
44. D. Harel and A. Pnueli. On the development of reactive systems. pages 477–498, 1985.
45. A. Harhurin, J. Hartmann, and D. Ratiu. Motivation and Formal Foundations of a Comprehensive Modeling Theory for Embedded Systems. Technical Report TUM-I0924, Technische Universität München, 2009.
46. H. Heinecke, J. Bielefeld, K.-P. Schnelle, and N. M. et al. AUTOSAR - Current results and preparations for exploitation. In *Proc. of 7th Euroforum Conference*, 2006.
47. H. Heinecke, W. Damm, B. Josko, A. Metzner, H. Kopetz, A. Sangiovanni-Vincentelli, and M. D. Natale. Software components for reliable automotive systems. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, pages 549–554, New York, NY, USA, 2008. ACM.
48. H. Heinecke, M. Rudorfer, P. Hoser, C. Ainhauser, and O. Scheickl. Enabling of AUTOSAR system design using Eclipse-based tooling. In *Proceedings of ERTS, 2008*.
49. R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System Level Performance Analysis - the SymTA/S Approach. IEE Proceedings Computers and Digital Techniques, 2005.
50. ISO Standard ISO 11898-1:2003. Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling, 2003.
51. M. Joseph and P. K. Pandya. Finding Response Times in a Real-Time System. *Comput. J.*, 29(5):390–395, 1986.
52. R. Kirner, R. Lang, G. Freiberger, and P. Puschner. Fully Automatic Worst-Case Execution Time Analysis for Matlab/Simulink Models, June 2002.
53. H. Kopetz. Why Time-Triggered Architectures will Succeed in Large Hard Real-Time Systems. In *Proceedings of the 5th IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 2–9, Chenju, Korea, Aug. 1995.

54. H. Kopetz. The Time Triggered Architecture. *ERCIM NEWS*, 52(52):24–25, Jan. 2002.
55. H. Kopetz and G. Grünsteidl. TTP-A Protocol for Fault-Tolerant Real-Time Systems. *Computer*, 27(1):14–23, 1994.
56. J. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. *Proceedings of Real-Time Systems Symposium*, 11:201–209, 1990.
57. J. P. Lehoczky, L. Sha, and Y. Ding. Rate-Monotonic Scheduling Algorithm: Exact characterization and average case behavior. In *Proc. of the 11th IEEE Real-time Systems Symposium*, pages 166–171, Dec. 1989.
58. J. Y. T. Leung and J. Whitehead. On The Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks. *Performance Evaluation*, 2:237–250, Dec 1982.
59. C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.
60. J. W. S. W. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.
61. R. S. Marques, N. Navet, and F. Simonet-Lion. Frame Packing under Real-time Constraints. *Proc. of 5th IFAC Intl. Conference on Fieldbus Systems and their Applications*, 5, 2007.
62. Mathworks. The Mathworks Simulink User Manual. Technical Report R2010b MathWorks Documentation, <http://www.mathworks.de/help/toolbox/simulink/ug/bqchgknk.html>, 2010.
63. M. Mächtel. *Echtzeitsysteme - Script zur Vorlesung an der FH München*. FH München, 2004/03/06 Revision 1.21.
64. B. Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992.
65. P. Montag, S. Görzig, and P. Levi. Applying static timing analysis to component architectures. In *SEAS '06: Proceedings of the 2006 international workshop on Software engineering for automotive systems*, pages 21–28, New York, NY, USA, 2006. ACM Press.
66. N. Navet, Y. Song, F. Simonot-Lion, and C. Wilwert. Trends in Automotive Communication Systems. *Proceedings of the IEEE*, 93(6):1204–1223, 2005.
67. H. Negele. Systems Engineering Challenges and Solutions from an Automotive Perspective. Keynote Presentation. In *International Council on Systems Engineering (INCOSE) Symposium 2006*, Orlando, FL, June 27th 2006.
68. R. Nossal and T. Galla. Solving NP-Complete Problems in Real-Time System Design by Multichromosome Genetic Algorithms. In *Proceedings of the SIGPLAN 1997 Workshop on Languages, Compilers, and Tools for Real-Time Systems*, pages 68–76. ACM SIGPLAN, June 1997.
69. OSEK/VDX. Time Triggered Operating System, Version 1.0, July 24th, 2001.
70. OSEK/VDX. Operating System Specification, Version 2.2.3, February 17th, 2005.
71. R. Racu, K. Richter, R. Ernst, and M. Jersak. A Virtual Platform for Architecture Integration and Optimization in Automotive Communication Networks. In *Proceedings of SAE World Congress & Exhibition*, April 2007.
72. S. Reichelt, O. Scheickl, and G. Tabanoglu. The influence of real-time constraints on the design of FlexRay-based systems. In *DATE*, pages 858–863, 2009.
73. K. Richter. *Compositional Scheduling Analysis Using Standard Event Models*. PhD thesis, Technical University Braunschweig, Germany, 2005.
74. K. Richter and R. Ernst. Event Model Interfaces for Heterogeneous System Analysis. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '02, pages 506–, Washington, DC, USA, 2002. IEEE Computer Society.
75. T. Ringler. *Entwicklung und Analyse zeitgesteuerter Systeme*. PhD thesis, Universität Stuttgart, Fakultät für Elektrotechnik und Informationstechnik, 2002.
76. M. Rudorfer, P. Hoser, O. Scheickl, H. Heinecke, T. Ochs, M. Thiede, and M. Mössmer. Being on time using AUTOSAR methodology. *Elektronik automotive*, S2 - Special Issue AUTOSAR:12–14, 2007.

77. M. Rudorfer, C. Knüchel, S. Voget, S. Eberle, and A. Loyer. Artop - an Ecosystem Approach for Collaborative AUTOSAR Tool Development. In *Proceedings of ERTS2*, 2010.
78. R. Saket and N. Navet. Frame packing algorithms for automotive applications. *J. Embedded Comput.*, 2(1):93–102, 2006.
79. M. Saksena. Real-Time System Design: A Temporal Perspective. In *Proc. of IEEE Canadian Conference on Electrical and Computer Engineering*, pages 405–408, 1998.
80. O. Scheickl. Spezifikation und Implementierung von Metriken zur Analyse des Echtzeitverhaltens von verteilten automotive Systemen. Master’s thesis, TU München, Fakultät für Informatik, 2007.
81. O. Scheickl, C. Ainhauser, and M. Rudorfer. Distributed Development of Automotive Real-time Systems based on Function-triggered Timing Constraints. In *Proceedings of ERTS2*, 2010.
82. O. Scheickl and M. Rudorfer. Automotive Real Time Development Using a Timing-augmented AUTOSAR Specification. *Proceedings of ERTS2008*, 4, 2008.
83. O. Scheickl, M. Rudorfer, C. Ainhauser, N. Feiertag, and K. Richter. How Timing Interfaces in AUTOSAR can Improve Distributed Development of Real-Time Software. In *GI Jahrestagung (2)*, pages 662–667, 2008.
84. K. Schild and J. Würtz. Scheduling of Time-Triggered Real-Time Systems. *Constraints*, 5(4):335–357, 2000.
85. J. Schäuffele and T. Zurawka. *Automotive Software Engineering: Principles, Processes, Methods, and Tools*. Vieweg, 3rd edition, 2006.
86. L. Sha, R. Rajkumar, and S. S. Sathaye. Generalized rate-monotonic scheduling theory: A framework for developing real-time systems. In *Proceedings of the IEEE*, volume 82, page 68.82, January 1994.
87. F. Simonot Lion and Y. Trinquet. Vehicle Functional Domains and Their Requirements. In Nicolas Navet and Françoise Simonot Lion, editors, *Automotive Embedded Systems Handbook*, Industrial Information Technology Series. Taylor & Francis / CRC Press, 2008.
88. B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic Task Scheduling for Hard Real-Time Systems. *Real-Time Systems*, 1(1):27–60, 1989.
89. D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.
90. A. S. Tanenbaum and M. v. Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.
91. A. J. Tavares and C. A. Couto. A Machine Independent WCET Predictor for Microcontrollers and DSPs. In *Proceedings of the IEEE International Symposium on Industrial Electronics*, volume 2, pages 989–994, 2001.
92. L. Thiele, S. Chakraborty, and M. Naedele. Real-time Calculus for Scheduling Hard Real-Time Systems. In *Proceedings of Circuits and Systems*, 2000.
93. J. Thyssen, D. Ratiu, W. Schwitzer, A. Harhurin, M. Feilkas, and E. Thaden. A System for Seamless Abstraction Layers for Model-based Development of Embedded Software. In *Proceedings of Envision 2020 Workshop*, 2010.
94. Tindell, Burns, and Wellings. Allocating Hard Real-Time Tasks: An NP-Hard Problem Made Easy. *RTSYSTS: Real-Time Systems*, 4, 1992.
95. K. Tindell, A. Burns, and A. Wellings. Calculating Controller Area Network (CAN) Message Response Times. *Control Engineering Practice*, 3:1163–1169, 1995.
96. P. Uthaisombut. Generalization of EDF and LLF: Identifying All Optimal Online Algorithms for Minimizing Maximum Lateness. *Algorithmica*, 50:312–328, January 2008.
97. E. Weisstein. Floyd-Warshall Algorithm. Wolfram MathWorld <http://mathworld.wolfram.com/Floyd-WarshallAlgorithm.html>, August 2010.