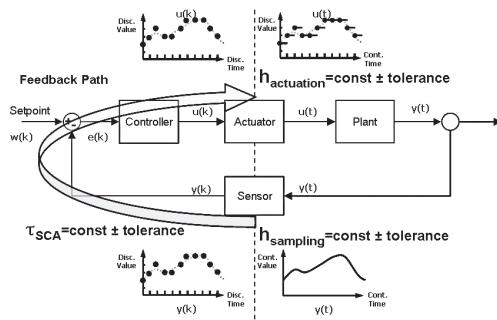
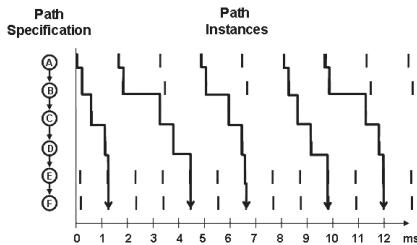


A Timing Model for Real-Time Control-Systems and its Application on Simulation and Monitoring of AUTOSAR Systems

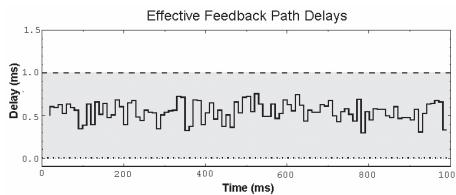


Dissertation

Patrick Frey



Universität Ulm



2010



ulm university universität
u**u****l****m**

**A Timing Model for Real-Time Control-Systems and its
Application on Simulation and Monitoring of AUTOSAR
Systems**

DISSERTATION

zur Erlangung des Doktorgrades **Dr. rer. nat.**
der Fakultät für Ingenieurwissenschaften und Informatik
der Universität Ulm

Patrick Christopher Frey
aus Göppingen

Universität Ulm
Fakultät für Ingenieurwissenschaften und Informatik
Institut für Programmiermethodik und Compilerbau
Institutsdirektor: Prof. Dr. Helmuth Partsch

2010

Amtierender Dekan: Prof. Dr.-Ing. Klaus Dietmayer

Erstgutachter: Prof. Dr. rer. nat. Helmut Partsch

Zweitgutachter: Prof. Dr.-Ing. Frank Slomka

Drittgutachter: Prof. Dr.-Ing. Dr. h.c. Peter Göhner

Tag der Promotion: 6. Dezember 2010

A Timing Model for Real-Time Control-Systems and its Application on Simulation and Monitoring of AUTOSAR Systems

Abstract

AUTOSAR is a common initiative of the automotive industry with the goal to standardize substantial aspects of the software development for automotive embedded systems by covering them in an integrated, rigorous approach. For this purpose, AUTOSAR defines a multitude of concepts such as a standardized software architecture for electronic control units, a software component technology and a development methodology.

Many automotive applications which are realized in software are real-time applications, i.e., applications where certain timing requirements are imposed on the temporal execution in an embedded computer system. The AUTOSAR standard does not yet provide concepts in order to express application-specific timing requirements and to evaluate their degree of fulfillment. The goal of this thesis is to enable the description of application-specific timing requirements in AUTOSAR systems and to provide a means to evaluate their degree of fulfillment.

In this thesis, timing requirements are mainly considered in the context of a specific yet important class of real-time applications: control applications. Compared to classic deadline-centric timing requirements of reactive applications, these impose specific timing requirements towards the execution in an embedded computer system. These stem from the discrete-time realization of continuous-time descriptions of dynamic processes. For example, the minimization of the latency between a sensor data acquisition and an output data actuation that is based upon the sensor data is such a timing requirement, or the maintenance of constant sensor data acquisition rates and output data actuation rates.

The formal description of timing requirements of real-time applications is a specific challenge. Timing requirements such as those on reaction times or path latencies must be formulated with respect to input and output signals that stand in causal relation to each other, i.e., with respect to input and output signals for which there exists a continuous signal path in the embedded computer system. The description of the signal path is, however, the complex challenge: the employment of concepts for modularization and hierarchical composition used for the description of function and software architectures leads to a segmentation of signal paths such that there are currently no adequate means to describe signal paths. Consequently, application-specific timing requirements cannot be adequately described.

A solution for the description of signal paths and the determination of timing properties is provided in this thesis. The introduced concepts allow the description of signal paths as chains of cause-and-effect on the basis of events and actions that need to be observed in a system. The event chains denote signal paths that can then be associated with timing requirements. A special concept for the hierarchical description of these chains of cause-and-effect allows the seamless integration with concepts for modularization as employed in component-based development approaches. From the hierarchical signal path specifications, an instrumentation of the source code of the embedded computer system can be automatically derived such instances of events and actions can be captured during simulation- or monitoring experiments. Furthermore, new kinds of path analysis algorithms are introduced that allow the determination of effective instances of chains of cause-and-effect and signal paths on the basis event traces recorded from simulation- or monitoring experiments. From the effective path instances determined by means of the path analysis algorithms, timing properties can be derived (e.g., reaction times, path delays). These can then visualized by means of a new type of diagram introduced in the thesis, the so-called Timing Oscilloscope Diagram, such that the degree of fulfillment of the timing requirements can be directly evaluated.

The results of the work presented in this thesis not only provide the necessary concepts for a Timing Model for AUTOSAR, they are also important in a more general context: they offer an integrated approach firstly for an adequate description of application-specific timing requirements, especially those of control applications, based on descriptions of signal paths, and secondly for the determination and visualization of corresponding timing properties such that the degree of fulfillment of the latter can be evaluated. These results can be applied also in other domains that are currently not addressed by AUTOSAR such as the aerospace and automation domains where embedded real-time applications, especially control applications, can also be found.

A Timing Model for Real-Time Control-Systems and its Application on Simulation and Monitoring of AUTOSAR Systems

Kurzfassung

AUTOSAR ist eine gemeinsame Initiative der Automobilindustrie mit dem Ziel wesentliche Aspekte der Softwareentwicklung für eingebettete Systeme im Automobil zu standardisieren und ganzheitlich abzudecken. Hierzu definiert AUTOSAR eine Vielzahl an Konzepten, darunter eine standardisierte Steuergerätesoftwarearchitektur, eine Softwarekomponententechnologie und eine Entwicklungsmethodik.

Viele Anwendungen im Automobil, die in Software realisiert werden, sind Echtzeitanwendungen, d.h. Anwendungen, die bestimmte zeitliche Anforderungen an die Ausführung in einem eingebetteten Rechnersystem stellen. Der AUTOSAR Standard bietet bislang jedoch noch keine Konzepte, um anwendungsspezifische zeitliche Anforderungen zu formulieren und den Grad ihrer Erfüllung zu überprüfen. Ziel dieser Arbeit ist es, die Beschreibung anwendungsspezifischer zeitlicher Anforderungen in AUTOSAR Systemen zu ermöglichen und eine Möglichkeit zu bieten den Grad ihrer Erfüllung zu überprüfen.

In dieser Arbeit werden zeitliche Anforderungen hauptsächlich im Kontext einer bestimmten, jedoch wichtigen Klasse von Echtzeitanwendungen, den regelungstechnischen Anwendungen, betrachtet. Sie stellen neben den klassischen Echtzeitanforderungen reaktiver Systeme besondere zeitliche Anforderungen an eine Implementierung in einem eingebetteten Rechnersystem. Diese ergeben sich aus der zeitdiskreten Realisierung von zeitkontinuierlichen Beschreibungen dynamischer Prozesse. Zu den zeitlichen Anforderungen zählt beispielsweise die Minimierung der Latenzzeit zwischen einer Sensordatenerfassung und einer darauf basierten Aktuatoransteuerung, oder die Aufrechterhaltung konstanter Sensordatenerfassungsraten und Ausgangsdatenaktuierungsgraten.

Die formale Modellierung zeitlicher Anforderungen von Echtzeitanwendungen stellt eine besondere Herausforderung dar. Zeitliche Anforderungen wie beispielsweise Reaktionszeiten oder Pfadlatenzzeiten müssen in Bezug auf in kausalen Relation stehender Eingangs- und Ausgangssignale formuliert werden, d.h. in Bezug auf Eingangs- und Ausgangssignale zwischen denen es im eingebetteten Rechnersystem einen durchgängigen Signalpfad gibt. Die Beschreibung des Signalpfads stellt dabei die besonders komplizierte Herausforderung dar: die Verwendung von Konzepten zur Modularisierung und Hierarchiebildung bei der Beschreibung von Funktions- und Softwarearchitekturen führt zu einer Segmentierung von Signalpfaden, so dass

es bislang keine passenden Konzepte zur Beschreibung von Signalpfaden gibt und zeitliche Anforderungen somit auch nicht angemessen modelliert werden können.

Eine Lösung für die Spezifikation von Signalpfaden und der Bestimmung von Zeiteigenschaften wird in dieser Arbeit vorgestellt. Die eingeführten Konzepte ermöglichen es, Signalpfade in Form von Wirkketten auf der Basis von zu beobachtenden Ereignissen und Aktionen zu modellieren und mit zeitlichen Anforderungen zu verknüpfen. Ein spezielles Konzept zur hierarchischen Beschreibung von Wirkketten erlaubt eine nahtlose Integration mit Modularisierungskonzepten wie sie in komponenten-basierten Entwicklungsansätzen verwendet werden. Von den hierarchischen Signalpfadspezifikationen kann eine Instrumentierung des Quellcodes des eingebetteten Rechnersystems abgeleitet werden, so dass Instanzen der Ereignisse und Aktionen in Simulations- und Monitoringexperimenten erfasst werden können. Des Weiteren werden neue Pfadanalysealgorithmen vorgestellt die zur Bestimmung von konkreten Instanzen von Wirkketten und Signalpfaden auf der Basis der Ereignisspuren dienen die während Simulations- oder Monitoringexperimenten aufgezeichnet wurden. Aus den mit Hilfe der Pfadanalysealgorithmen bestimmten Pfadinstanzen lassen sich direkt zeitliche Eigenschaften ableiten (z.B. Reaktionszeiten, Pfadlatenzzeiten). Diese können in einem in der Arbeit entwickelten neuen Typ von Diagramm, dem sogenannten Timing-Oszilloskop Diagramm, grafisch dargestellt werden, so dass sich zeitliche Eigenschaften gegenüber zeitlichen Anforderungen direkt überprüfen lassen.

Die Ergebnisse, die in dieser Arbeit vorgestellt werden, stellen nicht nur die notwendigen Konzepte für ein Timing Modell für AUTOSAR bereit, sondern sie sind auch in einem generelleren Kontext von Bedeutung: sie bieten einen integrierten Ansatz um zum einen anwendungsspezifische zeitliche Anforderungen, insbesondere solche regelungstechnischer Anwendungen, angemessen zu beschreiben, und zum anderen zeitliche Eigenschaften zu bestimmen und diese zu visualisieren, so dass der Grad der Erfüllung der zeitlichen Anforderungen überprüft werden kann. Diese Ergebnisse lassen sich auch auf andere Domänen als die, die momentan von AUTOSAR adressiert wird, anwenden, wie beispielsweise die Luftfahrttechnik oder die Automatisierungstechnik, in denen es ebenfalls eingebettete Echtzeitanwendungen, insbesondere regelungstechnische Anwendungen, gibt.

Danksagung

Die vorliegende Arbeit entstand während meiner Tätigkeit als Doktorand bei der ETAS GmbH im Rahmen des Doktorandenprogramms der Robert Bosch GmbH. Ich möchte mich bei der ETAS GmbH, insbesondere bei Herrn Dr. Christoph Störmer und Herrn Dr. Ulrich Freund, bedanken mir die Möglichkeit gegeben zu haben diese Arbeit anzufertigen. Bei Herrn Prof. Dr. Helmut Partsch möchte ich mich für die Betreuung der Arbeit von universitärer Seite aus bedanken.

Mein Dank gilt Herrn Prof. Dr. Helmut Partsch für die Übernahme des Erstgutachtens. Des Weiteren danke ich Herrn Prof. Dr. Frank Slomka für seine Beiträge zum Gelingen der Arbeit und für die Übernahme des Zweitgutachtens. Herrn Prof. Dr. Peter Göhner danke ich recht herzlich für die Übernahme des Drittgutachtens.

Insbesondere möchte ich der Abteilung Engineering Advanced (ENA) danken, in der ich die 3 1/4 Jahre meiner Doktorandenzeit verbracht habe, allen voran Dr. Christoph Störmer für das in mich gesetzte Vertrauen, sowie Dr. Wilrid Dubitzky, Oliver Wieland, Dr. Andrew Borg und Dr. Gary Morgan für die moralische Unterstützung.

Dr. Ulrich Freund gilt mein besonderer Dank für die fachliche Anleitung, insbesondere in die für mich seiner Zeit neuen Themenkomplexe AUTOSAR und Regelungstechnik, und sein Vertrauen in das Gelingen der Arbeit.

Wichtig für das Gelingen dieser Arbeit waren auch Studenten, die im Rahmen ihrer Diplom- und Masterarbeit in meinem Themengebiet arbeiteten. Hierbei möchte ich besonders Daniel Munzinger und Fabian May für ihre Unterstützung bei der Umsetzung der AUTOSAR-konformen Motorsteuerungssoftware danken, sowie Simon Zilliken für seinen Beitrag zur Modellierung von AUTOSAR Systemen.

Den Mitstreitern in den Forschungsprojekten, an denen ich seitens ETAS mitwirken durfte, möchte ich ebenfalls danken. Allen voran gilt mein Dank Dr. Henrik Lönn für die spannenden fachlichen Diskussionen sowie Prof. Dr. Martin Törngren für die wertvollen Hinweise im Bereich Regelungstechnik.

Mein ganz besonderer Dank gilt meiner Familie, allen voran meinen Eltern Gerhard und Esther sowie meinem Bruder Philipp. Ohne ihre liebevolle Unterstützung wäre diese Arbeit nicht möglich gewesen. Vielen herzlichen Dank!

Zu guter Letzt gilt mein Dank all denjenigen namentlich hier Unbenannten die mich inhaltlich und moralisch unterstützt haben - ohne euch wäre diese Arbeit nicht möglich gewesen!

Contents

List of Figures	xiii
List of Tables	xxi
List of Algorithms	xxii
1 Introduction	1
2 Motivation, Goals and Thesis Outline	3
2.1 Motivation	3
2.2 Goals	6
2.3 Thesis Outline	7
3 Related Work	11
3.1 Control Engineering Domain	11
3.2 Telecommunication Domain	14
3.3 Aerospace Domain	19
3.4 Automotive Domain	20
4 Foundations	29
4.1 Control Theory	29
4.1.1 Physical Systems (Processes)	29
4.1.2 Signal Representation in Time and Value Domain	31
4.1.3 Mathematical Models for Dynamic Systems	32
4.1.4 Response Analysis of Dynamic Systems	36
4.1.5 Control Systems	38
4.1.6 Open and Closed-Loop Control Systems	38
4.1.7 Control Systems with Time-Delays	40
4.1.8 Closed-Loop Control System Architectures	43
4.2 Timing Models	45
4.2.1 Events and Actions	46
4.2.2 Ordering Mechanisms	49
4.2.3 A Formal Model for Real Clocks	54
4.3 AUTOSAR	57
4.3.1 Introduction	57
4.3.2 Standardized, Layered Software Architecture for Electronic Control Units	61

4.3.3	Development Methodology	62
4.3.4	Virtual Functional Bus View and System View	68
4.3.5	Concepts of the Software Component Technology	69
4.3.6	Communication Patterns	75
4.3.7	Virtual Functional Bus Tracing Mechanism	82
4.4	Summary and Overview of Work	86
4.4.1	Summary to Control Theory	86
4.4.2	Summary to Timing Models	87
4.4.3	Summary to AUTOSAR	87
4.4.4	Overview of Work	89
4.4.5	An Integrated Example	90
5	Specification of Timing Requirements based on Formal Path Specifications	93
5.1	Introduction	93
5.2	Events and Actions	94
5.2.1	Runnable Entity Events	96
5.2.2	Runtime Environment API Actions	97
5.2.3	Communication Events	106
5.2.4	Operating System Events	107
5.2.5	Overview on Event and Action Classes	109
5.3	Clocks	109
5.3.1	Planning Phase (Virtual Functional Bus View)	110
5.3.2	Realization Phase (System View)	113
5.4	Event Chains	113
5.4.1	Basic Concepts	115
5.4.2	Atomic Event Chains	117
5.4.3	Composite Event Chains	119
5.4.4	Multiple Instantiation of Components and Event Chains	124
5.4.5	Event Chains for Alternative Cases	126
5.5	Top-Level Event Chains and Path Specifications	127
5.6	Timing Requirements	129
5.6.1	End-to-End Delay Requirements	130
5.6.2	Interval Delay Requirements	131
5.6.3	Synchronization Requirements	133
5.7	Integration with Views, Development Methodology and Templates	135
5.7.1	Views	136
5.7.2	Development Methodology	137
5.7.3	Templates	139

6	Instrumentation of AUTOSAR Systems for Simulation and Monitoring	141
6.1	Introduction	141
6.2	Derivation of Flat Ordered Sets of Runtime Environment API Actions	143
6.3	Derivation of Concrete Path Specifications based on Flat Logical Path Specifications	146
6.3.1	Intra-Task Communication	147
6.3.2	Inter-Task Communication	150
6.3.3	Inter-ECU Communication	152
6.3.4	Concrete Path Specifications for the Example AUTOSAR Sys- tem	155
6.4	Generation of Software Instrumentations for Simulation and Monitoring	157
6.4.1	Event-Logging Mechanisms	157
6.4.2	Implementation of Virtual Function Bus Trace Hook Functions	162
6.4.3	Integration into AUTOSAR Development Methodology and ECU Software Build Process	168
7	Analysis of Event Traces from Monitoring and Simulation	171
7.1	Introduction	171
7.2	Measurement Of Temporal Distances Between Events And Actions .	173
7.3	Determination of Path Instances and Timing Properties	176
7.3.1	All Possible Path Instances between the Instances of two Event or Action Classes	181
7.3.2	Feasible Path Instances	182
7.3.3	Joining Path Instances	185
7.3.4	Feasible Path Instances for Path Specifications of Arbitrary Length	187
7.3.5	Shortest (feasible) Path Instances	189
7.3.6	Intersection Points of Path Specifications	191
7.3.7	Intersection Point Instances of Path Instances	194
7.3.8	Join and Fork Path Segments from Intersecting Path Instances	196
7.4	Timing Oscilloscope Diagrams	201
7.4.1	Reactive Real-Time Applications	202
7.4.2	Control Applications	203
7.5	Analysis of the Example AUTOSAR System	208
7.5.1	Description of AUTOSAR System and Base Configuration .	209
7.5.2	Configuration 1	212
7.5.3	Configuration 2	214
7.5.4	Configuration 3	216
7.5.5	Configuration 4	218
7.5.6	Configuration 5	220

8 Case Study: Engine Control Application	223
8.1 Introduction	223
8.2 Internal Combustion Engines and Tasks of an Engine Control Application	223
8.3 Overview of the Basic Functionalities	225
8.4 Air System	226
8.4.1 Structure	226
8.4.2 Signal Paths	227
8.4.3 Timing Requirements	227
8.5 AUTOSAR System	230
8.6 Specification of Timing Requirements based on Formal Path Specifications	231
8.7 Derivation of a Monitoring Instrumentation	238
8.8 Evaluation of the Timing Requirements	238
8.8.1 Path Delays	239
8.8.2 Input Interval Delays	240
8.8.3 Output Interval Delays	241
8.8.4 Input Synchronization Intervals	242
8.9 Summary and Conclusion	242
9 Summary, Extensibility and Transferability	245
9.1 Summary	245
9.2 Extensibility	249
9.3 Transferability	252
Appendix A AUTOSAR Meta-Model Excerpts	253
Appendix B Graphical Notation for AUTOSAR	261
Appendix C Description of AUTOSAR Runtime Environment API Functions and Macros	269
C.1 Sender/Receiver Communication	270
C.2 Inter-Runnable Communication	271
C.3 Client/Server Communication	272
Appendix D Amendments to the Virtual Function Bus Tracing Mechanism	273
D.1 Monitoring of Signal (Group) Transmission and Reception via COM Events	273
Appendix E Amendments to AUTOSAR Timing Model	277
E.1 Amendments to Definitions for Event Classes and Action Classes	277
E.1.1 COM Events for Transmission and Reception of System Signal Groups	277

E.2	Meta-Model Specifications for AUTOSAR-specific Event and Action Classes	278
E.2.1	RunnableEntity Events	278
E.2.2	Runtime Environment (RTE) API Actions	279
E.2.3	Communication (COM) Events	284
E.2.4	Operating System (OS) Events	285
E.3	Overview on AUTOSAR-specific Event and Action Classes	286
E.4	Meta-Model Specification for Hierarchical Event Chains	288
E.5	Meta-Model Specifications for Timing Requirements	289
E.5.1	Abstract Superclasses for Timing Requirements	289
E.5.2	Delay Requirements	289
E.5.3	Interval Delay Requirements	290
E.5.4	Synchronization Requirements	291
E.6	Amendments to Derivation of Concrete Path Specifications	294
E.6.1	Intra-Task Communication	294
E.6.2	Special Cases of Intra-Task Communication	295
E.6.3	Special Cases of Inter-Task Communication	297
E.6.4	Special Cases of Inter-ECU Communication	298
E.7	Amendments to Event Logging in Simulation and Monitoring-Based Approaches	301
E.8	Amendments to Path Analysis Algorithms	305
E.8.1	Feasible Backward Path Instances	305
E.8.2	All Feasible Backward Path Instances	306
E.8.3	Shortest Feasible Backward Path Instances	308
Appendix F	Amendments to Case Study	311
Bibliography		319

List of Figures

Related Work	11
3.1 Overview on relevant related work	12
3.2 Conceptual architecture of distributed monitoring systems (figure adapted from [48])	16
3.3 Layered software architecture for event-based analysis systems (figure adapted from [57])	17
Foundations	29
4.1 Physical system or process	29
4.2 Relation of input and output signals in linear and non-linear systems	30
4.3 Relation of input and output signals in time-invariant systems	31
4.4 The four possible signal representations in continuous and discrete time and value domains	32
4.5 Effect of a time delay of a continuous-time output signal on the discrete-time input signal	35
4.6 Unit step response of a second order system	37
4.7 Basic structure of open and closed loop control systems	39
4.8 Closed-loop control system with time delay in plant process	41
4.9 Closed-loop control system with time delay in sensor, controller and actuator processes	42
4.10 Structure of a MIMO closed-loop control system	44
4.11 Relationship between the concepts of Event Class, Event Instance, Action Class and Action Instance	47
4.12 Examples for working principles of scalar and vectorial logical clocks	52
4.13 Hierarchy of different clock types and their behavior	55
4.14 Dependencies between AUTOSAR templates	59
4.15 AUTOSAR layered ECU software architecture	62
4.16 Overview of AUTOSAR methodology - Planning phase	63
4.17 Overview of AUTOSAR methodology - Realization phase	64
4.18 Example logical software architecture	65
4.19 Example system topology	65
4.20 Mapping of logical software architecture to system topology: SWC-to-ECU and data mappings	66
4.21 Components, Ports and Interfaces, and Connectors	69

4.22	RunnableEntities, Declaration of Data Accesses and Service Provisions/Needs, RTEEvents	72
4.23	Examples for Sender/Receiver, Client/Server and Inter-Runnable communication	75
4.24	Classification of the different realizations of the communication patterns in concrete AUTOSAR implementations	81
4.25	Overview of the VFB Trace Events in the architecture of an AUTOSAR system	83
4.26	State transition diagrams for tasks in OSEK/VDX and AUTOSAR compliant RTOS	85
4.27	Overview of the work presented in this thesis towards a the Timing Model for AUTOSAR	89
4.28	Example closed-loop single-input-single-output (SISO) control application with timing requirements associated with the feedback path	91
4.29	Example AUTOSAR system for a simple single-input-single-output (SISO) control application	92
Specification of Timing Requirements based on Path Specifications		93
5.1	Meta-model specification for event and action classes in AUTOSAR	95
5.2	Example AtomicSoftwareComponentType with specification of RunnableEntityEvents	97
5.3	Example AtomicSoftwareComponentTypes with specification of RTE-APIActions for Sender/Receiver communication	99
5.4	Specification of InterRunnableVariable and concepts for specifying reading/writing data access of a RunnableEntity to it (excerpt from AUTOSAR meta-model [6])	100
5.5	Specification of InterRunnableVariable and changed concepts for specifying reading/writing access of a RunnableEntity to it	100
5.6	Example AtomicSoftwareComponentTypes with specification of RTE-APIActions for Interrunnable communication	101
5.7	Example AtomicSoftwareComponentType with a SynchronousServerCallAction	104
5.8	Example AtomicSoftwareComponentType with an AsynchronousServerCallStartAction and an AsynchronousServerCallReturnAction	104
5.9	Example AtomicSoftwareComponentType with an AsynchronousServerCallStartAction and an AsynchronousServerCallReturnAction belonging to different RunnableEntities	105
5.10	Example AtomicSoftwareComponentType with OperationEvents	106
5.11	State transition models for basic and extended tasks in OSEK/VDX and AUTOSAR compliant real-time operating systems	108
5.12	Planning Phase: Binding of event and action classes from the logical software architecture to the system-wide ideal clock	111
5.13	Planning Phase: Characterization of real-time clocks in the ECUs	112

5.14	Realization Phase: Binding of event and action classes from instances of atomic software components to real-time clocks in ECU instances	114
5.15	Description of order relations by means of AtomicEventChainTypes for Sender/Receiver communication	117
5.16	Description of order relations by means of an AtomicEventChainTypes for Interrunnable communication	118
5.17	Example AtomicEventChainType for a synchronous Client/Server communication	118
5.18	Two example AtomicEventChainTypes for asynchronous Client/Server communication (client side)	119
5.19	CompositeEventChainType for a CompositionType (with one hierarchy level)	120
5.20	CompositeEventChainType for a CompositionType (with two hierarchy levels)	121
5.21	CompositeEventChainType for a CompositionType (with mixed hierarchy levels)	122
5.22	CompositeEventChainType for a CompositionType with synchronous Client/Server communication (one hierarchy level)	123
5.23	CompositeEventChainType for a CompositionType with asynchronous Client/Server communication (one hierarchy level)	123
5.24	Specification of an AtomicEventChainType for multiply instantiable SensorActuatorSoftwareComponentType	124
5.25	CompositeEventChainType specifying the order of RTEAPIActions for ComponentPrototypes Sensor1 and Voter	125
5.26	CompositeEventChainType specifying the order of RTEAPIActions for ComponentPrototypes Sensor2 and Voter	125
5.27	Example for specification of distinct AtomicEventChainTypes for the InternalBehavior of an AtomicSoftwareComponentType	127
5.28	Example AUTOSAR system with a TopLevelEventChain that is a PathSpecification	128
5.29	Integration of timing requirements with AUTOSAR system description	130
5.30	Example for an AUTOSAR software architecture with a PathSpecification and an associated PathDelayRequirement	132
5.31	Example for an AUTOSAR software architecture with two CompositeEventChainTypes and an associated InputSynchronizationRequirement	135
5.32	Overview of AUTOSAR methodology with integrated timing model concepts	138
5.33	AUTOSAR Templates with extensions for AUTOSAR timing model	139
Specification-Based Instrumentation of AUTOSAR Systems for Simulation and Monitoring		141

6.1	Overview on specification-based instrumentation of AUTOSAR systems for simulation and monitoring	142
6.2	Logical software architecture of the AUTOSAR system for the control application example	145
6.3	Flat logical PathSpecification as directed graph for the example AUTOSAR system	145
6.4	Example logical Sender/Receiver communication	147
6.5	System Mapping and ECU configuration decisions leading to intra-ECU, intra-task communication of a logical Sender/Receiver communication	148
6.6	Intra-task Sender/Receiver communication: Explicit write and explicit read actions	149
6.7	Intra-task Sender/Receiver communication: Implicit write and implicit read actions	149
6.8	System Mapping and ECU configuration decisions leading to intra-ECU, inter-task communication of a logical Sender/Receiver communication	150
6.9	Inter-task Sender/Receiver communication: Explicit write and read actions	151
6.10	Inter-task Sender/Receiver communication: Implicit write and read actions	151
6.11	System Mapping and ECU configuration decisions leading to inter-ECU, inter-task communication of a logical Sender/Receiver communication	153
6.12	Inter-ECU Sender/Receiver communication: Explicit write and read actions	153
6.13	Inter-ECU Sender/Receiver communication: Implicit write and read actions	154
6.14	Example AUTOSAR system for the simple control application with relevant ECU configuration details (OS configuration)	155
6.15	Flat concrete PathSpecification for the example AUTOSAR system of the simple control application	156
6.16	Overview of AUTOSAR methodology with usage of timing model concepts for automatic generation of an RTE tracing instrumentation	168
Analysis of Event Traces from Monitoring and Simulation and Determination of Application-Specific Timing Properties	171	
7.1	Overview on determination of path instances for determination of timing properties and their visualization	173
7.2	Inner and outer distance between instances of event and action classes	174
7.3	Example for two action classes and the inner and outer distance between them	176

7.4	Example: Event trace scenario with three event classes and their instances and a given PathSpecification	177
7.5	Example: Event trace scenario with three event classes and their instances, a given PathSpecification and several feasible PathInstances	178
7.6	Example: Same event trace scenario of three event classes, their instances and identified relations between them	179
7.7	Event trace scenario with event classes A and B and their instances	182
7.8	Event trace scenario with computed set of all possible PathInstances between instances of event classes A and B	183
7.9	Event trace scenario with computed set of all feasible forward PathInstances between instances of event classes A and B	184
7.10	PathInstances between instances of event classes A and B, and B and C, respectively	186
7.11	Joined PathInstances between instances of event classes A and B and C	187
7.12	Event trace scenario with all feasible forward PathInstances for the PathSpecification $A \rightarrow B \rightarrow C$	188
7.13	Shortest feasible forward PathInstances for the PathSpecification $A \rightarrow B \rightarrow C$	190
7.14	Examples for overlapping PathSpecifications with IntersectionPoints	192
7.15	Example for overlapping PathSpecifications with IntersectionPoints that are not JoinPoints or ForkPoints	193
7.16	Example for two intersecting PathSpecifications	198
7.17	Event trace scenario of action instances with shortest feasible forward PathInstances	199
7.18	Event trace scenario of action instances with shortest feasible forward PathInstances, JoinPathSegments and ForkPathSegments	200
7.19	Event trace scenario of event instances with identified PathInstances and determined ReactionTimes	202
7.20	Timing Oscilloscope Diagram with ReactionTimeRequirement and ReactionTimes	203
7.21	Event trace scenario of event instances with identified PathInstances and determined PathDelays	204
7.22	Timing Oscilloscope Diagram with PathDelayRequirement and PathDelays	205
7.23	Event trace scenario of event instances with identified PathInstances and determined InputIntervalDelays	206
7.24	Timing Oscilloscope Diagram with InputIntervalDelayRequirement and InputIntervalDelays	206
7.25	Event trace scenario of event instances with identified PathInstances and determined InputSynchronizationIntervals	207
7.26	Timing Oscilloscope Diagram with InputSynchronizationRequirement and InputSynchronizationIntervals	208
7.27	Example AUTOSAR system of the SISO control application used for monitoring experiments	210

7.28	Order of RTEAPIActions described by the logical PathSpecification in figure 7.27	211
7.29	Order of AUTOSAR-specific Observable for configuration 1	213
7.30	Configuration 1: Timing Oscilloscope Diagrams	214
7.31	Order of AUTOSAR-specific Observable for configuration 2	215
7.32	Configuration 2: Timing Oscilloscope Diagrams	216
7.33	Configuration 3: Timing Oscilloscope Diagrams	217
7.34	Configuration 4: Timing Oscilloscope Diagram	218
7.35	Excerpt of the example AUTOSAR system of the MIMO control application used for monitoring experiments	219
7.36	Event trace scenario for clock drift effects	221
7.37	Configuration 5: Timing Oscilloscope Diagrams	222
Case Study: Engine Control Application		223
8.1	Schematic overview of engine control application with relevant input and output signals	224
8.2	Overview of the internal structure of the engine control application and its basic functionalities	225
8.3	Overview on the internal structure of the air system	227
8.4	Signal paths from input signals ThrottlePosition1/2 to output signal DesiredThrottlePosition and associated timing requirements	229
8.5	Signal paths segments from ThrottlePosition1 and ThrottlePosition2 to ThrottlePosition and associated timing requirements	230
8.6	Overview of the AUTOSAR system for the engine control application	233
8.7	Specification of timing requirements on the effective feedback path delay, sampling and actuation rates based on a formal path specification of the feedback path	234
8.8	Specification of input synchronization timing requirement for synchronized throttle sensor data acquisition	238
8.9	Flat logical and concrete PathSpecifications for the feedback path of the throttle control application	238
8.10	Timing Oscilloscope Diagrams for PathDelays	240
8.11	Timing Oscilloscope Diagrams for InputIntervalDelays	241
8.12	Timing Oscilloscope Diagrams for OutputIntervalDelays	242
8.13	Timing Oscilloscope Diagrams for InputSynchronizationIntervals	243
Summary, Extensibility and Transferability		245
9.1	Overview of work	246
9.2	Concepts of the Timing Model for AUTOSAR: Support of a compositional bottom-up approach for internal PathSpecifications	250
9.3	Additional concepts: Support of a decompositional, top-down approach for external PathSpecifications	251

9.4 External logical Path Specification vs. Internal logical Path Specification	251
AUTOSAR Meta-Model Excerpts	253
A.1 Meta-model excerpt for components	253
A.2 Meta-model excerpt for ports and interfaces	254
A.3 Meta-model excerpt for data types	255
A.4 Meta-model excerpt for connectors	256
A.5 Meta-model excerpt for internal behavior and RunnableEntity	257
A.6 Meta-model excerpt for RTEEvents	258
A.7 Meta-model excerpt for communication patterns	259
Amendments to the Virtual Function Bus Tracing Mechanism	273
D.1 API functions of RTE and COM involved in remote signal communication (primitive data)	274
D.2 API functions of RTE and COM involved in remote signal communication (complex data)	275
Amendments to AUTOSAR Timing Model	277
E.1 Meta-model specification of RunnableEntityEvents	278
E.2 Meta-model specification of abstract superclass RTEAPIAction	279
E.3 Meta-model specification of abstract superclasses SendDataAction and ReceiveDataAction	279
E.4 Meta-model specification for concrete RTEAPIActions or Sender/Receiver communication	280
E.5 Meta-model specification for the concrete RTEAPIActions of Inter-runnable communication	281
E.6 Meta-model specification of abstract superclasses ServerCallAction and OperationEvent for Client/Server communication	282
E.7 Meta-model specification for concrete RTEAPIActions on client side	282
E.8 Meta-model specification of concrete EventClasses for server side	283
E.9 Meta-model specification of COMSignalEvents for Inter-ECU communication	284
E.10 Meta-model specification of COMSignalGroupEvents for Inter-ECU communication	284
E.11 Meta-model specification of TaskEvents	285
E.12 Meta-model specification of EventEvents	285
E.13 Meta-model specification for hierarchical event chains	288
E.14 Meta-model specification for timing requirements	289
E.15 Meta-model specification for DelayRequirements	290
E.16 Meta-model specification for IntervalDelayRequirements	290
E.17 Meta-model specification for SynchronizationRequirements	291

E.18	Meta-model specification for InputSynchronizationRequirement	292
E.19	Meta-model specification for OutputSynchronizationRequirement	293
E.20	Example logical Interrunnable communication	294
E.21	Intra-Task Interrunnable Communication: Explicit write and explicit read actions	295
E.22	Intra-Task Interrunnable Communication: Implicit write and Implicit read actions	295
E.23	Intra-task Sender/Receiver communication: Implicit write and ex- plicit read actions	296
E.24	Intra-Task Sender/Receiver Communication: Explicit write and Im- plicit read actions	296
E.25	Inter-task Sender/Receiver communication: Implicit write and ex- plicit read actions	297
E.26	Inter-task Sender/Receiver communication: Explicit write and im- plicit read actions	298
E.27	Inter-ECU Sender/Receiver communication: Implicit write and ex- plicit read actions	299
E.28	Inter-ECU Sender/Receiver communication: Explicit write and im- plicit read actions	300
E.29	Event logging in simulation-based approaches	301
E.30	Event logging in monitoring-based approaches for AUTOSAR systems with a single ECU	302
E.31	Event logging in monitoring-based approaches for AUTOSAR systems with a multiple distributed ECUs and synchronized real-time clocks	303
E.32	Event logging in monitoring-based approaches for AUTOSAR systems with multiple distributed ECUs and a distributed monitoring system with synchronized clocks	304
E.33	Computed set of all feasible backward PathInstances between in- stances of event classes A and B	306
E.34	All feasible backward PathInstances for the PathSpecification A → B → C	307
E.35	Shortest feasible backward PathInstances for the PathSpecification A → B → C	308
F.1	Overview of components of an internal combustion engine and engine control application	311

List of Tables

Foundations	29
4.1 Overview of RTEEvents	74
4.2 Possible occurrences of Sender/Receiver communication pattern	78
4.3 Possible occurrences of Inter-Runnable communication pattern	79
4.4 Possible occurrences of Client/Server communication pattern	80
4.5 VFB Tracing COM hook functions for signal and signal group communication	85
4.6 Timing requirements for discrete-time, linear time-invariant control applications	86
Specification of Timing Requirements based on Path Specifications	93
5.1 Concrete RTEAPIActions for Sender/Receivr communication	98
5.2 Concrete RTEAPIActions for Interrunnable communication	101
5.3 Concrete RTEAPIActions for Client/Server communication (client side)	103
5.4 Availability of AUTOSAR-specific event and action classes under the specific AUTOSAR view	136
Specification-Based Instrumentation of AUTOSAR Systems for Simulation and Monitoring	141
6.1 Overview of generic event logging function	159
6.2 Overview of event logging functions provided by chronSim	160
6.3 Overview of event logging functions provided by RTA-TRACE	161
6.4 Overview on event logging functions for simulation and monitoring	162
Analysis of Event Traces from Monitoring and Simulation and Determination of Application-Specific Timing Properties	171
7.1 Configuration 1: Implicit Sender/Receiver communication, single-rate time-triggered activation, single OS-task	212
7.2 Configuration 2: Implicit Sender/Receiver communication, multi-rate triggering, multiple OS-tasks	214
7.3 Configuration 3: Explicit Sender/Receiver communication, multi-rate triggering, multiple OS-tasks	216

7.4 Configuration 5: Explicit Sender/Receiver communication, multiple OS-tasks, clock drift	220
Graphical Notation for AUTOSAR	261
B.1 ComponentTypes	261
B.2 CompositionTypes and ComponentPrototypes	261
B.3 RequiredPortPrototypes and ProvidedPortPrototypes of Sender/Receiver communication	262
B.4 RequiredPortPrototypes and ProvidedPortPrototypes of Client/Server communication	262
B.5 AssemblyConnectorPrototypes and DelegationConnectorPrototypes	263
B.6 InterRunnableVariables	263
B.7 Implicit Sender/Receiver communication	264
B.8 Explicit Sender/Receiver communication (Unqueued, i.e. data semantics)	264
B.9 Explicit Sender/Receiver communication (Queued, i.e. event semantics)	265
B.10 Synchronous and Asynchronous Client/Server communication (client side)	265
B.11 Implicit Inter-Runnable communication	266
B.12 Explicit Inter-Runnable communication	266
B.13 RTE Events	267
B.14 RTE Events (cont.)	268
Description of AUTOSAR RTE API Functions and Macros for the AUTOSAR Communication Patterns	269
C.1 RTE API functions for Sender/Receiver communication	270
C.2 RTE API functions for Inter-Runnable communication	271
C.3 RTE API functions for Client/Server communication	272
Amendments to AUTOSAR Timing Model	277
E.1 Overview of RunnableEntityEvents	286
E.2 Overview of OSEvents	286
E.3 Overview of COMEvents	286
E.4 Overview of RTEAPIActions	287
E.5 Translation of logical causal relations to concrete causal relations (Intra-Task Interrunnable communication)	294

List of Algorithms

6.1	Algorithm to derive a flat ordered set of RTEAPIActions from a logical PathSpecification	143
7.1	Algorithm to compute the inner distance between any two instances of event or action classes	175
7.2	Algorithm to compute the outer distance between any two instances of event or action classes	175
7.3	Algorithm to compute all possible PathInstances between the instances of two event classes	181
7.4	Algorithm to extract all feasible forward PathInstances from a set of possible PathInstances	183
7.5	Algorithm to join two sets of PathInstances	185
7.6	Algorithm to compute all feasible forward PathInstances for a Path-Specification	187
7.7	Algorithm to extract the shortest PathInstances from a set of forward PathInstances	189
7.8	Algorithm to compute the IntersectionPoints for two PathSpecifications	192
7.9	Algorithm to determine if an IntersectionPoint is a JoinPoint	193
7.10	Algorithm to determine if an IntersectionPoint is a ForkPoint	193
7.11	Algorithm to compute the JoinPoints and ForkPoints for two Path-Specifications	194
7.12	Algorithm to compute the IntersectionPointInstances of two sets of PathInstances	194
7.13	Algorithm to extract the JoinPointInstances from a set of Intersection-PointInstances for a given JoinPoint	195
7.14	Algorithm to extract the ForkPointInstances from a set of Intersection-PointInstances for a given ForkPoint	195
7.15	Algorithm to extract the JoinPathSegments to a given JoinPoint-Instance from a set of PathInstances	196
7.16	Algorithm to extract the JoinPathSegments to a given set of Join-PointInstance from a set of PathInstances	196
7.17	Algorithm to extract the ForkPathSegments to a given ForkPoint-Instance from a set of PathInstances	197
7.18	Algorithm to extract the ForkPathSegments to a given set of Fork-PointInstance from a set of PathInstances	197
E.1	Algorithm to extract all feasible backward PathInstances from a set of possible PathInstances	305

E.2	Algorithm to compute all feasible backward PathInstances for a Path-Specification	306
E.3	Algorithm to extract the shortest PathInstances from a set of backward PathInstances	308

1 Introduction

In the last two decades, embedded real-time systems have been the main driver for key innovations in the automotive industry. Today, embedded systems and computer controlled functions can be found in all kinds of motor vehicles, from passenger vehicles over motorcycles to heavy duty trucks. The steady growth of the number of functionalities has been driven by customer and legislative demands towards increased comfort, safety, energy efficiency and ecological sustainability.

In order to realize these functionalities, the electric/electronic (E/E) system of modern vehicles has evolved over the last two decades. Formerly, it has been a set of stand-alone, monolithic electronic control units (ECUs) for each individual functionality (e.g., separate engine control unit, transmission control unit). Today, it is a distributed embedded computer system with 60-80 computational nodes, interconnected by 3-5 in-vehicle networks.

To master the development complexity of electric/electronic systems, automotive-specific standards such as AUTOSAR provide the necessary concepts to successfully build systems. AUTOSAR conceptually integrates and extends the concepts of several of the previously existing automotive standards (e.g., OSEK/VDX, MSR-SW) towards a rigorous approach for automotive embedded systems development. The AUTOSAR specifications feature a standardized, layered ECU software architecture with standardized application programming interfaces (APIs) for the basic software modules which constitute the platform software. Furthermore, AUTOSAR provides a software component technology which allows the description of automotive applications in terms of componentized software entities. An AUTOSAR-specific development methodology describes how to build AUTOSAR compliant systems employing the former concepts.

Many of the functions realized in the electric/electronic system of a vehicle are control applications. They either assist the driver (e.g., adaptive cruise control application) or perform autonomous tasks completely hidden from the vehicle passengers (e.g., throttle control application as part of an engine management system). Control applications are inherently real-time applications as they have timing requirements on their execution in a computer system that guarantee their operation in synchrony with the physical environment and that they keep pace with its proceeding dynamics. For example, the timing requirements of control applications demand that the computation and communication delay between a data transformation from a sensor to an actuator is either negligibly small or constant. Other timing requirements demand that the input data acquisition by multiple sensors must be synchronized. Violations of these timing requirements can lead to deteriorated control performance

1 Introduction

or even loss of control. The latter often means that the physical process under control is influenced in such a way that damages can occur, e.g. the mechanical system that is to be controlled is destroyed. The fulfillment of the timing requirements is thus of utmost importance.

While it is possible to implement and realize control applications by means of the concepts of the AUTOSAR standard, the latter does not provide concepts to adequately specify the application-specific timing requirements and to evaluate their degree of fulfillment. For example, it is not possible to determine how long a data transformation from an input value acquired at a sensor to an output value effectuated by an actuator does take. The determination of the synchrony by which input values from multiple sensors are acquired has not even been considered in the control engineering domain so far. This, however, is required to properly account for the correct functional and temporal (=operational) behavior of control applications both during the development of such systems as AUTOSAR systems and during their operation in the vehicles.

This thesis introduces a Timing Model for AUTOSAR that provides all necessary concepts to specify application-specific timing requirements of control applications and to determine their corresponding timing properties such that the degree of fulfillment of the former can be evaluated. For this, a formal notion of time is introduced into AUTOSAR by means of a formal clock model that is adapted from existing related work. Concepts for the adequate description of application-specific timing requirements, especially those of control applications which are derived from control theory, are introduced. Special attention is given to the different communication patterns provided by AUTOSAR (Sender/Receiver-, Interrunnable- and Client/Server-communication), their specific variants (implicit vs. explicit, synchronous vs. asynchronous) and the AUTOSAR development methodology with its two phases (planning phase, realization phase). For the evaluation of the degree of fulfillment of the timing requirements, simulation- and monitoring-based approaches are employed. The instrumentation of the AUTOSAR system required for such approaches is directly derived from the description of signal paths that underly the specification of the timing requirements. For the interpretation of the timing properties, a new form of run-time oriented diagram is introduced, the so-called Timing Oscilloscope Diagram. It provides an adequate and familiar visualization of timing requirements and timing properties especially to control engineers who have to assess the performance of the control applications.

2 Motivation, Goals and Thesis Outline

2.1 Motivation

The electric/electronic system of modern automotive vehicles consists of a multitude of computer systems which are interconnected through diverse vehicle bus systems. This parallel and distributed computer system realizes several real-time applications amongst which there are diverse control applications. Control applications are inherently real-time applications as they directly interact with physical processes in the real-world environment they are embedded in and influence their behavior. In order to keep pace with the proceeding dynamics of real-world physical processes, several timing requirements must be constantly maintained during the execution of a discrete-time control application in a potentially distributed computer system such as an automotive electric/electronic system. These timing requirements are oriented on important signal paths within the control applications. These are the signal paths between sensors and actuators that sense and affect the same physical quantity. For an adequate description of the timing requirements, a precise specification of these signal paths, the so-called feedback paths, is required. Furthermore, for the determination of corresponding timing properties such as the feedback path delay, i.e. the time it takes for a signal to effectively travel along the feedback path, causal and temporal aspects of communication between parallel and distributed computational processes must be adequately considered.

AUTOSAR [5] is an standardization organization of the automotive industry with the goal to provide a de-facto open industry standard for automotive electric/electronic systems development. AUTOSAR provides a standardized, layered software architecture for electronic control units (ECUs) where the application software realizing automotive vehicle functions are separated from the platform software operating the ECU through a transparent middleware layer, the so-called Runtime Environment (RTE). The application software of an AUTOSAR system is described by means of an own software component technology provided by AUTOSAR. Furthermore, AUTOSAR provides a development methodology that describes how AUTOSAR systems are to be built. It distinguishes two major phases, a planning phase and a realization phase.

The concepts of the AUTOSAR standard pose several challenges with respect to the adequate description of application-specific timing requirements and the determination of corresponding timing properties such that the degree of fulfillment of the latter can be evaluated:

- Real-time application such as control applications are described by means of the software component technology provided by AUTOSAR. The AUTOSAR

2 Motivation, Goals and Thesis Outline

software component technology allows for a hierarchical description of software architectures such that the description of signal paths through the hierarchical software architecture is not trivial.

- The application software technology of AUTOSAR provides three distinct communication patterns (Sender/Receiver, Interrunnable and Client/Server communication) which each have further different specializations. The description of a signal path is consequently not trivial as different communication patterns can be employed which are also implemented differently in the source code of the AUTOSAR system.
- The AUTOSAR development methodology distinguishes between two major phases, a planning phase and a realization phase. In the planning phase, only the logical software architecture is specified without considering how it is finally realized as a potentially distributed computer system. This poses challenges for the description of signal paths to which timing requirements can refer in the planning phase and which are consistent with the signal paths that are ultimately present technical system in the realization phase and based on which timing properties need to be determined.

These challenges have so far hindered the development of adequate concepts for a Timing Model for AUTOSAR.

For the evaluation of the degree of fulfillment of timing requirements of real-time applications realized in parallel and distributed computer systems, there are three different conceptual approaches. These are either based formal analysis, simulation or monitoring.

Formal analysis-based approaches target the determination of safe best-case and worst-case figures for certain properties of a real-time computer system. Example formal analysis approaches are best-case and worst-case execution time analysis (BCET/WCET) that targets the determination of boundaries on the execution time of a sequential piece of code executed on a single-node computer system. Such analysis is also referred to as code-level analysis. System-level timing analysis takes effects such as blocking and preemption induced by the employed scheduling strategies on the computational nodes and bus arbitration policies for network communication into account. While originally being developed for single resources such as a single computational node (Liu and Layland [52]) or communication bus system (Tindell and Burns [82]), the system-level timing analysis have also been enabled to parallel and distributed computer systems in various specific approaches (Holistic Scheduling Analysis by Tindell and Clark [83], SymTA/S approach by Richter et al. [41, 70], Modular Performance Analysis with Real-Time Calculus (MPA-RTC) by Thiele et al. [18]). Although these approaches are capable of determining safe bounds on timing properties, these timing properties only partially match the timing requirements of control applications.

Simulation and monitoring-based approaches are very akin as they allow a performance analysis of parallel and distributed computer systems during their runtime

(Klar et al. [48]). The basis for both approaches is an abstraction from the detailed execution behavior of the parallel and distributed computer system. The behavioral abstraction is to only observe places where interesting state changes and actions take place which give rise to the performance of the real-time applications.

Simulation-based approaches have the advantage that no real hardware is required to conduct experiments for performance analysis. This means that performance evaluations with the software of the real-time applications can be performed at an earlier stage of the development even when the target hardware of the parallel and distributed computer system is not yet available. Disadvantages are that the results obtained from simulation experiments cannot be fully trusted as might be faults in the underlying assumptions (e.g., with respect to the assumed execution times of computational processes). However, simulation-based performance analysis approaches help to identify potential problems early in the development process and consequently reduces costs that otherwise incur due to late changes.

Monitoring-based approaches allow to evaluate the performance of the target real-time computer systems composed both of hardware and software. The advantage is that high confidence can be put in the results obtained from monitoring experiments as they stem from the target system that is built from both hardware and software. Disadvantages are that for parallel and distributed real-time computer systems, complex monitoring systems with synchronized measurement clocks and monitoring units are required. The critics that the instrumentation that is required for software-based monitoring approaches influences the actual dynamic behavior of the computer system compared to the uninstrumented system can be debilitated by accepting the instrumentation as part of the final system¹.

The combination of simulation- and monitoring-based performance analysis approaches covers the complete development cycle of parallel and distributed computer systems. For ultimate confidence in the timing properties of the system, formal analysis can be employed for the final built system. For this, however, the system must be designed and configured in such a way that the formal analysis approaches are applicable.

The fact that by means of simulation- and monitoring-based performance analysis approaches the dynamic behavior of a parallel and distributed computer system with its real-time applications can be evaluated suits well to the also dynamic nature of control applications. The performance of control applications in the value domain is also evaluated through dynamic methods, e.g. by means of response time analysis methods. Simulation- and monitoring-based performance analysis approaches are thus a suitable complement for the time domain.

Due to the latter, in this thesis, simulation- and monitoring-based approaches are considered rather than static analysis approaches as these dynamic, runtime-oriented approaches suit better to the class of real-time applications, control applications, considered in this thesis.

¹This is similar to software required for calibrating automotive software which is also accepted as part of the final system.

2.2 Goals

The higher goal of the work presented in this thesis is to develop the necessary concepts to enable the precise specification of application-specific timing requirements, especially those of control applications, and to provide a tailored means for the determination of the corresponding timing properties such that the degree of fulfillment of the latter can be evaluated. To achieve this higher goal, several aspects must be considered with intermediate goals to be achieved:

Precise specification of timing requirements based on signal path specifications

For the precise description of timing requirements of control applications, concepts are required to specify signal paths that denote feedback paths of control applications in an AUTOSAR system. For this, an adequate behavioral abstraction of the real-time applications realized in an AUTOSAR system must be found.

Integration of specification of timing requirements and performance evaluation

For the evaluation of the degree of fulfillment of timing requirements it is important that the efforts required to conduct simulation or monitoring experiments is minimized. The goal is to specify the signal paths to which the timing requirements refer in such a way such that the necessary instrumentation can be automatically derived.

Integration of performance evaluation in development methodology

In order to evaluate the degree of fulfillment of timing requirements in parallel to the existing development phases of automotive real-time applications and systems (simulation, prototyping, target implementation), the performance evaluation by means of simulation- and monitoring-based approaches needs to be integrated with the AUTOSAR development methodology.

Provision of adequate and easy to interpret visualizations for timing properties

During simulation and monitoring experiments, large amounts of event trace data is generally collected. The interpretation of this data requires adequate and easy to interpret visualizations in the form of application-specific diagrams. As the existing diagrams either provide too much detail (e.g., Gantt diagrams) or do not provide an adequate for visualization for the interested group of specialists, especially control engineers, a new form of diagram needs to be developed.

Accounting for specifics of AUTOSAR software component technology

In order to precisely specify signal paths for real-time applications being realized in an AUTOSAR system, it is necessary to develop concepts that seamlessly integrate with the existing AUTOSAR concepts for describing hierarchical software architectures by means of software components.

Accounting for specifics of AUTOSAR communication patterns

AUTOSAR provides a set of three distinct communication patters which each have further

specializations. The communication patterns are implemented differently in the source code of an AUTOSAR system, e.g. in order to automatically guarantee data consistency when multiple computational processes intend to read and write the same data at the same time. In order to determine valid timing properties, it is necessary to adequately account for issues related to the distinct communication patterns.

Accounting for intra-task, inter-task and inter-ECU communication Real-time applications being specified in the software architecture of an AUTOSAR system can be realized either as single-ECU systems or as distributed systems. Due to this and due to further configuration decisions for how the single computational processes deployed on an ECU are to be executed, the cases of intra-task, inter-task and inter-ECU communication need to be distinguished when determining timing properties.

2.3 Thesis Outline

At first, an overview on relevant related work originating from different domains and engineering disciplines is given (chapter 3). The different domains are the telecommunications domain, aerospace domain and automotive domain where real-time applications are realized as potentially parallel and distributed embedded computer systems. Furthermore, as the timing requirements originate from the control engineering discipline, related work that addresses the description of timing requirements and the determination of timing properties originating from this discipline is also discussed.

Then, the foundations of the three distinct fields that underly the contributions of this thesis are introduced (chapter 4). These are the foundations of control theory from where the timing requirements that can be formulated for discrete-time control applications originate from. The timing requirements are subsequently derived, including explanations for how they need to be conceptually expressed. Furthermore, the foundations of timing models are introduced as these are important for the development of a Timing Model for AUTOSAR. This comprises discussions about events and actions as a suitable means for behavioral abstractions of real-time applications realized in parallel and distributed computer systems and the two important order relations that are defined on sets of events, i.e. the causal and the temporal order relation. In order to make safe statements about causal and temporal relations among instances of events and consequently to determine meaningful timing properties, a formal notion of time is required. For this, the concepts of a formal model for the precise characterization of real clocks that exists from related work is described. Finally, the foundations of AUTOSAR are introduced as these form the basis for how automotive real-time applications such as control applications are developed and realized for modern automotive vehicles. This includes descriptions and discussions of all relevant aspects such as the standardized, layered ECU software architecture. Furthermore, the two-phased development methodology of AUTOSAR is described

2 Motivation, Goals and Thesis Outline

according to which AUTOSAR systems are being developed, as well as the concepts of the AUTOSAR software component technology by which any applications, including control applications, are being specified and realized in an AUTOSAR system. As AUTOSAR offers three conceptually distinct communication patterns which all have further different specializations, these are described in detail. Finally, the so-called Virtual Function Bus Tracing Mechanism provided by AUTOSAR is described which forms the basis for simulation- or monitoring-based determination of application-specific timing properties introduced in this thesis. The chapter also provides an overview of the work presented in this thesis, i.e. the concepts that constitute the Timing Model for AUTOSAR. Furthermore, an integrated example for a control application that is being realized as an AUTOSAR system is described which is used as an example throughout the thesis when the details of the single concepts are introduced.

The concepts that constitute the Timing Model for AUTOSAR introduced in this thesis are then presented in chapter 5, 6 and 7.

At first, the necessary concepts for the adequate specification of application-specific timing requirements are introduced and seamlessly integrated with the existing AUTOSAR concepts (chapter 5). This comprises definitions for what kinds of events and actions can be observed in an AUTOSAR system which are relevant for a behavioral abstraction and which can be used to describe signal paths between sensors and actuators. Furthermore, a formal notion of time is introduced through the introduction of the concepts of a formal model to characterize the behavior of real-time clocks in an AUTOSAR system. Then, a concept is introduced that allows to specify signal paths in the hierarchical software architecture of an AUTOSAR system. These path specifications can then be associated with timing requirements that resemble the timing requirements from discrete-time control theory.

For the determination of timing properties by means of simulation- or monitoring-based approaches it is required to obtain event trace data from an AUTOSAR system. For this, an instrumentation of the source code of the AUTOSAR system is required. The derivation of such an instrumentation from the specifications of the signal paths to which the timing requirements refer is described in chapter 6. From a technical perspective, it is outlined how the instrumentation integrates with the existing AUTOSAR-compliant ECU software architecture. From a methodological perspective, it is described how the process of deriving an instrumentation integrates with the AUTOSAR methodology. The obtained instrumentation enables the recording of event trace data within simulation or monitoring experiments.

For the determination of meaningful timing properties, algorithms are introduced that allow the determination of instances of effective signal paths (chapter 7). The algorithms target the identification of effective causal and temporal relations between instances of events that constitute a signal path. This allows the determination of timing properties, for example the feedback path delay, i.e. the latency of a data transformation of an input signal acquired at a sensor to an output signal effectuated by an actuator. Furthermore, a new type of runtime-oriented diagram is introduced, the Timing Oscilloscope Diagram. It shows the development of the determined

2.3 Thesis Outline

timing properties over time and allows a direct evaluation of the degree of fulfillment of the timing requirements.

After the concepts of the Timing Model for AUTOSAR are introduced, they are applied to a case study of an AUTOSAR-compliant engine control application (chapter 8). As the complete engine control application contains several functionalities for which application-specific timing requirements exist, in this thesis, only an excerpt, the air system, is discussed. It is shown that the concepts developed in this thesis for a Timing Model for AUTOSAR are applicable also to complex real-world automotive systems.

The thesis closes with a summary on the presented work and an outlook on its extensibility with further concepts and transferability to other domains (chapter 9).

3 Related Work

In this chapter, relevant related work from different domains of real-time computer systems (telecommunication domain, aerospace domain, automotive domain) and the control engineering domain is discussed. Figure 3.1 shows an overview on the considered related work as well as the projects it originates from on a timeline. The projects are either specific targeted research projects by individual researchers, groups or companies, or public funded projects and standardization initiatives by industrial and academic consortia. In the following, the scope of these projects and their relations is also described.

3.1 Control Engineering Domain

Relevant research in the control engineering domain that deals with timing aspects of control applications has been performed more or less independent from the work conducted in the other domains. It is, however, well known in the control engineering domain that there can be negative influences on the control performance due to effects that can occur in a potentially distributed real-time computer systems [85]. Such effects can blocking and preemptions of tasks and network messages, depending on the chosen scheduling strategy and their configuration. The performance-degrading effects can be non-zero constant or even time-varying delays that are effective on the feedback path, sampling jitter at the sensor(s) or actuation jitter at the actuator(s) as well as unsynchronized sampling or actuation actions if multiple sensor or actuator devices are employed. In the following, relevant related work from the control engineering domain is discussed that take timing aspects both from the control engineering perspective as well as the real-time computer systems perspective into consideration.

Törngren's Work

Based on the fundamentals of input-sampled discrete-time control theory, Törngren [84] has firstly researched the modeling and design of discrete-time control applications [2] and their realization as distributed real-time computer systems and secondly effects induced by the latter on the control performance.

Törngren defined a minimal yet sufficient framework of concepts for the modeling of the structure of control applications and the specification of their triggering behavior. The modeling concepts are based on the notion of *logical functions* that can be hierarchically aggregated. Besides that, Törngren also described what kinds of application-specific timing requirements need to be considered for control applications and how these could be represented as annotations to control application

3 Related Work

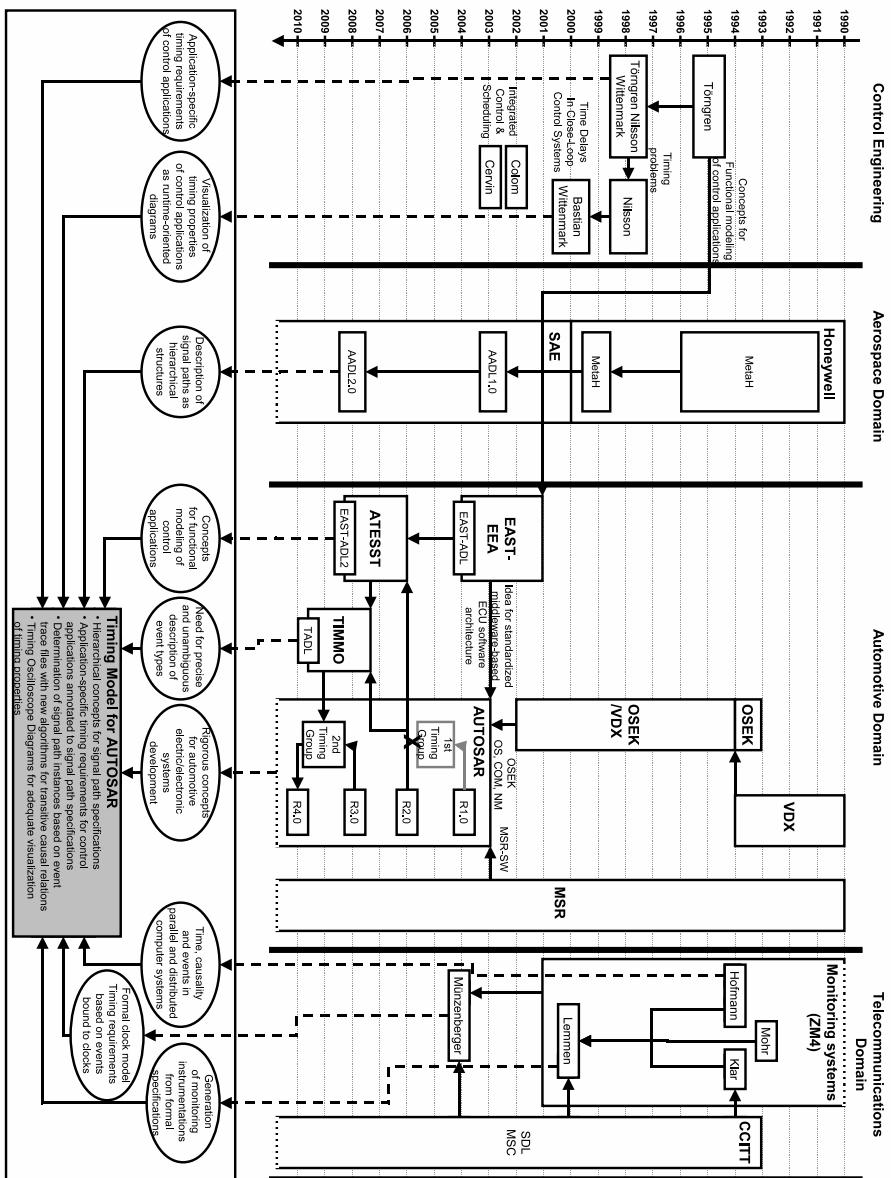


Figure 3.1: Overview on relevant related work

models described by means of the framework. The timing requirements described by Törngren, i.e.

- negligibly small or constant feedback path delay
- equidistant sampling and actuation actions
- synchronization between related sampling or actuation actions if multiple sensors or actuators are employed

are the same that we derive from discrete-time control theory in section 4.1.

The link between control application design and real-time computer system implementation is the assumption that logical functions are realized by real-time tasks that are scheduled under fixed priority preemptive scheduling strategy. Törngren's approach for the determination of timing properties is then a formula-based approach that takes specific scheduling-induced effects (blocking, preemption) of the real-time computer system into account. For an evaluation of the degree of fulfillment of the timing requirements in AUTOSAR systems, the approach, however, cannot be directly applied as Törngren's framework and the concepts of AUTOSAR are not directly compatible.

As an important observation, Törngren notes that there is an obvious gap between the control engineering and the real-time systems disciplines as discrete-time control theory and the real-time computer science have evolved separately over the years. A good overview of the framework by Törngren can be found in [85].

The concept of describing the structure and triggering behavior of control applications by means of logical functions has later been adapted in the EAST-ADL (see related work in the automotive domain).

Work by Nilsson, Bastian and Wittenmark

In [89], Nilsson, Wittenmark and Törngren describe timing problems of real-time control applications. Based on that, Nilsson [60] has researched the effects of time delays in networked control systems, i.e. control systems that are realized as a distributed embedded real-time computer system.

Based on Nilsson's work, Bastian [14] has analyzed time delays in control loops. Control applications that can be analyzed by Bastian's approach must fit into a specific layered structure, meaning that not all possible kinds of control applications can be analyzed.

For the determination of timing properties such as the feedback path delay, Bastian introduces a formula-based approach where probability distributions for the time delays are derived. Histograms are employed to show the results of this statistical analysis. The approach assumes that all parts of a control application (i.e., sensor, controller and actuator) are executed periodically and communicate in either a synchronous or asynchronous way, depending on the relation of their triggering rates.

3 Related Work

Bastian derives timing properties also from simulations that implicitly assume a global, ideal clock. The results are visualized as runtime-oriented diagram — also referred to as time histories of delays in [88] — where the development of the determined path delays are shown over time. The latter kind of diagram is akin to our Timing Oscilloscope Diagrams. Bastian and Wittenmark, however, only display path delays over time whereas we employ Timing Oscilloscope Diagrams also for the visualization of effective sampling and actuation rates as well as the synchronization between input data acquisitions by multiple sensors or output data effectuation by multiple actuators. Furthermore, it is not described how these diagrams are constructed.

Other Work

In the control engineering domain, further work has then been conducted in another direction. Cervin [17] proposes the approach of feedback-scheduling where the scheduler of a real-time computer system is perceived as a process that can be dynamically controlled. The approach requires that important scheduling parameters such as the period of a task or its priority can be adjusted online, i.e. during runtime, in order to compensate for overload situations or degraded performance of the control applications executed by the real-time computer system. These parameters, however, cannot be adjusted in practical industrial OSEK- or AUTOSAR-based systems as these systems employ a static scheduling strategy (fixed priority preemptive) where the parameters are fixed before runtime. The work of Colom [19] goes into a similar direction.

3.2 Telecommunication Domain

Since the 1970's, much research has been performed in the area of performance analysis of parallel and distributed computer systems. Parallel computer systems are computer systems that have more than one processing unit in a computational node. Distributed computer systems consist of multiple computational nodes that are interconnected by a network. Parallel and distributed computer systems have been developed for applications where the computation power of computer systems with a single processor was not sufficient, e.g. computationally intensive problems such weather forecasting, or where the application required equipment that is spatially distributed, e.g. the terminal stations of a banking system.

In the 1980s, the telecommunications domain was the first domain to employ parallel and distributed systems in commercial products subject to mass-production (e.g., digital telephone systems). In the automotive domain, the vehicles' electric/- electronic systems have evolved to a complex distributed computer system since the

early 1990s. Recently, parallel computer systems also found their way into the electric/electronic system in the form of dual-core and later multi-core electronic control units¹.

Comprehensive work towards the performance analysis of parallel and distributed computer systems has been conducted since the mid 1970s to the mid 1990s at the Friedrich-Alexander University in Erlangen-Nürnberg. In the following, performance analysis of parallel and distributed computer systems is discussed based on their research that later mainly addressed the telecommunication domain.

Monitoring-Based Performance Analysis of Parallel and Distributed Computer Systems

A widely employed approach to evaluate the performance of a parallel and distributed computer system is to analyze its dynamic runtime-behavior with the help of monitoring or simulation. In contrast to static analysis approaches that are based on abstracted data (e.g., best-case and worst-case execution times as in scheduling analysis approaches), monitoring-based approaches allow to observe the actually appearing behavior in a system and its timing. This is useful not only for debugging-purposes, but also for the exact determination of timing properties. This allows the evaluation of the degree of fulfillment of timing requirements at each point in time during the runtime of the system.

The foundations for performance analysis of parallel and distributed computer systems are based on careful considerations of the causal and temporal ordering of events that can be monitored in such systems. It is important that the events are ordered correctly even if they are monitored in different parts of the distributed system such that safe conclusions can be drawn between causes and effects.

In order to monitor the dynamic behavior of a parallel and distributed computer system, a monitoring system consisting of specific technical equipment and a software for the analysis of the monitored events is required.

Figure 3.2 shows a conceptual architecture of monitoring systems for parallel and distributed computer systems.

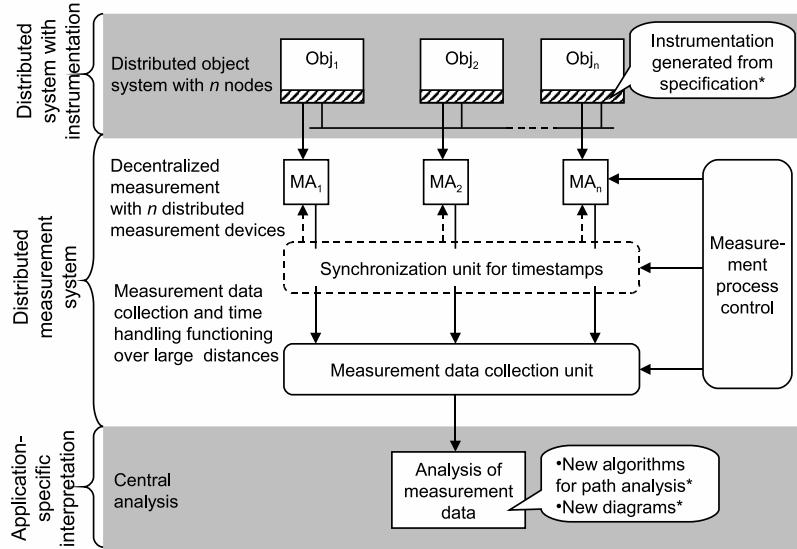
The main parts of such a system are:

- A parallel, distributed monitoring system that seamlessly integrates with the parallel and distributed computer system that is to be monitored, the so-called object system. For this purpose, the “Zählmonitor” (ZM)-termed monitoring system was developed at the University of Erlangen-Nürnberg. ZM4 is the latest and most universal² version in a series of developments.
- A flexible analysis software by which the event traces from the monitoring system can be analyzed and where the results are adequately visualized. For this purpose, the analysis software package SIMPLE [58] was developed.

¹In release 4.0 of the AUTOSAR standard that was published in early 2010, the concepts of a multi-core real-time operating system are standardized.

²Universal in terms of supported hardware and software interfaces to different kinds of object systems.

3 Related Work



* Places of own contributions for control applications and AUTOSAR.

Figure 3.2: Conceptual architecture of distributed monitoring systems (figure adapted from [48])

Such a monitoring system can in principle also be used for object systems such as AUTOSAR-compliant electric/electronic vehicle systems.

In our own work, we focus on

- the generation of software instrumentations that can be used for performance analysis in monitoring and real-time simulations of AUTOSAR systems. The instrumentation forms part of the monitoring system but integrates with the object system.
- the analysis of the event trace files obtained through monitoring or simulation and the application-specific interpretation of analysis results towards control application timing.

The parts highlighted in grey in figure 3.2 are the fields where our work provides conceptual contributions.

Contributions by Mohr

Based on the distributed monitor system ZM4, Mohr [57] has developed a flexible software for the analysis of event trace files called SIMPLE [58]. Mohr describes the principal architecture of such an analysis software based on a layered software

architecture (see figure 3.3). The lowest layer is the monitor layer where events are obtained from the object system and stored in event trace files. Mohr describes how event records and event trace files should be constructed such that it can be unambiguously identified when (time) and where (location) a specific event occurred. In order to obtain a single consistent event trace file from event trace files from the monitoring of a distributed computer system, Mohr describes how event trace files obtained from multiple computational nodes can be merged. This is the task of the data access layer. The selection layer offers functionality to navigate within a event trace file. The tool support layer provides general utility functions such as statistical functions. The tool layer implements the different ways how event traces can be analyzed. This includes event trace validation, statistical analyses, and visualizations of the runtime behavior of the system. SIMPLE offers different kinds of statistical diagrams (e.g., histograms) and runtime-oriented diagrams (e.g., state-time diagrams, Gantt diagrams) to visualize analysis results.

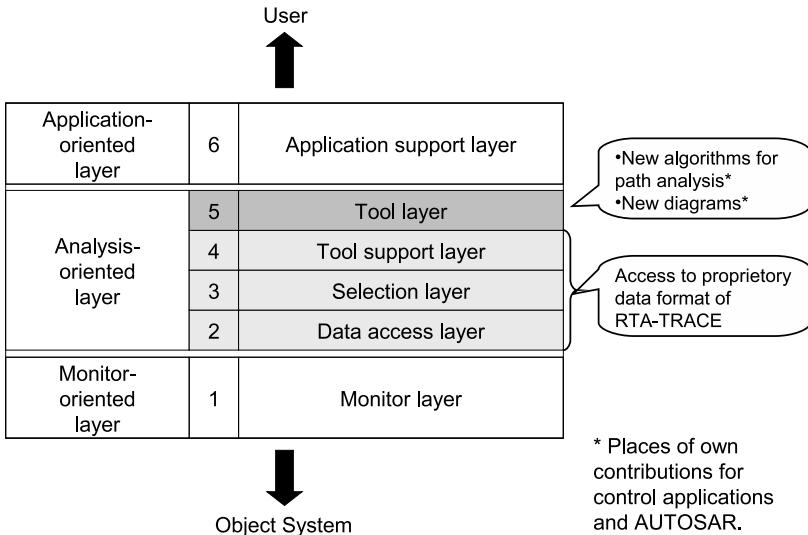


Figure 3.3: Layered software architecture for event-based analysis systems (figure adapted from [57])

The focus of our work is mostly on the tool layer where we provide new algorithms to analyze chains of cause and effect and new forms of runtime-oriented diagrams (Timing Oscilloscope Diagrams) to adequately visualize the results. The latter are comparable to the runtime-oriented diagrams provided by SIMPLE, however, they have been tailored to show the timing properties of control applications rather than reactive real-time applications as in the case of SIMPLE.

3 Related Work

Contributions by Lemmen

Lemmen [50] developed concepts to extend the monitoring-based performance analysis as described by Klar et al. [48] through a specification-driven concept that allows to obtain an instrumentation for the relevant events and actions of an object system based on a formal specification.

In the telecommunication domain, the Specification and Description Language (SDL) [15] is used to specify the structure and execution behavior of systems under development. SDL employs processes that communicate via messages. Message Sequence Charts (MSC) [16] are used in combination with SDL to describe the sequence of messages exchanged between system components as well as the internal states of relevant processes based on a logical time axis. MSCs specify the causal relations between messages and processes.

Lemmen's concepts complement the functional specification means provided by SDL/MSC such that the relevant events and actions that need to be monitored for an adequate behavioral abstraction of the object system can be automatically obtained.

As telecommunication systems modeled by means of SDL/MSC are inherently reactive real-time computer systems (parallel and distributed), work towards the formal description of timing requirements has only been conducted for this class of real-time applications. Timing requirements of control applications have not been considered.

Contributions by Münzenberger

Based on the foundations of event-based monitoring of parallel and distributed computer systems, i.e. events and the causal and temporal ordering principles of these, Münzenberger et al. [56] introduce a formal clock model that allows an exact characterization of different types of clocks (ideal clock, real clocks, discrete and derived counters). The formal clock model is the basis for the simulation of the timing behavior of parallel and distributed computer systems and consequently for the possibility to quickly and efficiently conduct performance evaluations in the planning phase and parallel to the development phase of such systems. The formal clock model is introduced in more detail in section 4.2.3 and used as a basis for our timing model for AUTOSAR.

Münzenberger [69] furthermore uses the formal clock model as a basis to specify real-time requirements based on events. Events are bound to clocks that are formally characterized such that the readings of the clock (i.e., the time-stamps of the events) become comparable, enabling the determination of timing properties. Furthermore, effects such as clock drift can be analyzed.

The concepts introduced by Münzenberger for the specification of timing requirements assumes that events of a *releasing event class* are related to events of a *demanded event class* through a logical condition and associated with a time bound. The concept is suited for reactive real-time applications such as those found in the telecommunications domain. Münzenberger has integrated his concept of formally

specified clocks and the specification of timing requirements based on events bound to clocks with SDL. While the concept allows the specification of timing requirements for reactive real-time applications, it is not directly suited for control applications as with the two event classes alone that are used in a timing requirement specification (releasing events, demanded events), signal paths cannot be adequately specified. For this, transitive causal relations between events need to be considered which are not supported.

3.3 Aerospace Domain

Within the Aerospace domain, there is one notable stream of research and standardization in the field of distributed embedded real-time computer systems that also deals with timing aspects to some extent: These are MetaH and AADL.

MetaH by Honeywell

MetaH is a language and toolset for modeling the software- and hardware-related artifacts of an embedded real-time application, and for the integration of both into a complete system. The concepts were originally developed by Honeywell Technology in the late 1980's. The language and toolset has been developed within a Domain-Specific Software Architecture program funded by DARPA between 1991 and 1996. In a follow-up project between 1997 and 1999 in the Evolutionary Design of Complex Systems program, the MetaH toolset was extended.

At the time of the development of MetaH, available commercial tools were primarily tailored for the telecommunications domain [76]. MetaH was developed to target the Aerospace domain where it also had its widest applications, mainly in demonstrator projects, but not in commercial products. Since the beginning of the 2000's, MetaH has first been renamed to the Avionics Architecture & Design Language (AADL) which has then been renamed to Architecture Analysis & Design Language (AADL). The latter is standardized and further developed by the Society of Automotive Engineers (SAE). A technical and historical overview on MetaH and the transition to AADL can be found in [76].

Architecture Analysis and Design Language by Society of Automotive Engineers

The Architecture Analysis and Design Language (AADL, [74], [29]) is a domain-specific language for the description of software-based embedded real-time applications, originally developed for the aerospace domain, but in principle also applicable to the automotive domain. It is the successor of MetaH and standardized in the Society of Automotive Engineers (SAE) standard AS5506 [75].

AADL provides concepts for modeling the software- and hardware-related artifacts of an embedded real-time application, and for the integration of both into a complete system. Systems described by means of AADL can be analyzed with

3 Related Work

respect to different non-functional properties such as performance and timing, but also safety, security and energy consumption. Performance evaluations based on simulation and/or monitoring are, however, not addressed by AADL.

AADL models are described by means of *components*, where a component belongs to one of three categories: application software, execution platform, or composite.

A specific concept of AADL is the concept for modeling so-called *flows*. A flow is an externally observable information path in a system. In application software components, flows are described by means of *flow sources*, *flow paths* and *flow sinks*. Flow paths can be annotated with timing properties to express the time it takes for data to travel along the path.

In [30], Feiler and Hansson present a flow latency analysis framework for AADL that allows the determination of “the end-to-end latency and age of signal stream data as well as their jitter.” The flow latency analysis framework targets control applications as these have timing requirements firstly on the latency between correlated sampling and actuation actions between sensors and actuators and secondly on the maintenance of constant sampling and actuation rates.

While the flow analysis framework addresses the right application-specific timing requirements for discrete-time control applications, it fails to provide a suitable means for the exact determination of the corresponding timing properties: firstly, only best-case and worst-case analyses can be conducted through which the application-specific timing requirements of control applications are not adequately addressed, and secondly only a limited set of AADL systems can be analyzed as the formula-based approach does not consider all modeling options that AADL provides. Furthermore, the calculations performed by the flow latency analysis plug-in of OSATE, the SEI Open Source AADL Tool Environment, which provides an implementation of the flow analysis framework are hardly retraceable.

Similar to AUTOSAR and EAST-ADL, AADL employs a component-based approach that allows the structuring of an application into components that can be hierarchically aggregated. The concept of flow paths that is included in AADL is also hierarchical as flow paths through a component can be composed from other flow paths of contained components. We provide a similar concept for AUTOSAR for the formal description of signal paths. This allows an adequate description of application-specific timing requirements, especially those of control applications.

3.4 Automotive Domain

In the automotive domain, due to the high pressure to reduce costs, vehicle manufacturers (OEMs) and their Tier-1 suppliers tightly collaborate. Much work is performed in joint initiatives such as public funded projects for applied research or standardization initiatives to agree on common concepts. The joint initiatives are conducted by consortia of both industrial partners (OEMs, Tier-1 suppliers, tool vendors), research institutes and universities. In the following, the scopes and relations of several of these initiatives that are relevant for our work are described.

EAST-EEA (EAST-ADL)

In the early 2000's, the automotive industry was facing the problem of growing complexity in automotive electric/electronic systems development. Software was in general specifically developed for single-purpose, dedicated electronic control units (ECUs). Application software realizing vehicle functions (e.g., control applications) that were in principal independent of a specific vehicle could neither be reused in other projects nor easily be migrated to another ECU. Reasons for this are microcontroller-specific implementations as well as a tight coupling of the application software realizing vehicle functions and the platform software of the ECU on which the application software is executed. In order to manage the growing development complexity, new concepts tailored for the automotive domain were required. A consortium of several industrial companies (OEMs, Tier-1's, tool vendors) as well as research institutes and universities was established, and a joint public funded research project called EAST-EEA [45] was conducted from 2001 to 2004.

At its center, the EAST-EEA project defined an *ontology of system models*, i.e. a set of system models that built on each other, each representing the relevant aspects for the development of electronic vehicle features and functions of a complete vehicle at a different level of abstraction. A system model on a certain level of abstraction is interpreted as the system being represented at a specific stage of a development. The ontology of system models comprised five abstraction levels: vehicle feature level, analysis level, design level, implementation level and operational level. The architectures on the analysis and design level represent the functionalities of a vehicle in different granularities by means of logical functions. The architecture on the implementation level represents the functionalities in terms of software and hardware entities. The operational level contains the actual artifacts representing the electric/electronic vehicle system, i.e. the hardware platforms and the code that is running on them.

The modeling concepts for describing the different architectures are described in the EAST-ADL [46], the architecture description language defined by the EAST-EEA project. Influences from Törngren's work can be found in the modeling concepts for function architectures [46]. Logical functions are used to specify the structure and triggering behavior of the vehicle functions to be realized in the electric/electronic (E/E) system (e.g, control applications). However, as the means to specify application-specific timing requirements where informal in Törngren's work, these were not introduced into the EAST-ADL.

Besides the modeling concepts that are part of the EAST-ADL, the EAST-EEA project also defined a middleware-centric ECU software architecture for automotive electronic control units (ECUs) [80]. The idea was to separate and thus decouple the application-software that realizes the vehicle functions from the underlying platform software (operating system, communication stack, I/O drivers) by means of a middleware transparent to the application software. This enables the reuse of application software in different development platforms and over different

3 Related Work

projects as well as ease the migration of application software between electronic control units. The middleware-centric ECU software architecture was then a primary target for standardization by the AUTOSAR initiative.

A limitation of the EAST-ADL is that it does not contain any concepts for the description of application-specific timing requirements or how any timing related engineering information could be handled.

ATESST (EAST-ADL2)

The ATESST project [77] was the successor of the EAST-EEA project and was conducted from 2006 to 2008. At that time, the AUTOSAR consortium has already released concepts that formerly formed substantial aspects of the EAST-EEA system model. These are firstly concepts used on the implementation level and the operational level, and secondly the notion of the layered ECU software architecture. Furthermore, other standardization bodies such as the Object Management Group (OMG) have released updates of the Unified Modeling Language (UML2.0) [61] as well as a specific profile called SysML (Systems Modeling Language) [63] for systems engineering.

The objectives of the ATESST project were thus:

- to align the concepts of the EAST-ADL from the EAST-EEA project with the latest concepts of the AUTOSAR standard at that time. For this, the meta-modeling approach chosen by AUTOSAR for the formalization of its concepts was adapted to the EAST-ADL, and the proprietary concepts on the operational level and implementation level of the EAST-EEA system model were replaced by the corresponding AUTOSAR concepts.
- to align the concepts of the EAST-ADL with the concepts of SysML. This was achieved by basing the functional modeling concepts of the EAST-ADL on concepts lend from SysML through the integration of both meta-models.
- to align the EAST-ADL with UML2.0. A UML2.0-profile was defined such that the EAST-ADL2 could be used in off-the-shelf UML2.0 modeling tools that support UML2.0 profiles.

The efforts of the ATESST project resulted in the updated version of the EAST-ADL, the EAST-ADL2 [47].

The EAST-ADL2 also contains further concepts that go beyond the original concepts of the EAST-ADL and AUTOSAR [35]. This includes a concept for describing timing requirements orthogonal to functional models (analysis and design level) or software architecture models (implementation level, i.e. AUTOSAR). Timing requirements are annotated to system model artifacts by means of chains of concatenated events where the events point to ports of the function or software entities. While the general ideas go in the right direction, the solutions provided in the EAST-ADL2 are, however, technically not sound as the concepts to express chains of cause and effects are not precise. For example, it is not possible to generate an

instrumentation for simulation or monitoring such that the degree of fulfillment of the timing requirements can be evaluated.

The notion of refinement of system models (i.e., from an analysis level model over a design level model to an implementation level model) is also reflected in how timing requirements are treated: timing requirements from a system model on a higher architecture abstraction level are to be refined into timing requirements on a system model on a lower architecture abstraction level.

A concrete concept for the determination of timing properties — be the concept based on analysis, simulation, or monitoring — is, however, missing in the EAST-ADL2. The EAST-ADL2 thus only insufficiently supports the use-case of evaluating the degree of fulfillment of application-specific timing requirements.

AUTOSAR

In 2003, the AUTomotive Open System ARchitecture (AUTOSAR) initiative [5] was established as an industrial standardization organization. The goal was to standardize substantial non-competitive aspects of automotive embedded systems development in order to reduce development complexity and consequently development time and costs. AUTOSAR is the successor of several previously existing automotive standards such as OSEK/VDX [65], CAN [33] [64], Flexray [32], and LIN [51], and integrates them into a rigorous industry standard. While the concepts of these standards were integrated into AUTOSAR, the previous standardization bodies discontinued the further development of their concepts (e.g., OSEK/VDX).

The main achievements of AUTOSAR are

- the standardized, layered ECU software architecture with its middleware layer, the so-called Runtime Environment (RTE),
- a software component technology for the description of application software, supporting hierarchical software architectures and different communication patterns,
- and a methodology that describes how to built AUTOSAR-compliant systems.

For the middleware-based, layered ECU software architecture, AUTOSAR was inspired by ideas from the EAST-EEA project. However, not all concepts developed within the EAST-EEA project were adapted. For example, the concepts for functional modeling that were part of the EAST-ADL were not considered. The focus of AUTOSAR is on the lower abstraction levels of the EAST-EEA system model, i.e. implementation level and operational level.

For its software component technology, AUTOSAR was inspired by concepts of the MSR-SW standard [53]. MSR-SW contains concepts for the modeling of hierarchical application software architectures and an XML-based data exchange format to ease the exchange of engineering information between vehicle manufacturers and suppliers. However, MSR-SW is restricted to single-ECU systems while AUTOSAR addresses single-ECU systems and distributed systems.

3 Related Work

The AUTOSAR initiative has undertaken several approaches towards the definition of concepts that form a timing model. However, none of these approaches has led to sound concepts yet that enable developers of AUTOSAR systems to evaluate the degree of fulfillment of application-specific timing requirements. In the following, the approaches are briefly discussed.

In 2004/2005, a first internal working group was established that discussed problems and potential solutions around the description of timing requirements and the determination of timing properties. The group, however, failed to provide a solution that was officially part of the standard. It was argued that the problems were subject to research and could not be solved directly by the standardization body. However, as no intermediate results (e.g., a problem specification report) were released by AUTOSAR, the research communities could not attack this problem directly and propose a solution. Timing remained an open problem.

Towards the release 3.0/3.1 of AUTOSAR (2006), a Timing Model for AUTOSAR has been proposed as part of the specification of the Virtual Functional Bus (VFB) [12]. However, this timing model has been explicitly marked as being informal and provided for information only. We have identified several problems with respect to the proposed concepts for the timing model. First of all, the concrete events that are described in the timing model are not precise. It is unclear when (time) and where (location) these events occur in an AUTOSAR system. This can lead to ambiguous signal path descriptions. Another problem with respect to the AUTOSAR communication patterns is that only two out of three communication patterns are considered (Inter-Runnable communication is not treated). A further problem is that possible effects of the implicit and explicit Sender/Receiver communication patterns on the timing behavior are not covered. The latter, however, is important to adequately deal with differences in the timing behavior due to the concrete choice of Sender/Receiver communication pattern. The timing model furthermore lacks concepts to express application-specific timing requirements such as those that we derived for control applications. It is thus not possible to specify application-specific timing requirements of this important class of real-time applications. Also, it is unclear how corresponding timing properties can be determined such that the degree of fulfillment of the timing requirements can be evaluated.

From 2007 to 2009, a joint public funded project called TIMMO (TIming MODel) [81] was conducted by a consortium of industrial partners of the automotive domain (OEMs, Tier-1's, tool vendors) and universities. The objective was to define a timing model for automotive embedded systems development and to describe a methodology how timing-related engineering information could be handled over the development phases. Rather than inventing a new architecture description language, it was decided to take the EAST-ADL2 and AUTOSAR as a reference architecture and to define new concepts based on these.

The results of the TIMMO project are firstly the Timing Augmented Description Language (TADL) that is based on EAST-ADL2 and AUTOSAR, and secondly a methodology that describes how to apply the TADL.

The TADL contains concepts to describe timing requirements based on chains of events, also referred to as *timing-chains*. The TADL specific events that are used by timing chains refer to structural elements in an EAST-ADL2 function architecture in an AUTOSAR software architecture. The problem with this is that there is no clear foundation for when and where these events occur, or for how such events can be monitored. Indeed, when it comes to monitoring such kinds of events in a real-time system, there are many open questions such as where to place instrumentation instructions in the source code of an AUTOSAR system.

The concept of TADL timing chains is to concatenate segments of timing chains to timing chains reaching from a sensor to an actuator. Unfortunately, the developed concepts do not integrate well with the concepts that allow the establishment of function hierarchies in EAST-ADL2 or software component hierarchies in AUTOSAR. This is problematic as it is then not possible to further use this information, e.g. in order to obtain an instrumentation for monitoring or simulation.

The TIMMO methodology describes how engineering information is derived and passed over between the different stages of a development. For the latter, the abstraction levels of the EAST-ADL2 were interpreted as different stages of a development.

In 2008/2009, a second internal working group was established within the AUTOSAR consortium. Due to a vivid exchange between this second internal working group and the TIMMO project, the AUTOSAR concepts are very akin to the TADL concepts. The results of that working group now form part of the most recent release 4.0 of the AUTOSAR standard (begin of 2010). The concepts, however, suffer from the same deficiencies as the TADL.

In summary, we can state that there have been several approaches towards a timing model for AUTOSAR, however, none of them has provided a sound solution yet. The idea to describe timing requirements based on chains of events that denote signal paths is in principle the same as we follow in our work. However, while our work targets the simulation and monitoring with the help of instrumentations that are generated from a formal specification in order to then analyze the event trace files and determine appropriate timing properties, the existing ideas try to integrate concepts from formal timing analysis approaches (e.g. scheduling analysis). The latter, however, does not work out properly as the approach has only conceptually been described so far, but not technically evaluated or demonstrated. The technical realization of these ideas is also not satisfying as it is unclear how to work with the information that can be specified by the official AUTOSAR or TIMMO concepts. The integration of the new timing model concepts with existing concepts from AUTOSAR is also not very well achieved as it entails ambiguities. Furthermore, the considered timing requirements only focus on reactive real-time applications and do not adequately consider application-specific timing requirements of control applications which are important in the automotive domain.

3 Related Work

Other Related Work and Tools

Besides the so far discussed related work, there is further related work that has been conducted and which should be mentioned. Also, there are tools that are based on similar concepts compared to AUTOSAR as well as tools that are comparable to the monitoring-based performance analysis tools in the telecommunication domain. These are also described in the following.

ASCET

ASCET [28] is tool suite from ETAS GmbH for the specification, design and implementation (by means of automatic code generation) of automotive ECU software. ASCET is based on concepts developed by Eppinger [20] and Bosch Corporate Research (Bosch CR) in the late 1980's and early 1990's.

For behavioral specifications, ASCET provides graphical means in terms of block diagram and state machine specifications (akin to Matlab/Simulink [78] and Stateflow [79] from The MathWorks), or the proprietary Embedded Software Description Language (ESDL) as a textual notation. For the specification of software architectures, ASCET provides own modeling concepts to abstract from C-code implementations. These concepts are used in combination with each other in a specific way: an ASCET-project represents an ECU software development project that consists of single-instantiable components termed ASCET-modules. ASCET-classes are components that can be instantiated one or multiple times within such modules. ASCET-modules contain so-called processes that operate on messages that live in a global name-space and that can be send or received by processes in other modules.

ASCET is akin to AUTOSAR in multiple senses:

- Firstly, ASCET provides means to specify hierarchical software architectures by means of components. Modules and classes are comparable with atomic and composite software components in AUTOSAR.
- Secondly, ASCET contains a concept to prevent from data-inconsistencies that can occur due to read/write conflicts of processes intending to access the same message in parallel. The concept is to automatically provide message-copies to processes where a potential read/write conflict can occur. The message-copies are automatically provided through code generation. A similar concept is realized by the Runtime Environment in AUTOSAR ECU systems when Runnable Entities communicate via implicit Sender/Receiver communication.

While ASCET is akin to AUTOSAR, however, it does not provide concepts to specify application-specific timing requirements either. It is thus not possible to adapt a an existing solution from ASCET to AUTOSAR.

Spider/TITUS

The Spider/TITUS tool suite [21] was developed in a joint project between ETAS GmbH, Vector Informatik GmbH and Daimler AG and addressed the need for a

divide and conquer approach for the application software of distributed automotive ECU software. While the Spider/TITUS tools suite has several concepts which are conceptually akin to AUTOSAR, e.g. components that can be hierarchically aggregated into a software architecture, it was never developed as a commercial product and remained a research prototype tool. The Spider/TITUS tool suite does also not provide concepts to describe application-specific timing requirements. It is thus also not possible to adapt an existing solution to AUTOSAR.

Performance Analysis Tools in the Automotive Industry

Although monitoring-based performance analysis is not widely employed in the automotive industry, there exist tools that allow to monitor and debug automotive ECU systems. For example, RTA-TRACE [26] from ETAS GmbH is a monitoring solution for OSEK-based automotive electronic control units. The monitoring instrumentation directly integrates with the real-time operating systems such that interesting events such as the start or termination of tasks can be monitored.

In principle, RTA-TRACE has a similar architecture compared to ZM4/SIMPLE, except that it targets single ECU systems rather than distributed and parallel computer systems. While an extension of RTA-TRACE towards parallel and distributed real-time systems would be possible, it is not the objective of our work.

As the efforts to develop a distributed monitoring system such as ZM4 based on RTA-TRACE is very high (e.g., the high efforts in developing suitable hardware interfaces), we have focussed on single-ECU AUTOSAR systems and used RTA-TRACE for our experiments and the case study. For distributed AUTOSAR systems, our concept requires a distributed monitoring system as for example ZM4. The analysis algorithms that we developed could be integrated in SIMPLE. As an alternative, RTA-TRACE could be extended by concepts to establish a global time base based on synchronized measurement clocks. As interface to the ECU, an ETK-device³ could be used. ETK-devices are widely used as high-performance ECU access devices together with measurement and calibration software in order to adjust the parameters of automotive vehicle functions.

³ETK=Emulator-Tastkopf (dt.) / Emulator Probe (eng.)

4 Foundations

4.1 Control Theory

Many applications that are embedded within the automotive electric/electronic system are control applications. They contribute to the regular operation of the vehicle (e.g., engine control, transmission control), assist the driver (e.g., adaptive cruise control, lane departure control) or perform other autonomous operations. All these applications share some inherent properties as they are control applications. Due to their tight interaction with real-world physical processes, control applications are real-time applications. The methods and terminology used in control theory are thus the foundation for developing these vehicle functions.

In this section, the foundations of control theory are reviewed. Furthermore, the timing requirements that must be satisfied by a computer-system based realization of control applications are subsequently derived. This forms the basis for the kinds of timing requirements that need to be described for such real-time applications that are realized in an AUTOSAR system.

4.1.1 Physical Systems (Processes)

In order to control a physical system or process it is necessary to capture its structure and behavior. This is generally done by describing the inputs and outputs of the physical system as signals and by describing the relation between these signals.

Figure 4.1 depicts the structure of a physical system (also called *process* or *plant*) as block diagram.

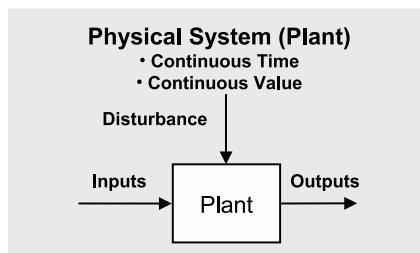


Figure 4.1: Physical system or process

Physical systems or processes are often represented as blocks along with their incoming and outgoing signals. Systems can furthermore be classified according to

4 Foundations

various characteristics of their input-output transfer behavior. In the following two important characteristics are discussed, linearity and time-variance.

Linearity

Systems can be classified either into *linear* systems or *non-linear* systems. In a linear system, the relation between the system inputs and the system outputs can be described by a linear equation, meaning that input signals are amplified or damped to output signals. Formally, linearity can be defined as follows:

Definition 1 (Linearity) Let S be a system with input u and output y . Let $u_1(t)$ and $u_2(t)$ be two different input signals, and $y_1(t) = S(u_1(t))$ and $y_2(t) = S(u_2(t))$ the respective output signals which can be observed. Furthermore, let $y_3(t) = S(u_1(t) + u_2(t))$.

S is a linear system iff

1. $y_3(t) = y_1(t) + y_2(t)$ (additivity) and
2. $\forall t, \alpha : S(\alpha u_1(t)) = \alpha y_1(t)$ (scaling)

The properties additivity and scaling are subsumed to the superposition principle.

Figures 4.2(a) and 4.2(b) depicts the relation of input and output signals in a linear system and a non-linear system, respectively.

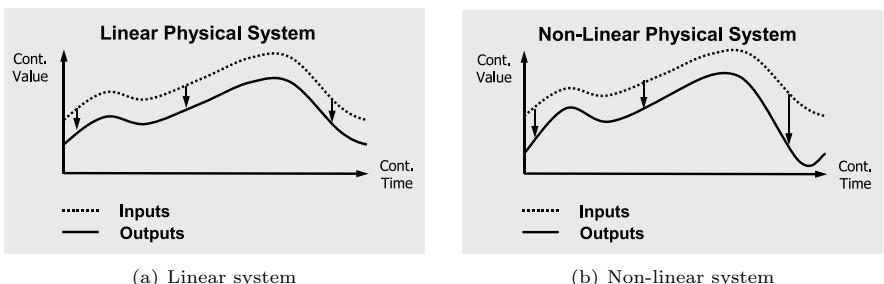


Figure 4.2: Relation of input and output signals in linear and non-linear systems

In the linear system example, the input signal is proportionally damped to the output signal. In the non-linear system example, the input signal is also damped to the output signal, however, the scaling is not homogeneous.

Time Variance

Systems can be classified into either *time-variant (TV)* or *time-invariant (TIV)* systems. A system is time-invariant if a time-shifted input signal results in a time-shifted output signal. A system is time-variant if this property does not hold.

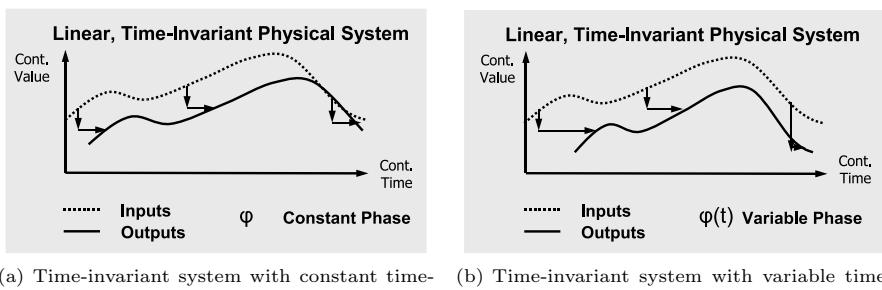
Formally, time-invariance can be defined as follows:

Definition 2 (Time-Invariance) Let S be a system with input u and output y .

S is time-invariant with constant phase φ iff $y(t - \varphi) = S(u(t - \varphi))$

S is time-invariant with variable phase $\varphi(t)$ iff $y(t - \varphi(t)) = S(u(t - \varphi(t)))$

Figure 4.3 depicts the relation of input and output signals in a constant and variable time-invariant system.



(a) Time-invariant system with constant time-shift (b) Time-invariant system with variable time-shift

Figure 4.3: Relation of input and output signals in time-invariant systems

Other characteristics according to which systems can be classified include, amongst others, *stability*, *causality*, and *parameter distribution*.

Control theory focuses on the mathematical description of systems. In the following, only linear systems are considered, and only time-invariant systems with constant phase are considered because this class of systems can be described by mathematical models with manageable complexity (i.e., no time-dependent parameters). Furthermore, we focus on causal systems as the relation between the system inputs and system outputs is clear and matches the properties of physical systems in nature. Mathematical models for dynamic systems are discussed in section 4.1.3. For the case of non-linear systems, we assume that a suitable linear model can be found which adequately approximates the non-linear system around the interested point of operation.

4.1.2 Signal Representation in Time and Value Domain

Another important aspect in the description of systems is the description of signals in time and value domain. System signals can be classified according to their representation in time and value domain either as being continuous or discrete. In the

4 Foundations

examples in section 4.1.1, the input and output signals have been represented as continuous time, continuous value signals. Figure 4.4 depicts the four combinations which are possible for signal representation in continuous and discrete time and value domain.

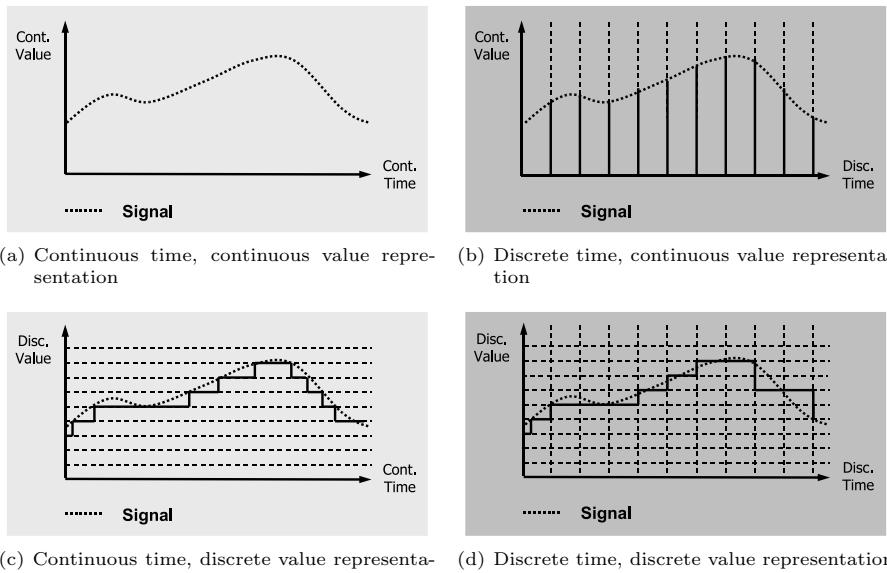


Figure 4.4: The four possible signal representations in continuous and discrete time and value domains

While most physical systems are continuous time systems where the signals can be represented as continuous time, continuous value signals, implementations in digital computer systems requires them to be represented in the discrete time domain. The translation of continuous time to discrete time representation is referred to as *discretization*. A continuous time signal can be discretized by sampling its values at equidistant instants.

4.1.3 Mathematical Models for Dynamic Systems

To describe the dynamic behavior of systems, mathematical models are used. The models are obtained by analyzing the relations between the inputs and outputs of systems. The objective is to study their behavior, either through various analysis techniques or by observations based on measurements or simulation.

Based on the input-output data of a dynamic system, a dynamic system can be modeled in a chosen mathematical form by determining the numerical values of

the parameters which are associated with the particular form of the model. The process of identifying and modeling system dynamics is termed *system identification* or *process identification*. System identification is an own discipline within the field of control engineering and is not further discussed in this thesis.

In the following, a prominent mathematical modeling technique for dynamic systems used in control engineering theory is briefly discussed: *State Space Descriptions*. We also discuss how time delays can be adequately be represented in the mathematical models and derive the first timing requirement. Other mathematical modeling techniques are for example Differential Equation models or Transfer Function models. These are not further described.

State Space Description Models

State space description models describe the structure of a system by capturing the system state in terms of state variables. The system dynamics are then expressed by a set of equations which describe how the system output $y(t)$ is produced based on the current state $x(t)$ and the system input $u(t)$, and which describe how the system state $x(t)$ is updated based on the system input $u(t)$ and the current state $x(t)$.

In the following, the state-space description models for continuous-time, time invariant linear systems without time delay and with time delay are described.

Systems without Delay

A continuous time, time-invariant linear system can be described by the first-order differential equation system [3]:

$$\dot{x}(t) = Ax(t) + Bu(t) \quad (4.1)$$

$$y(t) = Cx(t) + Du(t) \quad (4.2)$$

where

$x(t)$ State vector

$\dot{x}(t)$ Update of state vector

$u(t)$ Input signal vector

$y(t)$ Output signal vector

A Continuous time system matrix

B Continuous time input matrix

C Continuous time output matrix

D Continuous time direct (or feedforward/feedthrough) matrix

The mathematical model described by equations 4.1 and 4.2 is also called the *state-space description* of the continuous time system.

The state space model of the continuous time, time-invariant linear system can be discretized by sampling the system inputs at equidistant instants, i.e. with a

4 Foundations

constant sampling rate h . This leads to its discrete-time representation as difference equations system [3]:

$$x(kh + h) = \Phi(h) x(kh) + \Gamma(h) u(kh) \quad (4.3)$$

$$y(kh) = C x(kh) + D u(kh) \quad (4.4)$$

$$\Phi(h) = e^{Ah} \quad (4.5)$$

$$\Gamma(h) = \int_0^h e^{As} ds B \quad (4.6)$$

where

$k = 0, 1, 2, \dots$	Discrete-time sampling instant sequence number
h	Constant sampling interval
$x(kh)$	State vector at instant kh
$u(kh)$	Input signal vector at instant kh
$y(kh)$	Output signal vector at instant kh
A, B, C, D	System / Input / Output / Direct matrix
$\Phi(h)$	Discrete time system matrix
$\Gamma(h)$	Discrete time input matrix

An exact derivation of the system and input matrices for the discrete-time state space description model can be found in standard text books, e.g. [73] and [87].

Due to the discrete-time sampling of continuous-time signals, the following timing requirement must be satisfied by a discrete-time realization:

Timing Requirement 1 (Equidistant Input Sampling) *In a discrete-time realization of a continuous-time process, sampling of continuous-time signals must be performed at equidistant instants. For this, a constant input sampling interval h must be maintained.*

If this timing requirement is satisfied, then the output signals are also produced at equidistant instants. In general, this is implicitly assumed. The latter, however, cannot be taken for granted by a discrete-time realization of the system, e.g., in a computer system where variable computational delays can occur. To account for this, we formulate the following corresponding timing requirement:

Timing Requirement 2 (Equidistant Output Production) *In a discrete-time realization of a continuous-time process, output signals must be produced at equidistant instants. Through this, a constant output production interval h must be maintained.*

Systems with Constant Delay

In the case of a system with a constant time-delay τ , $0 < \tau \leq h$, the delay can be included into the state space description model in order to adequately account for it. The assumption of $0 < \tau \leq h$ means that the time delay is smaller than the sampling period. The state space description model is then written as follows:

$$\dot{x}(t) = Ax(t) + Bu(t - \tau) \quad (4.7)$$

$$y(t) = Cx(t) + Du(t) \quad (4.8)$$

Again, the state space model of the continuous time, time-invariant linear system with delay can be discretized by sampling the system inputs at equidistant instants h . The discretization leads to its discrete-time representation as a difference equation.

Figure 4.5 depicts a close-up of the effect of the time delay between the output signal $y(t)$ and the input signal $u(t)$. Note that $y(t)$ is the output signal of a physical system (continuous-time) which serves as input signal $u(t)$ for a considered discrete-time process.

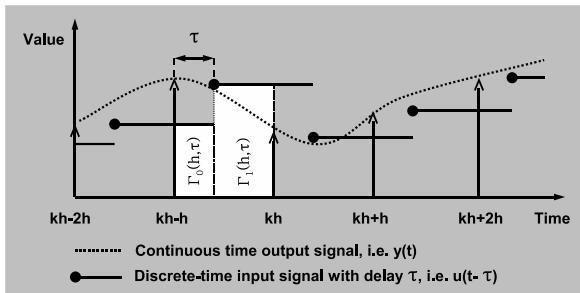


Figure 4.5: Effect of a time delay of a continuous-time output signal on the discrete-time input signal

In the example, the output signal of a physical system $y(t)$ is sampled with a constant input sampling interval h . The system time delay τ leads to a time delay in the input signal $u(t)$ of the process that samples the output signal, thus $u(t - \tau)$.

As can be seen from the figure the input signal must be considered as two parts over a single sampling interval. The discrete-time state-space model can thus be described by the following difference equations system [3]:

4 Foundations

$$x(kh + h) = \Phi(h) x(kh) + \Gamma_0(h, \tau) u(kh) + \Gamma_1(h, \tau) u(kh - h) \quad (4.9)$$

$$y(kh) = C x(kh) + D u(kh) \quad (4.10)$$

$$\Phi(h) = e^{Ah} \quad (4.11)$$

$$\Gamma_0(h, \tau) = \int_0^{h-\tau} e^{As} ds B \quad (4.12)$$

$$\Gamma_1(h, \tau) = e^{A(h-\tau)} \int_0^{\tau} e^{As} ds B \quad (4.13)$$

where

$k = 0, 1, 2, \dots$ Discrete-time sampling instant sequence number

h Constant sampling interval

$x(kh)$ State vector at instant kh

$u(kh)$ Input signal vector at instant kh

$y(kh)$ Output signal vector at instant kh

A, B, C, D System / Input / Output / Direct matrix

$\Phi(h)$ Discrete time system matrix

$\Gamma_0(h, \tau), \Gamma_1(h, \tau)$ Discrete time input matrices

In the case of time-varying delays, the state-space model is still valid, however, the system matrices will be time varying ([3], [89]), i.e. $A = A(t)$, $B = B(t)$, etc. This, however, is not further discussed in this thesis.

4.1.4 Response Analysis of Dynamic Systems

The characteristic performance properties of a dynamic system can be evaluated through so-called *transient response methods* to standard characteristic input signals. Such standard input signals are step, impulse, ramp or sine wave functions. The accuracy of the response to the input signal in the steady-state is a characteristic performance measure evaluated by means of response methods (*steady-state accuracy*). The responses methods are named after the input signal which are presented to the process, i.e. step response, impulse response, ramp response and sine wave response. Figure 4.6(a) shows the step response of an example second order system to a unit step (i.e., a step of size 1) in the continuous-time domain.

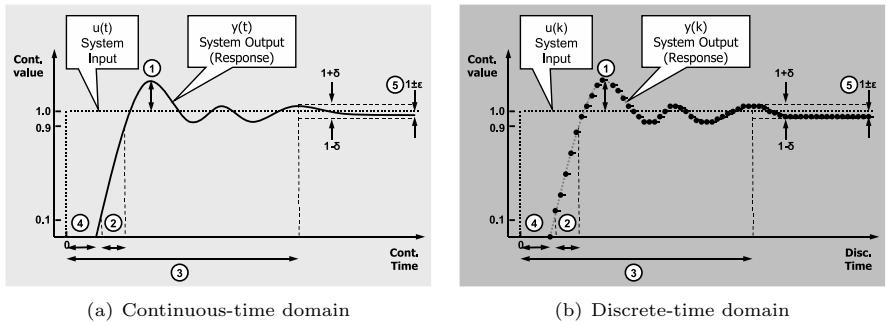


Figure 4.6: Unit step response of a second order system

The unit step response gives insight into the control performance as the properties of the following characteristics can be evaluated:

Item	Characteristic	Description
1	Overshoot	Denotes the maximum peak value of the response signal
2	Rise time	Denotes the time required for the response to rise from 10% to 90% of its final value
3	Settling time	Denotes the time required for the response to reach and stay within a range about the input value (e.g., within $\delta = 5\%$)
4	Dead time	Denotes the time lag of the system between the input signal and the output signal
5	Steady state error	Denotes the difference between the input (desired response) and the response of the system after settling time

Response analysis methods are an important control engineering tool as they allow the determination of the required sampling rates for a discretization of continuous-time signals considered under dynamic performance aspects. A general applicable rule of thumb [3] states that the sampling rate should be chosen in such a way that the continuous-time signal is sampled 4-10 times per rise time in the unit step response. This results in a set of viable sampling rates for the discretization of continuous-time signals. The control engineer has to decide for a specific sampling rate. Response analysis methods are thus a means to determine the size of a input sampling interval h (see state-space description model in 4.1.3).

Figure 4.6(b) shows the step response of the example second order system to a unit step in the discrete-time domain. In the example, a sampling rate of 5 samples per rise time has been chosen.

4 Foundations

As can be seen in the figure, the critical phase is during the rise of the signal where the samples experience the largest differences of their values due to the steep incline.

4.1.5 Control Systems

The control applications which can be found in modern automotive vehicles are realized as control systems that in general comprises both mechanics and electronics. The electronics of the control system are further subdivided into hardware and software parts. According to the IEEE, control systems can be defined as follows:

Definition 3 (Control System (IEEE)) *A system in which a desired effect is achieved by operating on various inputs to the system until the output, which is a measure of the desired effect, falls within an acceptable range of values.*

Due to the importance of control systems, a complete engineering discipline has grown and is devoted to the theory and practice of building such systems. *Control engineering* is the engineering discipline of mathematically modeling systems of a diverse nature, analyzing their temporally changing dynamics, and using control theory to create a controller that will cause the system to behave in a desired manner.

Control systems were first analyzed and designed as *analogue control systems* in the continuous-time domain. With the success of digital computer technology in the second half of the last century, almost all control systems are nowadays implemented as *digital control systems*, or *computer control systems* [3], in the discrete-time domain.

In the following, the two fundamental control system concepts, open-loop and closed-loop control, are described and distinguished upon their main characteristics.

4.1.6 Open and Closed-Loop Control Systems

Control systems can be classified into *open-loop control systems* and *closed-loop control systems*. This depends upon if the control decisions are influenced by information stemming from the process under control or not.

Definition 4 *A closed-loop control system is a control system where the control decisions are based on the feedback of information (i.e., one or multiple measurement signals) from the plant or process under control. The feedback is used by the controller to make decisions about changes to the control signal that acts on the plant.*

Definition 5 *An open-loop control system is a control system which does not have or use feedback from the plant.*

In control engineering terminology, the process output of the system shown in figure 4.1 is also referred to as a *controlled variable*, i.e., it denotes a condition or quantity that is measured or controlled. The process input is also referred to as the

manipulated variable, i.e., it is a condition or quantity that is varied by a controller in order to affect the value or condition of the controlled variable.

Figure 4.7 depicts the basic structure of an open-loop and a closed-loop control system in block diagram form ([3], [40]).

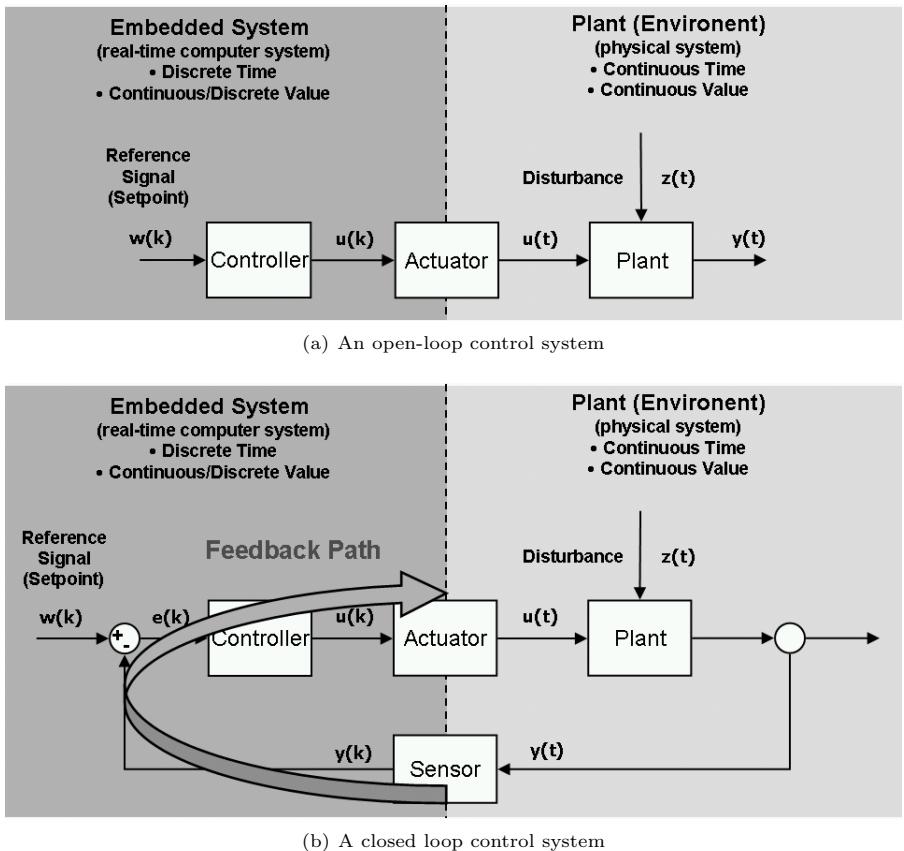


Figure 4.7: Basic structure of open and closed loop control systems

The main parts of an open-loop control system are the *plant* (i.e., the process under control), the *controller* and the *actuator* which links the controller to the plant. The main parts of a closed-loop control system are the *plant*, a measurement device generally termed *sensor*, the *controller* and the *actuator*. In the figures, it is assumed that the controllers are implemented as digital computer systems in the discrete-time domain.

4 Foundations

The controller part of an open-loop control system can for example consist of a single (linear or non-linear) gain which amplifies the setpoint value to match the input value range of the actuator in such a way that the process under control (i.e., the plant) is influenced as desired.

The controller part of a closed-loop control system is in general more complex. The output of the plant is a continuous-time signal, denoted $y(t)$, which is captured by the sensor, converted to a discrete-time representation $y(k)$ and fed to the controller. An error signal $e(k)$ is computed as the difference between the reference signal $w(k)$ (also referred to as setpoint signal), and the discrete-time version of the plant output $y(k)$. In the figure it is assumed that the reference signal is fixed and realized as a constant, or that it is captured by a sensor which is not shown, and that the value of the setpoint sensor has already been discretized. The controller processes the error signal and produces an input signal $u(k)$ for the plant. The output signal of the controller is effected on the plant through the actuator such that the control error is reduced or even removed. The actuator converts the discrete-time signal $u(k)$ to a continuous-time signal, e.g., through a zero-order or first-order hold circuit.

The task of the controller is to hold the controlled variable u either on a constant setpoint value $w(k) = \text{const}$ (*fixed command control*) or to track a time-varying reference variable $w(k) \neq \text{const}$ (*variable command control*), independent of external disturbances $z(t)$ that act on the plant, while minimizing the steady-state control error $e(t) = y(t) - w(t)$, i.e. $e(t) \leq \varepsilon$.

Compared to open-loop control systems, the principle of negative feedback (*feedback principle*) and thus *the existence of a closed signal path* is the main characteristic of closed-loop control systems. The signal path from the output $y(t)$ of the plant process over the control application in the embedded system to the input $u(t)$ of the plant process is also referred to as *feedback path*.

In the following, only closed-loop feedback control systems are considered as they are in general the preferred form of control application and the predominant form of control applications in the automotive domain.

4.1.7 Control Systems with Time-Delays

In the previous sections, the basic structure of control systems, especially closed-loop control systems, has been described. Furthermore, the mathematical models that allow a precise description of the dynamic behavior of systems have been described. This also included systems that include a time delay. It has been shown that constant time delays can be accounted for as a parameter in space state description models that are used to model the structure and behavior of a process.

In the following, two cases with respect to time delays in closed-loop control applications are distinguished: firstly, the effect of a time delay of a plant on controller design, and secondly the effects of a time delay in a discrete-time realization of a closed-loop controller on the controlled plant.

Time-Delays in the Plant

Figure 4.8 (see next page) depicts an example closed-loop control system where the plant process has a constant time delay. This time delay is denoted τ_{Plant} . It is effective on the signal path from the input signal to the output signal of the plant.

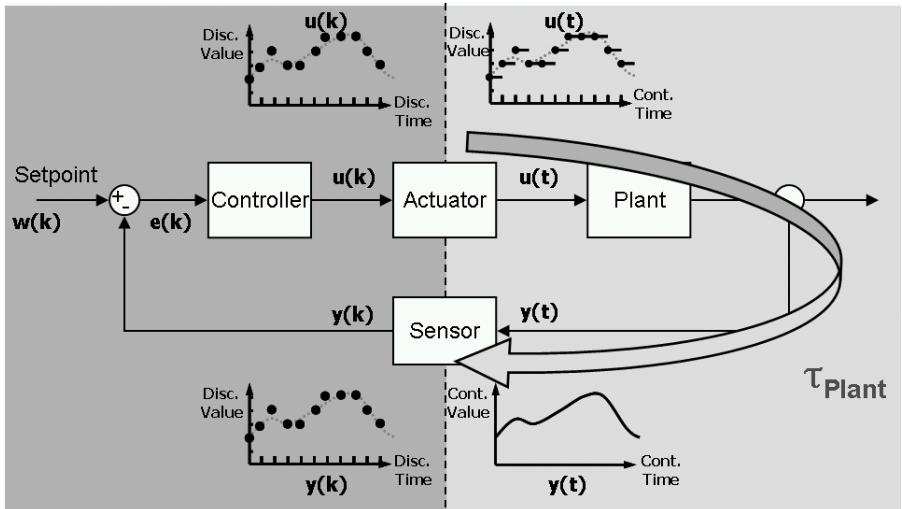


Figure 4.8: Closed-loop control system with time delay in plant process

Time-delays in plant processes are present when the signal transportation and transformation from an input to an output takes a considerable amount of time. Example systems are mass transportation systems (e.g., a conveyor belt) or systems where the sensor and the actuator are located far apart (e.g., satellite positioning control from ground station).

Time-delays in the plant can be accounted for by using special control architecture concepts such as a Smith-predicator [3]. The Smith-predicator makes a prediction of the future development of the control variable. For this purpose, the individual parts with and without dead-time of the process are considered individually, and a model of the delay is incorporated in the controller.

Time-delays in plants can thus be compensated by incorporating them in the design of the controller.

Time-Delays in the Controller Implementation

Due to the fact that discrete-time controllers are implemented in digital computer systems in terms of clocked hardware and software, inevitable delays are introduced into the controller. Such delays are commonly subsumed as computational delays.

4 Foundations

Figure 4.9 depicts an example closed-loop control system with a time delay $\tau_{\text{Sensor} \rightarrow \text{Controller} \rightarrow \text{Actuator}}$ (or τ_{SCA} for short) between the plant output signal $y(t)$ and the plant input signal $u(t)$. The time delay τ_{SCA} denotes the effective time delay between related sampling and actuation instants and is also referred to as *feedback path delay* or *feedback loop delay* ([84], [86], [85]).

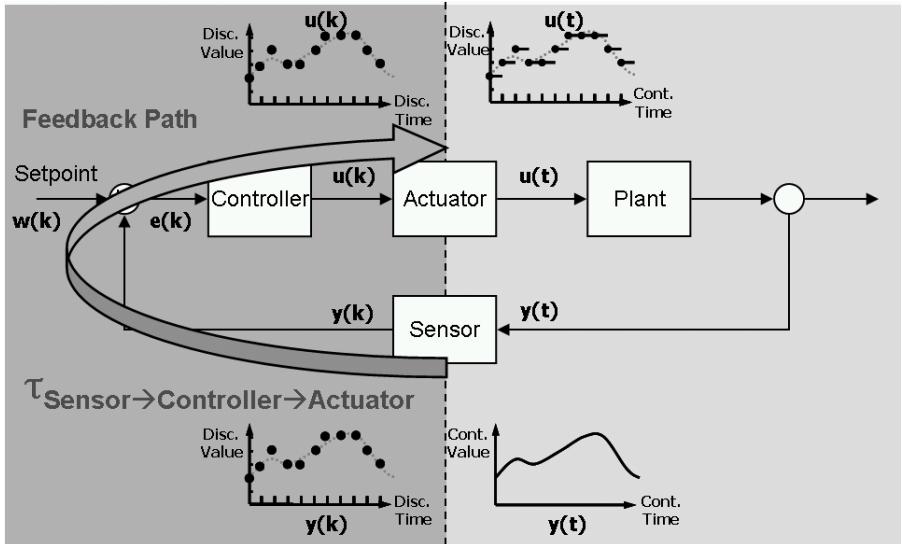


Figure 4.9: Closed-loop control system with time delay in sensor, controller and actuator processes

Time delays that are effective on the feedback path and thus on related sensor data acquisition and output data actuation actions can have negative effects such as deteriorated control quality or even loss of control.

The assumptions of the mathematical models (e.g., state space description models) by which the dynamic behavior of a system is described translate into timing requirements for the realization of a control system in digital hardware and software. These timing requirements are:

Timing Requirement 3 (Minimization of feedback path delay) *In a discrete-time realization of a control application, the feedback path delay from a sensor to an actuator must be minimized. Formally, this is expressed by $\tau_{\text{SCA}} \equiv 0$.*

If the minimized delay cannot be neglected in controller design then it must be incorporated in the controller design. This, however, requires the feedback path delay to be constant such that it can be incorporated in the mathematical model for the controller and thus leads to the following timing requirement:

Timing Requirement 4 (Maintenance of constant feedback path delay)

In a discrete-time realization of a control application, if feedback path delay from a sensor to an actuator is not negligible, then the end-to-end delay must be maintained constant to ease its compensation. Formally, this can be expressed as $\tau_{SCA} = const.$

Note that these two timing requirements are alternatives and cannot be specified for a control application at the same time.

4.1.8 Closed-Loop Control System Architectures

In the previous sections, the structure of control systems which consist of a control loop with a single sensor and a single actuator have been considered. In this section, a more complex control system architecture is additionally explored and described. From that, two further timing requirements are derived.

Depending on the number of input signals and output signals, control systems can be classified into *single-input-single-output (SISO)* and *multiple-inputs-multiple-outputs (MIMO)* systems in the extreme, or any mixed form of single input/output and multiple input/output in between (i.e., SIMO and MISO).

Single-Input-Single-Output (SISO) Systems

The closed-loop control systems discussed so far have been SISO closed-loop control systems. SISO systems can be described by state space description models where the input signals and output signals are scalars. The theory for linear time-invariant control systems assumes the timing requirements 1 (equidistant input sampling) and 2 (equidistant output production), 3 and 4 (minimal/constant feedback path delay) to be satisfied by a discrete-time realization.

As has been shown in section 4.1.3, a constant feedback path delay $\tau_{SCA} = const$ can be compensated during controller design while variable delays are more complex and should generally be avoided.

Multiple-Input-Multiple-Output (MIMO) Systems

Figure 4.10 depicts the structure of a MIMO closed-loop control system. Compared to the SISO closed-loop control system, m different physical quantities which are produced by the plant as outputs $y_1(t), y_2(t) \dots y_m(t)$ are sensed by m dedicated sensors and fed to the controller. The controller then produces n different input signals $u_1(t), u_2(t) \dots u_n(t)$ which are effected on the plant by n actuators. Conceptually, there are thus $n \times m$ signal paths leading from the output signals of the process $y_i(t), i \in \{1..m\}$ to the computed input signals $u_j(t), j \in \{1..n\}$. These $n \times m$ signal paths are feedback paths that can be considered individually or in relation to each other.

MIMO closed-loop control systems can be described by state space description models where the input signals and output signals are vectors of adequate dimension

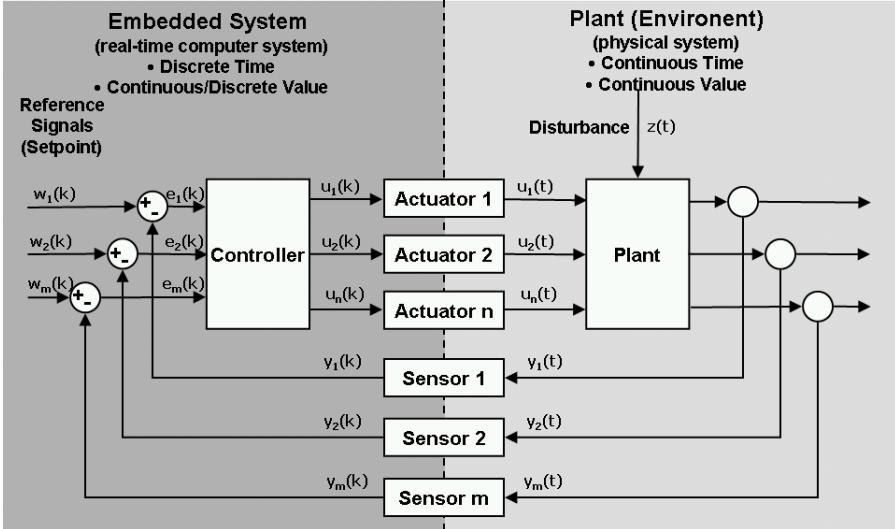


Figure 4.10: Structure of a MIMO closed-loop control system

(i.e., $\overrightarrow{u(t)}$, $\overrightarrow{y(t)}$, etc.). This implies that in the state-space description model, the respective matrices also have adequate dimension. Note that the structure of the MIMO system depicted in figure 4.10 can be represented in the same way as the SISO system when the scalar signals are represented as vectors. A representation of signals in MIMO systems as vectors makes sense if the signals are of the same physical type (e.g., wheel speed velocities with unit km/h, measures from redundant sensors, etc.) and can be subsumed for a more succinct graphical representation of the control system structure.

In MIMO systems, two additional timing requirements can be expressed compared to SISO systems. If the output or input signals of a physical process ($y_i(t)$ and $u_j(t)$, respectively) are of the same physical kind (e.g., the wheel speeds from the four wheels of a passenger vehicle) they in general have the same physical dynamics. This implies the following two timing requirements:

Timing Requirement 5 (Synchronized measurement variable sampling)
The measurement variables of the physical process need to be sensed in such a way that they are in synchrony with each other and form a temporally consistent basis for the input signal computations within the controller. For this, the time delays from each of the sensors to the controller must be less than a specified maximum time delay. Formally, this can be expressed as $\forall i \in \{1..n\} : d_{Sensor(i) \rightarrow Controller} \leq d_{input sync}^{max}$, where $d_{Sensor(i) \rightarrow Controller}$ denotes the delay from the acquisition of an input data at sensor $\langle i \rangle$ to the arrival of the data at the controller.

Timing Requirement 6 (Synchronized control signal actuation) *The control signals which are computed by the controller must be temporally consistent and effected in synchrony with each other on the plant process by the actuators. For this, the time delays from the controller to each of the actuators must be less than a specified maximum time delay. Formally, this can be expressed as $\forall j \in \{1..n\} : d_{\text{Controller} \rightarrow \text{Actuator}(j)} \leq d_{\text{outputsync}}^{\max}$ where $d_{\text{Controller} \rightarrow \text{Actuator}(j)}$ denotes the delay from the production of a data at the controller to the effectuation of the data at actuator $\langle j \rangle$.*

If the input or output signals are of different physical kinds (e.g., pressures, temperatures, etc.) they can have different physical dynamics. The timing requirements are then rather individual timing requirements which depend on the causal relation between individual output signals $y_i(t), i \in \{1..m\}$ and input signals $u_j(t), j \in \{1..n\}$.

Examples for MIMO control systems in the automotive domain are the anti-lock braking (ABS) system or the electronic stability program (ESP). In both, the wheel speeds of all four wheels of a passenger vehicle are sensed by individual wheel speed sensors and fed to the controller. The controller then processes the measured wheel speed signals and decides whether a control action to stabilize the vehicle needs to be performed or not. This results in individual output signals for the wheel brake actuators attached to the individual wheels of the vehicle. In the case of a control action, the wheel brake actuators perform a braking action based on the control signal. For control signal computation, it is important that the acquired wheel speeds from all four wheels are temporally consistent. For control signal actuation, it is important that the computed control signals are effectuated in synchrony on the plant. Otherwise, undesired effects can occur due to unsynchronized braking actions.

4.2 Timing Models

The determination of timing properties by means of monitoring or simulation of parallel and distributed computer systems is based on the observation of relevant events in the computer system and the determination of causal and temporal orderings on the collected event traces. The latter is necessary in order to make safe statements about the temporal behavior and the causal relations in a system. Due to that, in the following, the fundamentals of timing models are discussed.

At first we formally define — based on existing approaches from literature — what we understand as an event. Such events can in principle be ordered according two relations: according to the causal ordering relation and the temporal (or chronological) ordering relation. The relationship between these two ordering relations is important in order to develop concepts and approaches that allow correct statements about the temporal behavior of a computer system. There exist different ordering mechanisms to order events that have been monitored in a parallel and distributed computer system: these are central observers, logical clocks and real clocks. These

4 Foundations

ordering mechanisms are briefly described and compared to each other, with the conclusion that for the determination of timing properties such as latencies between events, real clocks are required. Thus, in section 4.2.3, we introduce a formal model for real clocks that has been described by Münzenberger et al., and which we use as a basis for the development of a timing model for AUTOSAR in the following chapters.

4.2.1 Events and Actions

Different perceptions about what an event actually is have been discussed in literature so far. Even when the context is restricted to parallel and distributed computer systems — in contrast to the colloquial usage of the term event — there are different perceptions and many mostly informal definitions. This has for example been already discussed by Mohr [57].

According to Mohr [57], an event can be characterized by the *class* of event it belongs to, a *location* describing where it occurred in the system, and a *time stamp* describing when it occurred. The event class unambiguously identifies the place in the computer system where instances of this event class, referred to events for simplicity reasons in literature, can be observed and monitored. This can, for example, be events which describe the start or termination of a process, or communication events which are associated with the sending or reception of a message between processes.

In order to conceptually distinguish between the notion of events that are used to mark a place in the static structure of a parallel and distributed computer system and the notion of events which are observed during the simulation or monitoring of the dynamic run of the same computer system, we introduce — akin to Mohr [57] and Münzenberger [69] — the terms *Event Class* and *Event Instance*:

Definition 6 (Event Class) *An Event Class precisely denotes a place in a system. An Event Class is uniquely identified through its name.*

In literature, event classes are also termed event types or actions (see Mohr [57]). In our further work, we will also use both terms, however, with a different — for our purposes more suitable — meaning.

Definition 7 (Event Instance) *An Event Instance is an instance of an event class. Event Instances are associated with a time value which denotes the point in time when the corresponding Event Class has been visited.*

Münzenberger [69] and Mohr [57] refer to Event Instances simply as events. However, as for our purposes the distinction between Event Classes and Event Instances is of essential importance, and in order to avoid confusion, we use these terms rather than just the term “event” whenever necessary.

The time value with which an Event Instance is associated is the reading value of a clock. Time values that are readings of logical clocks are integer values in scalar

or vectorial form, time values that are readings of real clocks are in general real numbers.

Events are instantaneous activities that can be monitored in a computer system. Event Classes are consequently markers of places at which these instantaneous activities occur, Event Instances are the corresponding runtime concept.

Analog to the definitions for Event Class and Event Instance we introduce — as an extension — the notion of Action Classes and Action Instances. Action Classes are used to unambiguously identify durational activities within a system with Action Instances being the corresponding runtime concept.

Definition 8 (Action Class) *An Action Class marks a durational action in a system. The action is characterized through two distinct Event Classes which mark the start and the end of the action.*

Definition 9 (Action Instance) *An Action Instance is an instance of an Action Class. Action Instances are associated with two Event Instances whose time values denote the start and end of the run through of an Action Class.*

The relationship between Event Class, Event Instance, Action Class and Action Instance can be seen in the class diagram shown in figure 4.11.

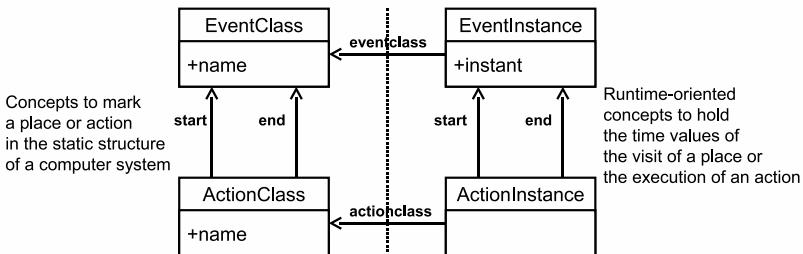


Figure 4.11: Relationship between the concepts of Event Class, Event Instance, Action Class and Action Instance

While events are suitable for the marking and monitoring of instantaneous activities (e.g., a state change in a system), actions are suitable for the marking and monitoring of durational activities (e.g., an operation performing a calculation). A common property of the concepts of events and actions for the marking of instantaneous and durational activities is that they are perceived as being *atomic* at the level of abstraction at which the behavior of a parallel and distributed computer system is considered.

The term action is defined similar by Münzenberger [69], however, there is no distinction between an Action Class and an Action Instance. With the concepts of Münzenberger it is not possible to mark a durational activity with only one entity (as with the Action Class in our case). This, however, is required for the specification

4 Foundations

of chains of cause-and-effect in AUTOSAR-systems. The reason are the available means offered by AUTOSAR for an adequate behavioral abstraction. At the level of abstraction provided by AUTOSAR, it is necessary to mark events and actions with single entities.

In the following, two different ordering relations for events are discussed for sets of events: the causal order relation and the temporal or chronological order relation. Note that the principles of these ordering relations could also be extended to actions. For this, it must be made clear what the order relations mean with respect to the start and end of actions.

Ordering Relations

For the set of event instances there are in principle two ordering relations: the causal ordering relation and the temporal or chronological ordering relation. As both ordering relations are important for the performance analysis of parallel and distributed computer systems, and as there is a non-trivial relationship between both, these are presented in more detail in the following.

Causal Order Relation

The term *causality* denotes the relation between a *cause* and an *effect*. The relation between a cause and an effect is also termed *causal order relation* or *relation of cause-and-effect*.

Causality can be mathematically defined as a strict partial-order relation on a set of events that denote causes and effects.

Definition 10 (Causal Order Relation) Let E be a set of events that denote causes and effects, and let $\mapsto \in (E \times E)$ be a binary relation expressing a causality between two events $a, b \in E$, $a \neq b$. The fact that event a causally affects event b is then formally expressed by $a \mapsto b$.

Note that there are two wordings for $a \mapsto b$:

- we say “*a causally affects b*” if we focus on the cause and formulate what effect results from it
- we say “*b is causally affected by a*” if we focus on the effect and formulate what cause has lead or contributed to it

In general, strict partial-order relations have three important properties: *irreflexivity*, *antisymmetry* and *transitivity*. However, for the causal order relation, only the transitivity property really makes sense as the irreflexivity and antisymmetry properties are not applicable from the definition of causality.

For actions, the causal order relation can be defined analogously. We assume that an action must have ended such that it can have an effect on another action.

Temporal Order Relation

Causality has an intrinsic temporal direction as an effect is always based on a temporally preceding¹ cause. This means that in a cause-and-effect relation, the cause must *happen before* or *occur before* the effect, otherwise, fundamental laws of Newtonian physics are violated, leading to contradictions.

Based on a set of events, the temporal order relation can formally be defined as follows:

Definition 11 (Temporal Order Relation) *Let E be a set of events, and let $\prec \in (E \times E)$ be a binary relation expressing a temporal order between two events $a, b \in E$, $a \neq b$. The fact that event a happens before event b is then formally expressed by $a \prec b$.*

The temporal order relation can equally be defined for actions if the start and end of actions are adequately respected.

With respect to the relationship between the causal order relation and the temporal order relation, the following two observations are important:

1. **Causality implies temporal order.** In a cause-and-effect relation where event a is the cause and b is the effect it is implied that a happens before b . I.e., $a \mapsto b \Rightarrow a \prec b$.
2. **Temporal order does not imply causality.** When an event a happens before an event b it is not implied that a has caused or affected b . I.e., $a \prec b \not\Rightarrow a \mapsto b$.

While temporal order between events is *necessary* to deduce a cause-and-effect relation between the events, it is not *sufficient*. The missing piece of information is a specification of whether it is actually possible that an event of a specific type can causally affect another event of another specific type, and whether this relation is direct or transitive. The latter is an important aspect that we will exploit for the determination of instances of chains of cause and effect. The latter requires firstly a set of events that have temporally consistent time stamps² and secondly a specification of cause-and-effect relations between the events. The latter is called a *chain of cause-and-effect* or *event chain*, for short.

4.2.2 Ordering Mechanisms

In order to give the events that have been collected in a parallel and distributed computer system a meaningful order there exist different so-called *ordering mechanisms*. These are the ordering mechanisms realized by a central observer, so-called logical clocks and real clocks. These ordering mechanisms are briefly described in the following. More detailed descriptions can be found in the work by Klar et al. [48] and the foundational works that these concepts originate from.

¹Note that we refer to Newtonian physics.

²i.e., they can be compared to each other with respect to temporal precedence as their time values either stem from a global clock or a set of clocks that is sufficiently precise synchronized

Central Observer

The concept of a central observer is based on the obvious idea of installing a central observer component in the system that has access to all relevant events that can occur in the system, and that is thus able to consistently monitor their occurrences in the correct order. If such a central observer component additionally has access to a real clock, then it is possible to assign globally valid time stamps to the instances of the events such that also safe temporal relations between the occurrences of events can be determined.

The advantages of such a central observer component are obvious: Only one monitoring component is required which means a simple and lean architecture of the monitoring system. No sharing of work is required among distributed components, neither for the decentralized collection of events nor for their merging into a consistent ordering.

The disadvantages are that such a central observer can only be realized with high technical effort in a parallel and distributed computer system. Klar et al. [48] argue that monitor systems which realize the concept of a central observer can only be built for distributed computer systems with a maximum distance of 5 m due to technical reasons. Through the large number of interfaces to the different computer nodes there is furthermore a bottleneck at the central observer which slows down the collection of events and their analysis.

As a monitoring system is in principle only required for performance analysis, the construction and realization of such a system as a central monitor system which forms part of the parallel and distributed computer system means additional efforts that do not directly contribute to the functional behavior. These efforts are only in seldom cases justifiable against the high costs that they cause.

Logical Clocks

In the following, two types of logical clocks are described which allow the determination of orders on sets of events observed in a parallel and distributed computer system. Logical clocks are based on a common system model consisting of a set of sequential processes $P = \{P_1, P_2, \dots, P_n\}$ which communicate via send/receive messages. The underlying assumption is that events that occur within a process are causally related, i.e. an event a_1 that occurs in a process A prior to an event a_2 is perceived as a cause for the latter. The sending and reception of messages by distinct processes is also perceived as an event where a sending event a in process A is the cause for its corresponding receiving event b in process B .

The initial concept of logical clocks was introduced by Lamport [49] in 1978. It allows the ordering of events that are observed in a parallel and distributed computer system with respect to the causal order in which they occurred using scalar logical time stamps.

The working principle of the ordering mechanism by means of scalar logical clocks is as follows: Each process P_i in the system is assigned a logical scalar counter C_i

which assigns integer values to events that occur in the process. The clock value is updated according to the following rules:

- If an event e (internal event or sending of a message) occurs in the process P_i , then the value of the logical clock is increased by 1, i.e. $C_i(e) := C_i + 1$
- If process P_i receives a message from process P_j which is notified by event e in P_i , then the counter value of the logical clock of P_i is set to the maximum of the current counter value and the counter value of the logical clock of P_j , increased by 1, i.e. $C_i(e) := \max(C_i, C_j) + 1$

After the update of the local logical clock value, the events are then assigned with the readings of the logical clocks of the processes in which they occurred. I.e. an event e that occurred in process P_i is assigned with $C_i(e)$. Through the rules, an ordering of the events is established which is consistent with the causal order in which the events occurred. Through this, the implication

$$e_k \mapsto e_l \Rightarrow C_i(e_k) < C_j(e_l) \quad (4.14)$$

is maintained. As multiple events can be assigned with the same logical clock value, only a partial order on the set of events can be established, i.e.

$$e_k \mapsto e_l \Rightarrow C_i(e_k) < C_j(e_l) \Rightarrow e_k \preceq e_l \quad (4.15)$$

Scalar logical clocks only satisfy the weak clock condition (Lamport [49]).

Vectorial logical clocks, or vector clocks for short, are an enhancement of the concept of scalar logical clocks that was developed in parallel, but independently at the end of the 1980s by Fidge [31] and Mattern [54]. Vectorial logical clocks employ vectorial time stamps for the ordering of events.

The working principle of the ordering mechanism by means of vectorial logical clocks is as follows: Each process P_i in the system is assigned a logical vectorial counter C_i whose dimension is the number of processes in the system, i.e. $C_i = (c_{i1}, c_{i2}, \dots, c_{in})^T$. The logical vectorial counter assigns vectorial integer values to events that occur in the process. The clock value is updated according to the following rules:

- If an event e_k (internal event or sending of a message) occurs in the process P_i , then only the value in the vector of P_i is increased by 1, i.e.

$$C_i(e_k) := C_i + (\delta_{i1}, \delta_{i2}, \dots, \delta_{in})^T \text{ with } \delta_{im} = \begin{cases} 1 & \text{if } m = i \\ 0 & \text{otherwise} \end{cases}.$$
- When a process P_j receives a message from another process P_i , then P_j also receives the current local clock vector of the sending process via the message. The local clock value of the receiving process is then updated as follows:

$$C_j(e_{receive}) = (\rho_{j1}, \rho_{j2}, \dots, \rho_{jn})^T$$

4 Foundations

$$\text{with } \rho_{jm} = \begin{cases} \max(c_{jm}, c_{im}) + 1 & \text{if } m = j \text{ (Fidge)} \\ c_{jm} + 1 & \text{if } m = j \text{ (Mattern)} \\ \max(c_{jm}, c_{im}) & \text{(otherwise)} \end{cases}$$

With vectorial logical clocks it is possible to determine a total order on the set of observed events. The reason for this is that such clocks satisfy the strong clock condition (Lamport [49]).

Figure 4.12 shows two space-time diagrams with three processes A , B and C each. At the marked points in time on the horizontal process lines, events occur which are either process internal (if there is no incoming or outgoing arrow) or where the processes communicate via messages (incoming/outgoing arrow). In figure 4.12(a), the events are assigned with scalar logical time stamps whereas in figure 4.12(b), the events are assigned with vectorial logical time stamps.

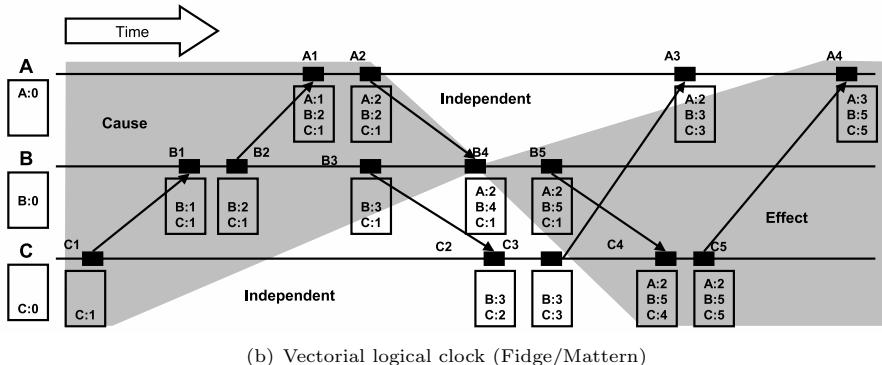
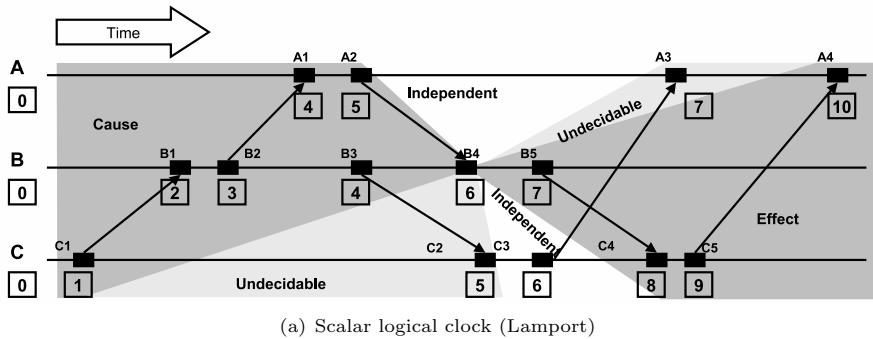


Figure 4.12: Examples for working principles of scalar and vectorial logical clocks

In each diagram, the focus is on the fourth event in process B which is marked with $B4$.

- The events in the dark gray area left of $B4$ are potential causes for $B4$, those in the dark gray area right of $B4$ its potential effects.
- The events in the white areas above and below of $B4$ are causally independent of $B4$, i.e. they are neither cause nor effect.
- An essential difference between logical scalar and vectorial clocks are the light areas present in figure 4.12(a) which are not present in figure 4.12(b). The light gray areas contain events for which it is undecidable if they are causes or effects or if they are causally independent of $B4$.

Scalar logical clocks thus only satisfy the weak clock condition as they allow the determination of a partial ordering on a set of events, vectorial logical clocks satisfy the strong clock condition as they allow the determination of total orderings on a set of events.

The implementation of logical clocks firstly requires to maintain logical clocks in terms of scalar or vectorial integer counters for each process in the computer system and secondly to attach scalar or vectorial logical time-stamps to messages that are exchanged between the processes. Both entails an implementation overhead that does not suit the constraints under that parallel and distributed embedded real-time computer systems are developed: minimal computational resources in terms of runtime and minimal computation and communication times.

Logical clocks do not show the time value of a physical clock but a logical counter value. It is thus not possible to calculate the latency between two events. Logical clocks are thus not suited for the determination of timing properties in terms of physically quantifiable time values.

Logical clocks are especially used in parallel and distributed transaction based computer systems (e.g. banking systems) such that transactions (e.g. bookings) can be executed in the correct causal order. In such systems, the time between two events is not important, but the problem is the determination of a correct order for distributed transactions.

Real Clocks

In order to determine the physical progress of time a measurement instrument is employed³ that is generally termed as a *clock*. The basic working principle of a clock still holds since the invention of mechanical clocks in the 17th century: an oscillator is employed for which the frequency is known and from which basic time units - even the base time unit second - are derived by counting a specific number of oscillations. For this, mechanical clocks employ a pendulum as an oscillator, digital clocks employ a crystal as oscillator (e.g., a Quartz crystal), and atomic clocks employ specific

³Note the peculiarity that time itself and its properties is defined via a measurement instrument.

4 Foundations

types of atoms as oscillators (e.g., the caesium ^{133}C atom). Furthermore, a display is employed which shows the *reading* of the clock at a specific point in time. The reading is expressed in terms of a *time value*. A time value consists of both a scalar value (integer or real) and a *time unit*. The scalar value determines the integer multiple of the time unit.

Digital clocks in real-time systems employ an oscillator, e.g. a quartz crystal, and a phase-locked loop (PLL) for frequency stabilization. Furthermore, a control unit is employed to release an interrupt at a specific clock reading in order to control the interaction of the real-time application with the environment it is embedded in.

One problem of all existing devices for measuring time is that they are imperfect compared to the idealized notion of time. The reading of digital clocks as they are employed in real-time systems can deviate from the ideal clock reading by several seconds per day. Rationales for this imperfection are fabrication tolerances, variations of the temperature in which the clocks are operated, variations in the voltage supply, and aging. In distributed real-time systems where the overall system consists of several subsystems, and where the subsystems must collaborate in order to fulfill the real-time applications tasks, specific clock parameters must be known in order to keep the individually working systems synchronized. The most important parameters are drift, granularity and offset.

4.2.3 A Formal Model for Real Clocks

In [55] and [56], Münzenberger et al. introduce a model of time that is based on different types of clocks. The clocks are characterized by different parameters, and the relations between the readings of the clocks is explained. The different types of clocks form a clock system in terms of a hierarchy of clocks where certain clocks act as reference clocks for others.

A common property of all types of clocks⁴ is that the value $C(t)$ that is measured by a clock at a given time t is called the reading of the clock. In the following, the different types of clocks as defined by Münzenberger et al. ([55], [56]) are described in detail.

Figure 4.13(a) depicts the relation of the different types of clocks in terms of a hierarchy of clocks. Figure 4.13(b) depicts the behavior of exemplary clocks of the different types in terms of the relation of their readings.

Ideal Clock The ideal clock is an idealization of absolute time as in Newtonian physics. The reading of the ideal clock C_{ic} is defined as identity function, i.e.,

$$C_{ic}(t) = t \tag{4.16}$$

with $t \in \mathbb{R}$. The time unit is in general one SI-second. In figure 4.13(a), the ideal clock is depicted as the basic reference clock from which all other clocks are derived. Figure 4.13(b) shows that the readings of the ideal clock resembles the identity function.

⁴One could argue that this is a property of an abstract clock.

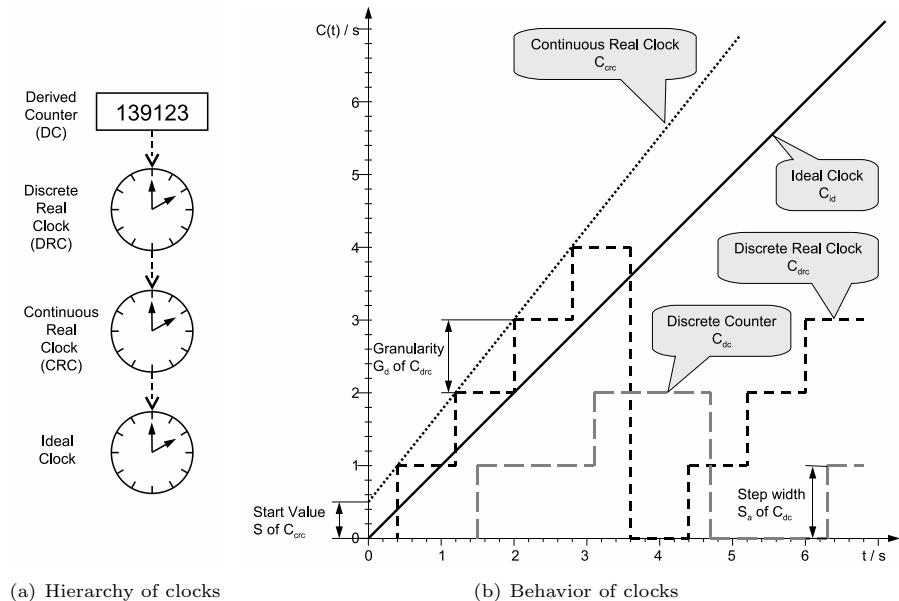


Figure 4.13: Hierarchy of different clock types and their behavior

Continuous Real Clock A continuous real clock represents a physical clock which implicitly refers to the ideal clock as reference clock. Two parameters characterize its properties compared to the ideal clock: its initial reading S at the origin of ideal time ($t = 0$) and its drift over time to express the imperfection of its physical properties. The latter is the difference of the clock speed compared to the ideal clock. The deviation is expressed in parts per million (ppm). Causes for such deviations for digital clocks are differences in fabrication of the oscillator, variations of the temperature in which the oscillator is operated, aging effects or variations in the voltage supply. The drift value D is given as the maximum possible deviation, i.e.,

$$D = \max_{\forall t} |d(t)| \quad (4.17)$$

bounds the possible deviations at all points in time. The reading of a continuous real clock C_{crc} is then determined by

$$(1 - D)t \leq C_{crc}(t) \leq (1 + D)t \quad (4.18)$$

4 Foundations

Figure 4.13(a) shows that in general the ideal clock serves as reference clock for a continuous real clock. Figure 4.13(b) shows the readings of a continuous real clock with a start value of 0.5 s and a positive drift of 250000 ppm⁵.

Discrete Real Clock A discrete real clock represents a clock where the readings lie within a discrete value domain. A discrete real clock C_{drc} is bound to a continuous real clock C_{crc} which serves as an underlying reference clock. Discrete real clocks are in general employed in technical systems, e.g., in terms of digital clocks. The resolution of the discrete value domain is given as the *granularity* of the clock. The granularity is formally captured by the parameter G_{drc} . It denotes the minimum difference between two subsequent readings of the clock. Furthermore, the value domain of the readings of a discrete real clock is in general limited by the capabilities of representing discrete values such that only values from a specific *range* can be displayed. The range of a discrete real clock is formally captured by the parameter R_{drc} . The reading of a discrete real clock at a given time t can then be determined by

$$C_{drc}(t) = \left(\left\lfloor \frac{C_{crc}(t)}{G_{drc}} \right\rfloor G_{drc} \right) \bmod R_{drc} \quad (4.19)$$

Figure 4.13(a) shows that a discrete real clock is bound to a specific continuous real clock acting as reference clock. Figure 4.13(b) depicts the readings of a discrete real clock which is bound to the previously described continuous real clock. The granularity of the discrete real clock is 1 second, and the range is 5, meaning that values between 0 and 4 are displayed.

In [68], a discrete real clock has a further parameter, the cycle period T , which replaces the parameter of the granularity at a specific place in equation 4.19. The cycle period denotes the frequency of the oscillator of the discrete real clock. Together with the granularity, it allows that the readings of the discrete real clock are integer multiples, e.g., 0, 2, 4, 6 ... for the first four readings instead of 0, 1, 2, 3 The clock reading is thus determined by

$$C_{drc}(t) = \left(\left\lfloor \frac{C_{crc}(t)}{T} \right\rfloor G_{drc} \right) \bmod R_{drc} \quad (4.20)$$

Derived Counter In real-time systems it is often necessary to implement specific functionalities in such a way that their execution rates relate to each other in a specific ratio or to derive specific tick rates from a reference clock. For this purpose, counters are introduced which are derived from a discrete real clock that acts as a reference clock. A derived counter is characterized through the following parameters:

- the *delay* $\tau \in \mathbb{R}^+$, specifying the displacement between the continuous real clock to which the reference clock is bound and the derived counter

⁵Note that the drift value has been chosen that big to visualize the effect of drift. Oscillators as employed in digital clocks have a drift rate of approximately 100 ppm or less.

- the *number of ticks* $N_{dc} \in \mathbb{N}$, specifying the number of times that the discrete real clock that acts as its reference clock has to change until the counter value is increased by one
- the *granularity* or *step width* G_{dc} , specifying the resolution of the derived counter
- the *range* $R_{dc} \in \mathbb{N}$, specifying the value domain of the counter's readings

The reading of a derived counter is then given by

$$C_{dc}(t) = \left(\left\lfloor \frac{\left\lfloor \frac{C_{crc}(t-\tau)}{G_{dc}} \right\rfloor}{N_{dc}} \right\rfloor G_{dc} \right) \bmod R_{dc} \quad (4.21)$$

Figure 4.13(a) shows that a derived clock is bound to a specific discrete real clock acting as its reference clock. Figure 4.13(b) depicts the readings of a derived counter which is bound to the previously described discrete real clock. The delay of the derived counter is 0.3 seconds, and the number of ticks of the reference clock is 2. Furthermore, the step width is 1 and the range is 3, meaning that the derived counter displays values between 0 and 2.

A clock that always shows a greater value than the ideal clock is called a *fast clock*, whereas a clock that always shows a smaller value than the ideal clock is called a *slow clock*.

4.3 AUTOSAR

4.3.1 Introduction

In 2003, the AUTomotive Open System ARchitecture (AUTOSAR) development partnership [5] has been established by several major OEM vehicle manufacturers and Tier-1 automotive suppliers. The objective of this still ongoing alliance is “to develop and establish a de-facto open industry standard for automotive E/E architecture” which serves as “a basic infrastructure for the management of functions within both future applications and standard software modules [5].”

The initiative is motivated and driven by the increasing complexity of automotive E/E systems and its development in multi-party development projects with its multiple and diverse supplier and stakeholder relationships. As all OEM vehicle manufacturers and Tier-1 automotive suppliers are facing the same challenge of mastering the development complexity, the common objective is to develop a foundational framework for automotive E/E software development. This is achieved by collaborating on the standardization of its common basic functionalities while enabling the development of innovative vehicle functions at the same time.

The results of the AUTOSAR initiative shall enable a modular and flexible realization of software-based vehicle functions in a multi-party development environment. Furthermore, they shall allow the reuse software functions across multiple products

4 Foundations

at both OEM vehicle manufacturers and Tier-1 suppliers and allow for different allocations of software functions onto ECUs.

The main achievements of the AUTOSAR initiative comprise:

Standardized, Layered ECU Software Architecture AUTOSAR proposes a standardized, layered ECU software architecture which distinguishes three major software layers: an application software layer (ASL), a middleware layer which is termed Runtime Environment (RTE), and a basic software layer (BSL). The application software is described by means of an own software component technology (see next issue). The middleware layer both conceptually and physically (i.e., in terms of source code) separates the application software from the ECU platform software infrastructure. The basic software layer forms the ECU basic software infrastructure and provides ECU specific services to the application software via the RTE. AUTOSAR defines standardized application programming interfaces (APIs) to increase inter-operability and ease exchangeability of basic software modules from different vendors. The standardized, layered ECU software architecture is further described in section 4.3.2.

Software-Component Technology The AUTOSAR software technology enables a component-based description, implementation and integration of the application software for automotive embedded functions. The software component technology is founded around the description of software components with clear interface specifications for signal exchange and service invocation. Furthermore, it allows the establishment of logical hierarchies of software components towards a system-wide software architecture. It also provides concepts for a separation of structure and behavior of software components. Together with the middleware layer, the runtime-environment (RTE), it enables a deployment-independent development of software components with the possibility of reallocation of software components within the vehicle network and their reuse across multiple ECU platforms and E/E system development projects. The AUTOSAR software component technology is further described in section 4.3.5.

Standardized Development Methodology The AUTOSAR development methodology describes how an AUTOSAR-based E/E system can be built. To support multi-party development environments, AUTOSAR defines two architecture views for the two most important development phases: the Virtual Functional Bus (VFB) view for the planning phase, and the System view for the realization phase. The development methodology is further described in section 4.3.3, and the two views are presented in section 4.3.4.

Development Approach of AUTOSAR

The AUTOSAR initiative develops its concepts using a UML2-based meta-modeling approach in order to define the relationships and semantics of its software component

technology for modeling software-based systems. For this purpose, AUTOSAR has adopted the hierarchy of meta-models corresponding to the four-layer meta-model hierarchy of the Object Management Group (OMG) [62] and extended the modeling concepts for UML2 basic class diagrams with an additional concept on the UML2 meta-model level (MOF/M3 level) [67]. Furthermore, guidelines [13] have been defined which specify how the AUTOSAR concepts in the meta-model are to be modeled.

When extensions to existing AUTOSAR concepts are defined, these need to be introduced in the AUTOSAR meta-model on the respective meta-model levels and aligned with the existing concepts. This ensures a formal specification of syntax and semantics for the new concepts and a seamless integration with existing concepts in AUTOSAR.

Organization of Concepts in the AUTOSAR Meta-Model

AUTOSAR specifies its concepts in so called *templates*. A template organizes a set of artifacts which semantically closely relate to each other. Links between artifacts that are defined in different templates naturally exist. The rationale for defining the artifacts in different templates is that conceptually they are created and used in different development phases.

In principle, six AUTOSAR templates are distinguished plus the Generic Structure package from which all templates inherit some fundamentals.

Figure 4.14 shows how the AUTOSAR templates are related to each other.

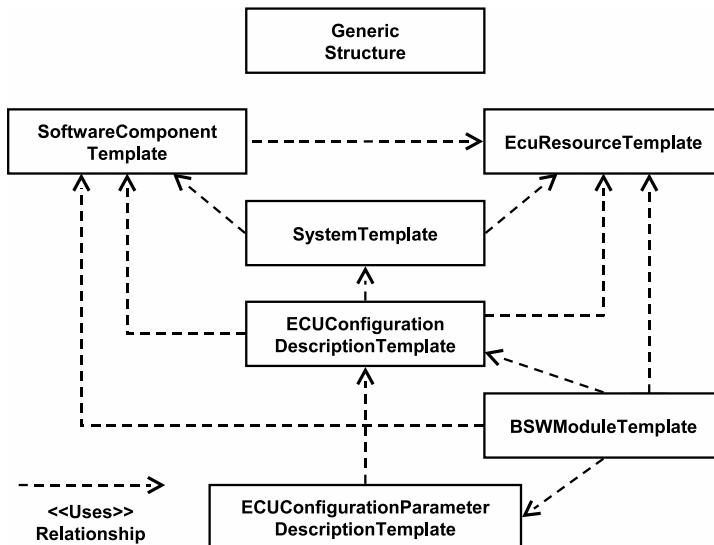


Figure 4.14: Dependencies between AUTOSAR templates

4 Foundations

The relations are established by the relationships of the contained meta-model classes for AUTOSAR elements. The relationship of the Generic Structure is not shown explicitly as the AUTOSAR elements defined there are used in all templates.

In the following, the content of the most important AUTOSAR templates for our purposes is briefly outlined:

Generic Structure The Generic Structure defines the infrastructure for AUTOSAR artifacts and templates. This comprises the definition of abstract superclasses from which all other classes for AUTOSAR artifacts inherit.

Software Component Template The Software Component Template is the most prominent AUTOSAR template. It specifies the concepts for the software component technology, including the concepts to define indivisible software components with precise interface definitions, the ability to establish component hierarchies (i.e., logical, hierarchical software structures), and the definition of the internal behavior of software components. Other concepts include the definition of modes in which software components are active, and the specification of the resource consumption of software components.

ECU Resource Template The ECU Resource Template defines the concepts to model hardware entities, i.e., ECUs, and the parts which they are made of (processing units, memories, I/O ports and pins, etc.). Especially the properties of hardware elements (e.g., voltage ranges, frequency ranges, etc.) can be described by means of the ECU Resource Template.

System Template The System Template is the central template which defines the concepts for the specification of the artifacts of a hardware system topology, including all parts such as instances of ECUs and communication clusters, the individual parts which these are made of and how they are related. Furthermore, the System Template contains the main anchor for an AUTOSAR system and defines the relations for mapping software components to computational nodes (ECUs), and how the logical communication between software components that are deployed on distinct ECUs is resolved as inter-ECU communication via system signals and network frames.

When extensions to AUTOSAR are defined, these need to be aligned with the partitioning of the AUTOSAR concepts in the different existing templates. This ensures that the newly introduced concepts are seamlessly integrated with the existing concepts and aligned with the development phases in which information from the single templates is used.

The Type/Prototype Concept and its Implications

Throughout all definitions of concepts, AUTOSAR makes exhaustive use of a specific concept called the *type/prototype concept*. The type/prototype concept is used for the definition of reusable types of software-related development artifacts and

their instantiation in concrete contexts. This has some implications on how AUTOSAR models are created, and also how references between single AUTOSAR artifacts need to be described.

All AUTOSAR elements which can be defined stand-alone represent *type* definitions. Each type definition can be instantiated in a context where the instantiated artifact is referred to as a *prototype*. The most obvious and understandable application of this concept is how hierarchies of software components are defined in AUTOSAR models: First, a set of atomic software component types need to be defined. These are self-contained and can be instantiated as so-called component prototypes in a composition type, thus defining a hierarchical component type.

In general, the distinction of whether an AUTOSAR modeling artifact denotes a type or a prototype is clear from the name of the modeling artifact: either the term “type” or “prototype” is used as a suffix, e.g., as in AtomicSoftwareComponentType or ComponentPrototype. However, the application of the type/prototype concept is not obvious for all AUTOSAR artifacts, i.e., not all AUTOSAR artifacts have such a suffix. For instance, an ECUType is referred to as ECU (i.e., without the suffix “type”) in the ECU Resource Template while they are instantiated to ECUInstances (i.e., instead of suffix “prototype”) in the SystemTemplate.

The consequent application of the type/prototype concept has some implications on how to deal with AUTOSAR modeling artifacts, especially when references are required to such artifacts. In order to refer to an AUTOSAR modeling artifact being a type, the reference concept provided by UML2 is sufficient. For this purposes, AUTOSAR specializes the UML2 association relationship to an “isOfType” (*TypeRef*) relationship. As the hierarchy of AUTOSAR artifacts can be arbitrarily deep, and as types can be used for typing prototypes multiple times in different contexts, a more powerful concept to refer to concrete instances (prototypes) of AUTOSAR artifacts is required than what is provided by UML2 class diagrams [67]. For this purpose, AUTOSAR introduced the concept of “instance references” (*InstanceRefs*) that allow to precisely denote the context in which an AUTOSAR artifact is instantiated. This allows the unambiguous identification and referencing of AUTOSAR artifacts within the definition of other AUTOSAR artifacts over all hierarchies, or in UML2 terms, part-of-parts boundaries.

When extensions to AUTOSAR are defined, the implications of the type/prototype concept need to be evaluated thoroughly, and the extension concepts potentially need to be extended to also apply the type/prototype concept. This ensures the compatibility of the newly developed concepts with the existing AUTOSAR concepts.

4.3.2 Standardized, Layered Software Architecture for Electronic Control Units

A fundamental concept of AUTOSAR is the separation of the application software from the ECU software infrastructure. This is achieved by defining a layered ECU software architecture with three main layers.

4 Foundations

Figure 4.15 depicts an overview of the layered ECU software architecture as defined by AUTOSAR [9].

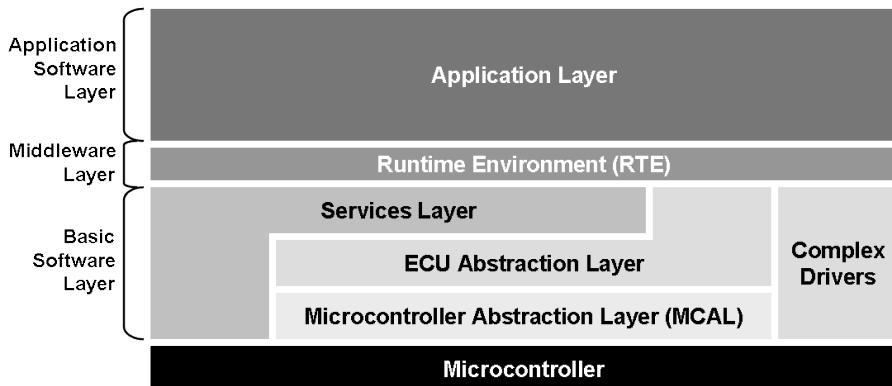


Figure 4.15: Layered ECU software architecture according to AUTOSAR [8]

The three major software layers are:

Application (Software) Layer (ASL) This layer contains application as well as sensor and actuator software components with potentially interacting behaviors which collaboratively establish a specific function or functionality.

Middleware Layer termed Runtime Environment (RTE) This middleware layer separates the application software from basic infrastructure services. This enables ECU independent development of software components as local and remote inter-component communication is routed through the RTE and thus transparent for the single software components.

Basic Software (BSW) Layer This layer contains basic software modules which provide various services (resource management, remote communication, memory access, I/O) through standardized interfaces to the application software components. This enables to abstract from the concrete hardware of the microcontroller.

The basic software layer is further divided into sublayers in which the basic software modules are organized according to their degree of hardware and platform abstraction. These sublayers are the *Services Layer*, the *ECU Abstraction Layer* and the *Microcontroller Abstraction Layer (MCAL)* (from top to bottom in figure 4.15).

4.3.3 Development Methodology

AUTOSAR defines “a common technical approach for some steps of system development [10]” which is commonly referred to as the AUTOSAR *development*

methodology. The AUTOSAR development methodology defines work products and their content as well as the activities which relate work products to each other, covering the steps which are required to come “from a system-level configuration to the generation of an ECU executable [10].” This means that in contrast to a process definition only dependencies between work products and activities are described.

In the following sections, the different phases of the AUTOSAR development methodology are briefly outlined.

Overview

AUTOSAR defines several development steps to proceed from the specification and configuration of a logical application software architecture to its implementation on the basis of concrete ECUs and vehicle bus systems. The latter provides the computation and communication infrastructure for the embedded applications. A central phase in an AUTOSAR system development is the system configuration which incorporates design decisions for software to hardware mapping and entailing data mappings. It is then possible to extract all relevant information to configure and build a single ECU. As each ECU software architecture is organized according to the AUTOSAR layered ECU software architecture, the basic software and the middleware layer provide the ECU software infrastructure for the application software.

Figure 4.16 and 4.17 depict an overview of the AUTOSAR development methodology with the major steps described in this section.

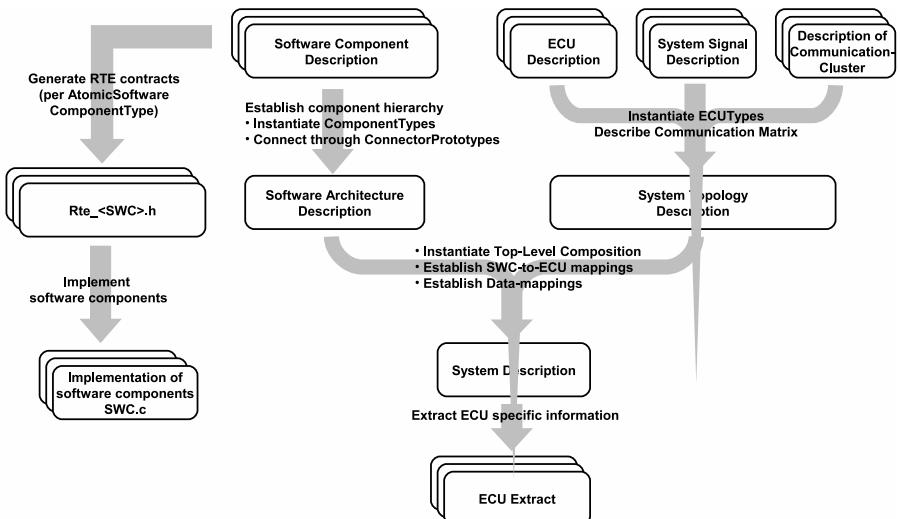


Figure 4.16: Overview of AUTOSAR methodology - Planning phase

4 Foundations

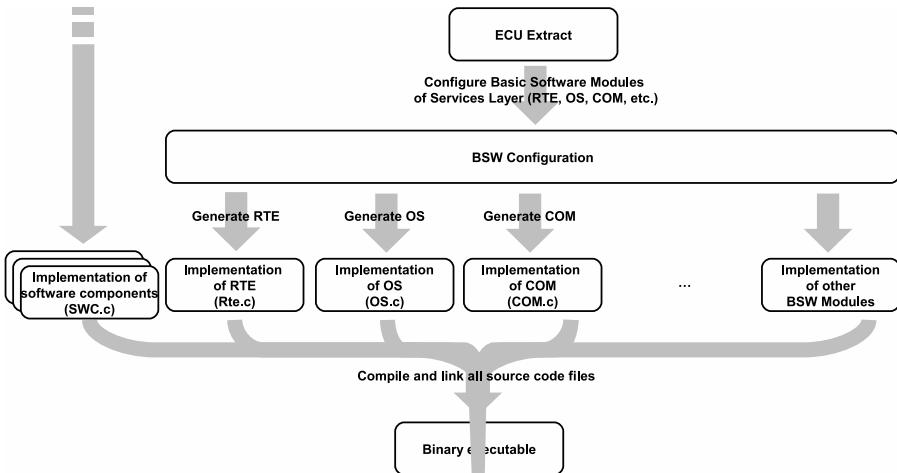


Figure 4.17: Overview of AUTOSAR methodology - Realization phase

The methodology describes the work products (e.g., AUTOSAR XML files with specific information as content, source code files generated by an AUTOSAR-compliant tool or provided by an implementor, etc.) which are required as input to conduct a single activity (i.e., a development step) and the work products which are “generated” by an activity. Activities are often supported by special purpose tools referred to as “guidelines”, e.g. an *authoring tool* for the specification and design of software components and software architectures, or *generation tools* for the OS, RTE or parts of the COM stack (basic software generators).

In the following, the main steps of the AUTOSAR development methodology are discussed. These are the specification of the logical software architecture and the hardware system topology, the system configuration procedure, and the ECU configuration and build process.

Software Architecture and System Topology Description

The first steps when building an AUTOSAR-based E/E system comprises the specification of a software architecture for the applications of the AUTOSAR system. This is achieved by defining software component types, their interfaces and ports, and by instantiating these in the context of higher order software component types (compositions) to establish a software hierarchy. The top of this software hierarchy forms the logical software architecture of the E/E system and is termed *top-level composition*. Figure 4.18 depicts an example software architecture specification for a simple application.

The means to describe a logical software architecture are defined in the Software Component Template and are described in more detail in section 4.3.5.

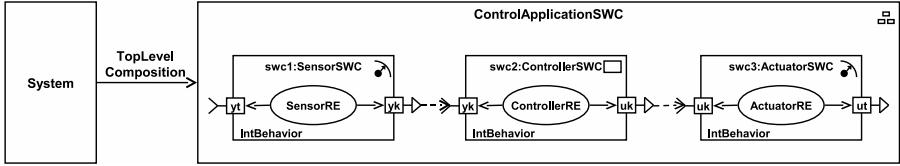


Figure 4.18: Example logical software architecture

In parallel to the specification of the logical software architecture, the hardware system topology on which the application software shall be executed must be specified. This is performed by using the concepts defined in the ECU Resource Template and the System Template (Topology part). Figure 4.19 depicts an example system topology description for a simple E/E system.

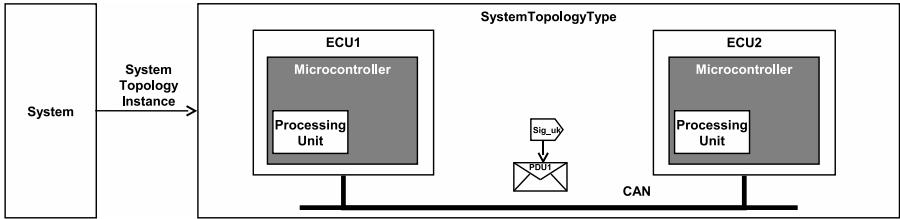


Figure 4.19: Example system topology

Note that in the first development phase, the software architecture and the hardware topology are developed independently from each other, enabling a hardware-software co-design.

System Configuration (Mappings)

There are several design decisions that need to be taken at the transition from the logical software architecture and a system topology specification to a technical realization of an AUTOSAR system. These are constituted by two so-called *mappings*:

1. The mapping of software components to ECUs: This is basically an assignment of which code structures (i.e., instances of software components) and executable entities operating on these code structures (i.e., the contained threads of control) are to be executed in which computational node (ECU). These mappings are referred to as *SWC-to-ECU-mappings*.

The mapping of software components to ECUs entails further mappings such as the mapping of data elements of the logical communication between two software components to system signals being send over the vehicle networks. For two communicating threads of control of two distinct atomic software components, this results

4 Foundations

either in intra-ECU or inter-ECU communication. Conceptually, there are two kinds of mappings:

2. The mapping of data elements (Sender/Receiver communication) and operation arguments (Client/Server communication) to system signals (primitive data types) and system signal groups (complex data types such as arrays and records). These mappings are generalized as *data-mappings*.

Implicitly, these mappings need to be established in sequence: First, software component instances need to be mapped to concrete ECU instances to decide upon intra-ECU or inter-ECU communication, and then the communicated data elements and operation arguments need to be mapped to system signals.

Figure 4.20 depicts the mapping decisions for the example AUTOSAR system.

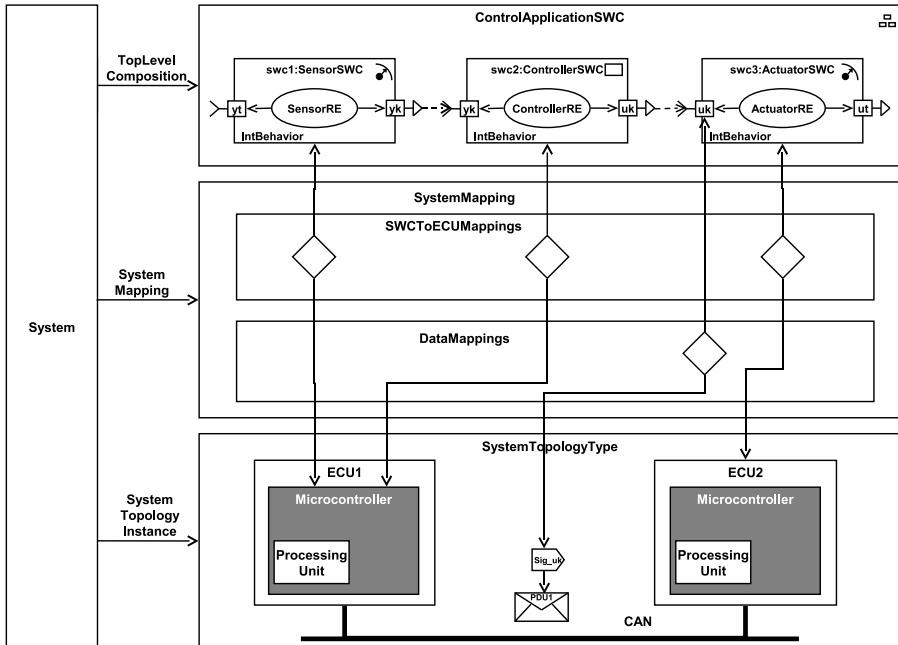


Figure 4.20: Mapping of logical software architecture to system topology: SWC-to-ECU and data mappings

The two mappings, the *SWC-to-ECU mapping* and the *data mappings*, then result in a set of so-called *ECU extracts* of the system topology (see figure 4.16). An ECU extract describes the ECU-specific extract of the logical software architecture and the data mappings that are relevant for an ECU system.

In order to obtain a technical realization of an AUTOSAR system, further configurations need to be performed for each ECU extract. These are described in the next section.

ECU Configuration and Software Build

Per ECU, so-called basic software configurations (*BSW configurations*) need to be established for each basic software module (*BSW module*) used in the technical implementation of an AUTOSAR ECU system (see figure 4.17).

In the following, we briefly describe the configurations for the most important basic software modules: the operating system (OS), the runtime environment (RTE), and the communication stack (COM).

Operating System (OS) To configure the operating system of an AUTOSAR-compliant ECU system, operating system objects such as OS-tasks, interrupt-service routines (ISRs), etc. need to be defined. This also includes the specification of their configuration parameters which decide upon their scheduling. For an OS-task, this for example includes the priority, the number of possible activations, etc. Other parameters such as when OS-tasks need to be activated are automatically determined by the RTE generator based on the ECU configuration information of which RunnableEntities are assigned to which OS-tasks.

Runtime Environment (RTE) The configuration of the RTE of an AUTOSAR-compliant ECU system comprises specific general configurations which influence the generation of the runtime environment. Important configurations include

- the assignment of the RunnableEntities of the mapped AtomicSoftware-Component instances to OS-tasks
- the assignment of data elements (Sender/Receiver communication) and operation arguments (Client/Server communication) to COM signals (see BSW module COM in the AUTOSAR layered ECU software architecture)⁶

Other configurations include the enabling/disabling of the generation of so-called VFB trace hooks which can be used to monitor all interaction of the application software with the RTE. This is important for our Timing Model for AUTOSAR and its concrete application for specification-based performance analysis by means of monitoring or simulation.

Communication Stack (COM) In order to configure the communication stack of an AUTOSAR-compliant ECU system, several BSW modules need to be configured, depending on the type of the bus system used. AUTOSAR specifies BSW modules that allow remote communication over vehicle bus networks

⁶ Alternatively, this information can be extracted from the SystemTemplate as well where it is also contained (redundant information in AUTOSAR models).

4 Foundations

using the CAN, Flexray, MOST or LIN protocol. Depending on the type of network, a large number of configuration parameters need to be specified. This is not further described in this thesis. Further information can be found in the AUTOSAR specification documents on the communication stack [5].

The basic software configurations are work products which are required as input to generate the source code of the basic software modules. Code generation for the basic software modules is performed by module- and potentially microcontroller-specific basic software generation tools, e.g., an RTE generator [25].

4.3.4 Virtual Functional Bus View and System View

Closely related to the development methodology, AUTOSAR defines two perspectives or *views* on an AUTOSAR system: a logical perspective and a technical, i.e. implementation-specific, perspective. The logical perspective is termed the *Virtual Functional Bus (VFB) view*, the technical perspective does not have a definite term of its own but is often referred to as *System view*.

The main difference between both perspectives is that in the logical software architecture perspective (VFB view), specific information which is available in the technical software architecture perspective (System view) is not present. This kind of information is the result of design decisions taken at the transition from the logical software architecture to the technical software architecture, and these decisions are constituted by the mappings and configurations described in section 4.3.3. The views thus also implicitly relate to different development phases.

According to the AUTOSAR meta-model specification [6], in the VFB view, no information which relates to the deployment of software components to ECUs is available. This means that all mapping-dependent information needs to be abstracted away from the technical perspective of an AUTOSAR system, only leaving a limited set of AUTOSAR concepts to describe the logical software architecture of an AUTOSAR system and the software entities therein. The VFB view of an AUTOSAR system serves two purposes:

1. To describe individual AtomicSoftwareComponentTypes such that a C code contract can be established between the specifier of the software component and the implementor.
2. To ensure that the software components which are used in a software architecture can successfully be integrated without failures due to problems with the interfaces of the software structures (i.e., interface problems are addressed). This requires that the interfaces of the ports of connected software components are compatible to ensure that the structural interfaces on the C code level are compliant.

The first issue requires all AUTOSAR concepts of the Software Component Template to describe an individual AtomicSoftwareComponentType. This also includes the InternalBehavior of AtomicSoftwareComponentTypes as without that information, no C code contract can be generated.

The second issue requires the AUTOSAR concepts which allow to define a software component hierarchy. These are explained in section 4.3.5.

The VFB view thus refers to an AUTOSAR system only consisting of its logical software architecture. Figure 4.18 shows the VFB view of our AUTOSAR system example. The System view refers to an AUTOSAR system consisting of its logical software architecture being mapped onto a hardware system topology, including all ECU-specific configurations. For our example AUTOSAR system, this corresponds to what is shown in figure 4.20.

4.3.5 Concepts of the Software Component Technology

The application software of an AUTOSAR system is described by means of the concepts of the software component technology defined by AUTOSAR. As outlined in section 4.3.1, the AUTOSAR concepts make exhaustive use of the *type/prototype concept*. In the following, the core concepts of the AUTOSAR software component technology are described with an explicit consideration of the type/prototype concept. Further information can be found in the Software Component Template specification [11].

Definition of Software Structures

In order to define the structure of a real-time application that is to be realized in an AUTOSAR system, the AUTOSAR software component technology provides concepts. Figure 4.21 shows an overview of the concepts. These are described in more detail in the following.

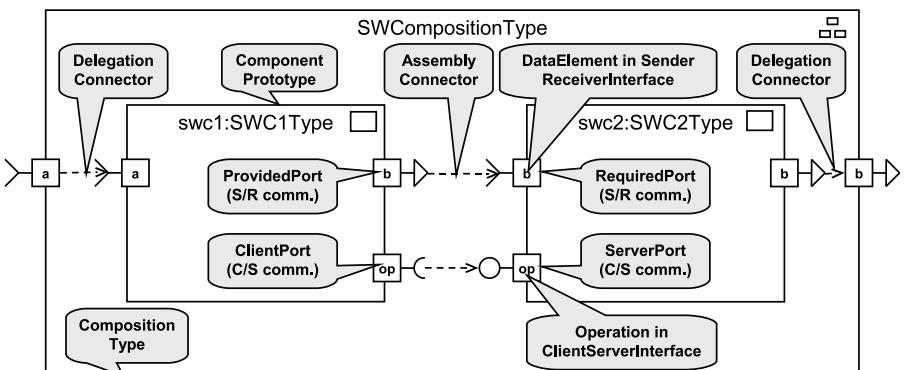


Figure 4.21: Components, Ports and Interfaces, and Connectors

4 Foundations

Components Software components in AUTOSAR are first defined as stand-alone *ComponentTypes* and then used in logical, hierarchical structures as *ComponentPrototypes*. The two most important ComponentTypes are the *AtomicSoftwareComponentType* and the *CompositionType*. Other ComponentTypes are the *ServiceComponentType* and the *SensorActuatorSoftwareComponentType* (both specializations of *AtomicSoftwareComponentType*), and the *CalPrmComponentType*⁷.

An *AtomicSoftwareComponentType* denotes a structural software entity which cannot be further decomposed with respect to the deployment to a computational node of an E/E system. In contrast to that, the *CompositionType* denotes a logical *ComponentType* which contains instances of other *ComponentTypes* and the connections that establish communication relations between them. An instance of a *ComponentType* in the context of a *CompositionType* is termed *ComponentPrototype*. As the *ComponentPrototype* references its *ComponentType*, logical hierarchies of software structures can be established.

It is important to note that the *CompositionTypes* are purely logical containers for *ComponentPrototypes* and the connections between them. *CompositionTypes* are thus a means which allow the structuring of a software architecture. When it comes to the implementation in software, they do not have any representation in code and thus do not bear any implementation or runtime overhead.

Figure A.1 in Appendix A shows the relevant excerpt from the AUTOSAR metamodel for components.

Ports and Interfaces To allow software components to communicate with each other, AUTOSAR specifies ports which are typed by an interface. In AUTOSAR terminology, this translates to *ComponentTypes* specifying one or more *PortPrototypes* which are typed by a *PortInterface*⁸. The specification of such *PortPrototypes* is part of the definition of a *ComponentType*.

There are two specializations for a *PortPrototype*: the *RPortPrototype* (or *RPort*, for short) and the *PPortPrototype* (*PPort*). An *RPortPrototype* is a requiring port of a software component. A *PPortPrototype* is a providing of a software component.

AUTOSAR supports two kinds of communication paradigms for the communication between software components:

- Direct and unidirectional signal exchange from a sender software component to a receiver software component. This is referred to as *Sender/Receiver communication* in AUTOSAR.
- Bidirectional service invocation of a client software component at a server software component. This is termed *Client/Server communication* in AUTOSAR.

In the first case, the respective ports of the communicating software components need to be typed by a *SenderReceiverInterface*, in the second case they need to be typed

⁷a *ComponentType* for handling calibration parameters

⁸The type/prototype concept is also consequently applied for *PortPrototypes* and *PortInterfaces*, although this is not obvious from the AUTOSAR specification. A *PortInterface* can be interpreted as a type specification for a *PortPrototype*, i.e. a *PortPrototype* instantiates a *PortInterface*.

by a *ClientServerInterface*. In the AUTOSAR meta-model, both are specializations of the abstractly defined *PortInterface*.

A *SenderReceiverInterface* is a collection of so-called *DataElementPrototypes* (*DataElements*, for short). A *DataElementPrototype* is a named element which is typed by a primitive or complex *DataType*.

A *ClientServerInterface* is a collection of *OperationPrototypes* (*Operations*, for short). Each *OperationPrototype* has itself an ordered list of *ArgumentPrototypes* (*Arguments*) which can have either the directions *in*, *inout* or *out*. Similar to *DataElementPrototypes*, *ArgumentPrototypes* are typed by *DataTypes*.

By means of *PortPrototypes* and *PortInterfaces*, the structural interface for AUTOSAR software components can be specified. Figures A.2 and A.3 in Appendix A show the relevant excerpts from the AUTOSAR meta-model for ports and interfaces as well as data types.

Connectors The communication between software components is expressed via *ConnectorPrototypes*. These are defined in the context of a *CompositionType* and establish the connections between the *ComponentPrototypes* by referring to the two *PortPrototypes* which are involved in the communication. Due to the fact that *CompositionTypes* are pure logical structuring means, and due to the fact that AUTOSAR allows the construction of hierarchical software architectures by means of component type definition and instantiation, two kinds of *ConnectorPrototypes* are distinguished: *AssemblyConnectorPrototypes* and *DelegationConnectorPrototypes*.

To establish a *Sender/Receiver* communication between two software components, an *AssemblyConnectorPrototype* is used. Similarly, to establish a *Client/Server* communication between two software components, an *AssemblyConnectorPrototype* is used to specify the connection between a *RPortPrototype* of a *ComponentPrototype* which acts as a client and a *PPortPrototype* which acts as a server.

In order to delegate communication data through the logical component hierarchy, *DelegationConnectorPrototypes* are used. A *DelegationConnectorPrototype* establishes a connection between a *PortPrototype* of the containing *CompositionType* and a *PortPrototype* of a contained *ComponentPrototype*. Two kinds of communication delegation can be distinguished: input delegation and output delegation.

To allow for two *PortPrototypes* being connected by a *ConnectorPrototype*, the *PortInterfaces* of the *PortPrototypes* must be *compatible*. This means that both *PortInterfaces* must be of the same kind (i.e., either a *SenderReceiverInterface* or a *ClientServerInterface*). When an *AssemblyConnectorPrototype* establishes a connection between a *PPortPrototype* and a *RPortPrototype*, the *SenderReceiverInterface* of the *PPortPrototype* must contain the same or more *DataElementPrototypes* as the *SenderReceiverInterface* of the *RPortPrototype*. In the case of a *DelegationConnectorPrototype*, the *PortInterfaces* must be exactly compatible (i.e., same *DataElementPrototypes* in the case of a *SenderReceiverInterface* or same *OperationPrototypes*, including *ArgumentPrototypes*, in the case of a *ClientServerInterface*). These rules are referred to as *interface compatibility rules* and must be obeyed when constructing software hierarchies and architectures with AUTOSAR software components.

4 Foundations

Figure A.4 in Appendix A shows the relevant excerpts from the AUTOSAR meta-model for connectors.

Definition of the Behavior of Software Components

So far, only the structure of software components which mainly comprises the interfaces and the data therein has been considered. In order to perform computations on this data, behavioral entities which have access to the interfaces of the software components need to be considered. AUTOSAR conceptually separates the description of the structural part of a software component from the description of its internal behavior. In this section, the behavioral definition of an AtomicSoftwareComponent is described and discussed. Figure 4.22 shows an overview of the concepts for the specification of the so-called *InternalBehavior* of an AtomicSoftwareComponent.

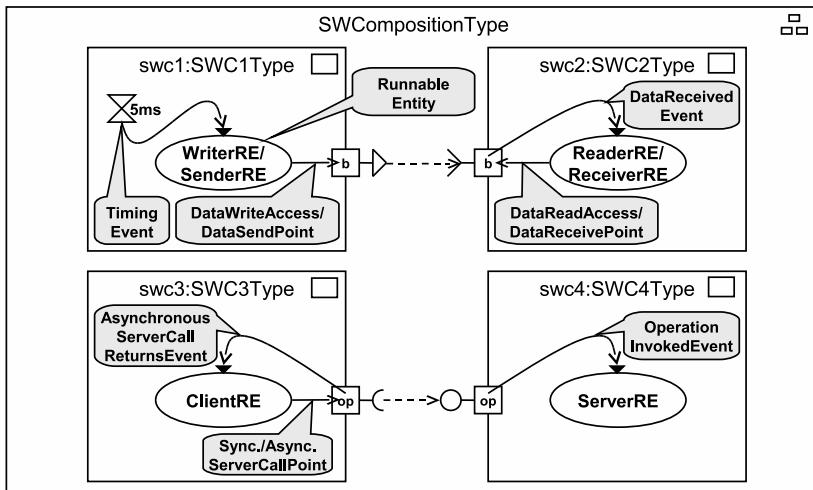


Figure 4.22: RunnableEntities, Declaration of Data Accesses and Service Provisions/Needs, RTEEvents

As described in section 4.3.5, an AtomicSoftwareComponentType is an indivisible software structure with respect to the deployment to a computational node in the vehicle E/E system. In contrast to CompositionTypes, AtomicSoftwareComponentTypes (as well as SensorActuatorSoftwareComponentTypes) are the software structures which have a direct counterpart in a software implementation in terms of source code or binary data. In terms of code, each instance of an AtomicSoftwareComponentType is represented by a data structure which is termed *Component (Instance) Data Structure*, or CDS (CIDS) for short. The term Instance in CIDS is due to the fact that an AtomicSoftwareComponentType can be instantiated multiple

times (if specified so) and deployed on the same ECU. Thus, to distinguish between multiple instances, an instance handle is generally required in the source code.

RunnableEntities The *InternalBehavior* of an *AtomicSoftwareComponentType* describes the behavioral entities and their properties which implement the algorithms of an application. These entities are software functions which operate on the software structure of an *AtomicSoftwareComponentType*, the CIDS. In AUTOSAR, these entities are referred to as *RunnableEntities*, or *Runnables* for short.

Note that a single *AtomicSoftwareComponentType* can contain more than one *RunnableEntity* in its *InternalBehavior*. Figure A.5 in Appendix A shows the relevant excerpts from the AUTOSAR meta-model for *InternalBehavior* and *RunnableEntities*.

Declaration of Data Accesses and Service Provisions/Needs The communication between the *RunnableEntities* of *AtomicSoftwareComponents* is established via the RTE of an AUTOSAR ECU system. In fact, the RTE manages the CIDS of all instantiated components on an ECU and provides an API which must be used by the *RunnableEntities* to read/write data or initiate/respond to service invocations. As the RTE is generated for each ECU, and as its implementation depends upon the deployed software structures, the software components must declare how they intend to interact with the RTE.

In case of Sender/Receiver communication, a *RunnableEntity* can declare read and/or write access to the *DataElementPrototypes* of the *SenderReceiverInterfaces* of the *PortPrototypes*. In case of Client/Server communication, a *RunnableEntity* can declare that it implements a server response which is provided as an *OperationPrototype*, or that it intends to call a service which is required as an *OperationPrototype*.

For the two communication paradigms (i.e., Sender/Receiver communication and Client/Server communication), different specializations of the data accesses do exist. These are further discussed in section 4.3.6 in general and under the specific aspects of the VFB view and the System view.

Note that in principle, multiple *RunnableEntities* can specify access to the same *DataElementPrototype* (reading or writing). Furthermore, the specifications of the data accesses are completely independent from implementation aspects such as when and how often a *RunnableEntity* performs a read or write action, calls or provides a service.

Activation of RunnableEntities through RTEEvents Another important concept are *RTEEvents* which specify how a *RunnableEntity* should be invoked at runtime, or when it shall be activated as a response to an event which occurs within an AUTOSAR system. It is the responsibility of the RTE of an AUTOSAR ECU system to manage the triggering events for *RunnableEntities* and to adequately activate the respective *RunnableEntities*.

AUTOSAR defines eight *RTEEvents* for different kinds of activation semantics of a *RunnableEntity*. The events can be classified as follows:

4 Foundations

Category	RTEEvent	Semantics
General	TimingEvent	... at a specified periodic rate
Sender/Receiver communication	DataReceived-Event	... upon reception of a new value in a referenced DataElementPrototype
	DataSentEvent	... upon successful completion of sending a new value in a referenced DataElementPrototype
	DataReceived-ErrorEvent	... upon an error in the reception of a new value in a referenced DataElementPrototype
Client/Server communication	Operation-InvokedEvent	... upon the invocation of a referenced OperationPrototype
	Asynchronous-ServerCall>ReturnsEvent	... upon the return of an asynchronous server call
Mode Management	ModeSwitch-Event	... upon the initiation of mode switch in the ECU state manager
	ModeSwitch-AckEvent	... upon the acknowledgement of a successful mode switch by the ECU state manager

Table 4.1: Overview of RTEEvents

Note that in principle, multiple RTEEvents can be specified for a RunnableEntity. Figure A.6 shows the relevant excerpts from the AUTOSAR meta-model for RTEEvents.

Other concepts: InterRunnableVariables, ExclusiveAreas In the Internal-Behavior, so-called *InterRunnableVariables* can be declared which are used for the communication between RunnableEntities belonging to the same AtomicSoftware-ComponentType. RunnableEntities can themselves declare reading or writing access to these InterRunnableVariables. Inter-Runnable communication can be considered as a special form of Sender/Receiver communication where the two communicating RunnableEntities belong to the InternalBehavior of the same AtomicSoftwareComponentType. Inter-Runnable communication is further described in section 4.3.6.

Another concepts are *ExclusiveAreas* which denote shared resources within an atomic software component can be used to prevent from multiple RunnableEntities accessing the same code at the same time.

Implementation Each InternalBehavior of an AtomicSoftwareComponentType can have multiple different *Implementations*. During the ECU configuration, exactly one implementation must be chosen for each component instance deployed on the ECU. The Implementation gives information about which source or object code files implement an AtomicSoftwareComponentType and an associated InternalBehavior, and on which files or libraries the Implementation depends. If object code is referred, the certain properties of the *Compiler* (name, vendor, version) used to build the binary software can be described to as well.

4.3.6 Communication Patterns

AUTOSAR distinguishes three major communication patterns to specify the communication between the behavioral entities of software components. These communication patterns are the Sender/Receiver communication pattern, the Client/Server communication pattern and the Inter-Runnable communication pattern. The third communication pattern can be considered as a special kind of Sender/Receiver communication which allows direct communication between RunnableEntities belonging to the same AtomicSoftwareComponentType.

Figure 4.23 depicts an example for Sender/Receiver communication, Client/Server communication and Inter-Runnable communication between two RunnableEntities of two (respectively one) software components.

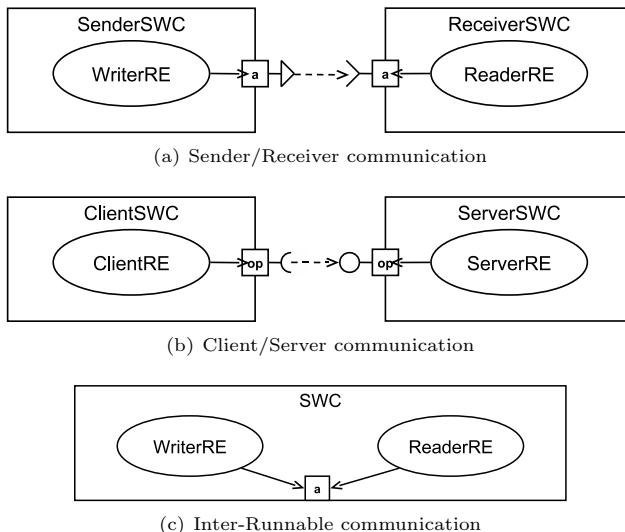


Figure 4.23: Examples for Sender/Receiver, Client/Server and Inter-Runnable communication

For each of the communication patterns, further specializations exist, e.g., for how the RunnableEntities access the DataElementPrototypes on the ports of the software components they belong to. This has a fundamental influence on when a communication is established and becomes effective and thus, it is relevant for timing.

In the following sections, the communication patterns and their occurrences are described and discussed in detail. Section 4.3.6 presents the general notion of the communication patterns and explains under which circumstances they are valid at

4 Foundations

all⁹. With respect to the AUTOSAR development methodology and the two views (see section 4.3.4), the communication patterns need to be considered under these different views as well. The reason is that these two views reflect different levels of detail about an AUTOSAR software architecture at different phases of a development (i.e., planning phase vs. integration and realization phase). Section 4.3.6 discusses these differences.

Communication Patterns in General

In the following, the general properties of the communication patterns are discussed. The relevant excerpt from the AUTOSAR meta-model is shown in figure A.7 in Appendix A.

Sender/Receiver Communication

The Sender/Receiver communication pattern (see example in figure 4.23(a)) is used to asynchronously distribute information from a sender component to one or more receiver components (1:n communication)¹⁰. The communication is asynchronous because the providing actions of the SenderSWC and the requiring actions of the ReaderSWC are not synchronized and completely independent from each other.

In the **SenderSWC**, the sending of the data element **a** is performed by the **WriterRE** which performs a write action onto data element **a**. In the **ReceiverSWC**, the reception of the data element **a** is achieved by the **ReaderRE** which performs a read action onto data element **a**. The specification of the read and write actions are denoted by the arrows from the RunnableEntities to the DataElementPrototypes in the Port-Prototypes.

AUTOSAR distinguishes two specializations of the Sender/Receiver communication pattern: *implicit* Sender/Receiver communication and *explicit* Sender/Receiver communication. Explicit Sender/Receiver communication is further distinguished into queued communication with so-called *event semantics* and unqueued communication with so-called *data semantics*. The general distinction of these specializations lies in how data consistency is achieved between multiple communicating Runnable-Entities in an implementation. Note that for Sender/Receiver communication with data semantics, the provider is termed *writer* while the requester is termed *reader*; for Sender/Receiver communication with event semantics, the provider is termed *sender* while the requester is termed *receiver*.

Explicit Sender/Receiver Communication In explicit Sender/Receiver communication, the RunnableEntity which specifies an explicit read or write action to

⁹For example, not all possible combinations of independently specified data accesses in a Sender/Receiver communications are valid. Specific combinations are not allowed and lead to invalid AUTOSAR models.

¹⁰Since AUTOSAR R3.0, n:1 Sender/Receiver communication (i.e., multiple senders and one receiver) is also allowed when the semantics of the communication is that of queued communication.

a DataElementPrototype has *direct* access to the globally visible data value in the CIDS that is managed by the RTE. In principle, each explicit access is different, meaning that for two consecutive explicit read accesses of a RunnableEntity, two different values can be retrieved, depending on the fact whether the data value managed by the RTE has changed between these accesses.

In explicit Sender/Receiver communication, an explicit read action is specified by a *DataReceivePoint* in the InternalBehavior of an AtomicSoftwareComponentType. A DataReceivePoint links a RunnableEntity that intends to perform an explicit read action to a DataElementPrototype in a RPortPrototype of the AtomicSoftwareComponentType. Analogously, an explicit write action is specified by a *DataSendPoint* which establishes a link between a DataElementPrototype in a PPortPrototype and a RunnableEntity.

Implicit Sender/Receiver Communication In implicit Sender/Receiver communication, the RunnableEntity which specifies an implicit read or write action to a DataElementPrototype has *indirect* access to the globally visible data value which is managed by the RTE. The access is indirect as the RunnableEntity operates on a data value which cannot be manipulated by other RunnableEntities during its runtime. This is achieved by providing a *copy* of the data value managed by the RTE within the context of the OS-task in which the RunnableEntity is executed.

In implicit Sender/Receiver communication, an implicit read action is specified by a *DataReadAccess* in the InternalBehavior of an AtomicSoftwareComponentType. A DataReadAccess links a RunnableEntity that intends to perform an implicit read action to a DataElementPrototype in a RPortPrototype of the AtomicSoftwareComponentType. Analogously, a implicit write action is specified by a *DataWriteAccess* which establishes a link between a DataElementPrototype in a PPortPrototype and a RunnableEntity.

Possible Occurrences of Sender/Receiver Communication The occurrence of a Sender/Receiver communication, i.e., how Sender/Receiver communication is established between two communication partners, depends on how the data access is specified for the WriterRE of the the SenderSWC and for the ReaderRE of the ReceiverSWC. Both read and write actions can be specified either according to the *implicit* or the *explicit* communication pattern. This also means that explicit and implicit communication can be mixed to some extent as the single communication partners are independent from each other. Furthermore, in the case of explicit communication, the communication can be either specified as queued communication (event semantics) or unqueued communication (data-semantics).

Table 4.2 depicts a matrix with the possible communication pattern occurrences of the Sender/Receiver communication pattern based on the example described above.

- For implicit communication only data-semantics (i.e. unqueued communication) is defined. Thus, no communication with event-semantics is shown for implicit communication in table 4.2.
- In the case of explicit Sender/Receiver communication, the specification of the communication between the SenderSWC and the ReceiverSWC as either being

		ReceiverRE		
		Implicit Data	Explicit Data	Event
SenderRE	Implicit	✓	✓	—
	Explicit	—	—	✓
	Data	✓	✓	—

✓ = combination allowed
— = combination not allowed

Table 4.2: Possible occurrences of Sender/Receiver communication pattern

queued (event-semantics) or unqueued (data-semantics) must be consistent for both the SenderSWC and the ReceiverSWC. Thus, those combinations where event-semantics is specified for the SenderSWC and data-semantics is specified for the ReceiverSWC and vice versa are invalid.

Inter-Runnable Communication

The Inter-Runnable communication pattern (see example in figure 4.23(c)) is used to asynchronously communicate information between two RunnableEntities of the same AtomicSoftwareComponent. The communication is asynchronous because write actions and read actions are not synchronized and completely independent from each other.

In the example, the WriterRE can perform a write action onto InterRunnableVariable *a*. The ReaderRE can perform a read action onto the same InterRunnableVariable *a*.

A RunnableEntity can declare a read or write access to an InterRunnableVariable by pointing to it through a reference. This is denoted by the arrows in figure 4.23(c).

Possible Occurrences of Inter-Runnable Communication AUTOSAR distinguishes two specializations of the Inter-Runnable communication pattern: implicit Inter-Runnable communication and explicit Inter-Runnable communication. Note that only Inter-Runnable communication with data semantics is specified, no event semantics and thus no communication of queued data. In contrast to Sender/Receiver communication where the providing and requesting actions in a Sender/Receiver communication with data semantics can be any combination of explicit and implicit communication actions, in Inter-Runnable communication, both the WriterRE and the ReaderRE must specify their access method to the InterRunnableVariable according to the same pattern.

Table 4.3 shows a matrix with the possible communication pattern occurrences of Inter-Runnable communication based on the example described above.

		ReaderRE	
		Implicit Data	Explicit Data
WriterRE	Implicit Data	✓	—
	Explicit Data	—	✓

✓ = combination allowed

— = combination not allowed

Table 4.3: Possible occurrences of Inter-Runnable communication pattern

Client/Server Communication

The Client/Server communication pattern (see example in 4.23(b)) implements a bidirectional request/response protocol where a client software component (more precisely, a client RunnableEntity) requests a service from a server component (more precisely, a server RunnableEntity).

The communication is bidirectional as the ClientRE of the ClientSWC first invokes the ServerRE of the ServerSWC which then performs some computational actions and returns the result to the ClientSWC. In the ClientSWC, the invocation of the service is performed by invoking a specific RTE access macro onto the OperationPrototype op. In an implementation, the RTE is then responsible to dispatch the client request (including ArgumentPrototypes) and activate the respective ServerRE via a special RTEEvent, an OperationInvokedEvent (see section 4.3.5). The ServerRE can then process the request with the given arguments. The RTE finally delivers the results to the ClientSWC which can process the results. Note that the fetching and processing of the result is performed differently in synchronous and asynchronous Client/Server communication.

Furthermore,

- multiple clients can be connected to one server,
- clients can make an arbitrary number of service requests to the same server,
- requests are queued on the server side and processed in first-in-first-out (FIFO) order.

Client/Server communication is further distinguished into synchronous and asynchronous Client/Server communication. In synchronous Client/Server communication, the client invoking the service is blocked until either a response has been received from the server, an infrastructure error is returned or a timeout occurs. In asynchronous Client/Server communication, the client invoking the service is either blocked or not blocked, depending on the configuration how the server result shall be processed. In either case, the infrastructure activates the respective RunnableEntity on the client side which is then responsible for collecting the result.

In Client/Server communication, a *ServerCallPoint* needs to be specified in the InternalBehavior of the ClientSWC. A ServerCallPoint establishes a link between

4 Foundations

an OperationPrototype in a RPortPrototype and the RunnableEntity that intends to call the operation. Note that the ServerCallPoint must be either a *SynchronousServerCallPoint* or an *AsynchronousServerCallPoint*, depending on the kind of Client/Server communication. On the server side, an OperationInvokedEvent must be specified in the InternalBehavior of the ServerSWC such that the respective RunnableEntity which implements the server operation is called by the RTE when the operation is invoked by the ClientRE.

Possible Occurrences of Client/Server Communication Table 4.4 depicts a matrix with the possible communication pattern occurrences of the Client/Server communication pattern based on the example described above.

		ServerRE		
		Synchronous	Asynchronous	
		Blocking	Blocking	Non-Blocking
ClientRE	Sync.	Blocking	✓	—
	Async.	Blocking	—	✓
		Non-Blocking	—	✓

✓ = combination allowed
— = combination not allowed

Table 4.4: Possible occurrences of Client/Server communication pattern

Communication Patterns in Different Views

In this section, the differences between the communication patterns with respect to the VFB view and the System view are discussed.

Communication Patterns on Virtual Functional Bus View As defined by the AUTOSAR development methodology, no mapping specific information is present under the perspective of the logical software architecture, i.e., the VFB view. This especially implies that

- no SWC-to-ECU mappings and thus also no data-mappings are defined. This means that intra-ECU communication cannot be distinguished from inter-ECU communication in the logical software architecture.
- no OS-tasks are defined. This means that intra-task communication cannot be distinguished from inter-task communication in the logical software architecture.

Due to these facts, the communication patterns considered on the VFB view can also only be perceived as being *logical*. In principle, a distinction between implicit and explicit Sender/Receiver or Inter-Runnable communication cannot be made due to the missing underlying concepts for how these communication patterns are implemented,

especially if certain data-consistency mechanisms such as copies are employed in an implementation or not.

Communication Patterns on System View On System view, all mapping specific information such as the SWC-to-ECU mappings and the data-mappings are available (see section 4.3.3). Furthermore, in order to build an AUTOSAR ECU software system, configurations such as assignment of RunnableEntities to OS-tasks and the configuration of the basic software modules (i.e. OS, RTE, COM etc.) are required and are thus available. As the full detail of information is available, the communication patterns can be considered in their full details with all their effects.

Conclusions for Communication Patterns in Different Views Figure 4.24 (see next page) shows the influences of the decisions taken during system configuration and ECU basic software configurations on the realization of the communication patterns in an AUTOSAR system implementation. Note that these must be evaluated for every single communication that is established between two RunnableEntities.

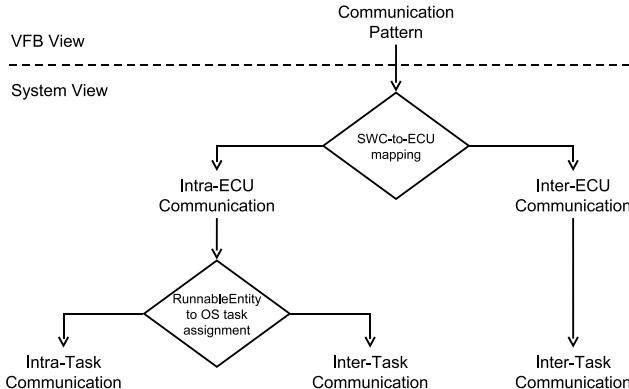


Figure 4.24: Classification of the different realizations of the communication patterns in concrete AUTOSAR implementations

Depending on the SWC-to-ECU mapping and the entailing data-mappings, two cases must be distinguished: inter-ECU communication and intra-ECU communication. In intra-ECU communication, two further cases must be distinguished depending on the assignment of RunnableEntities to OS-tasks: intra-task communication and inter-task communication. Especially for the latter case (inter-task communication), AUTOSAR provides specific data consistency mechanisms directly built into the communication patterns of implicit Sender/Receiver and implicit Inter-Runnable communication. For the latter, further cases can be distinguished which highly depend on the configuration information of the operating system. Influencing factors are the assignment of RunnableEntities to OS-tasks, the order of RunnableEntities within OS-tasks, and the properties of the OS-tasks such as their priority, etc.

4 Foundations

Furthermore, the following two issues make it difficult to consistently describe a signal path from a system input at a sensor to a system output at an actuator:

- The establishment of a single communication between two RunnableEntities requires the configuration of the respective actions of the two communication partners (e.g., a RunnableEntity sending a DataElementPrototype and another RunnableEntity receiving the same DataElementPrototype). These configurations are independent of each other. This allows a large variety of possible occurrences of communication patterns (e.g, implicit send action and explicit receive action). The main issue is large degree of freedom allowed by AUTOSARfor specifying communication between software components and the complexity of the AUTOSAR concepts.
- The actual realization of a single communication in terms of an implementation in the source code of an AUTOSAR system depends on decisions such as the system configuration (decides upon inter-ECU or intra-ECU communication) and the ECU basic software configuration (decides upon intra-task communication or inter-task communication). This information is, however, not available in the planning phase, i.e. on the VFB view, when the software architecture and the communication patterns are specified. The main issue is the

4.3.7 Virtual Functional Bus Tracing Mechanism

In [4], AUTOSAR defines a mechanism which enables developers of AUTOSAR systems to monitor the execution of a running AUTOSAR system. This mechanism is referred to as the *Virtual Functional Bus (VFB) Tracing Mechanism*. Its implementation is specified as a lightweight instrumentation of the RTE of an AUTOSAR-compliant ECU system. The Virtual Functional Bus Tracing Mechanism “is implemented by a series of “hook” functions that are invoked automatically by the generated RTE when “interesting events” occur. Each hook function corresponds to a single event [4].”

Trace Events

The events and actions which can be monitored by the VFB Tracing Mechanism are referred to as *VFB Trace Events*. Note that in contrast to the distinction made between instantaneous events and durational actions as in existing related work (Klar et al. [48], Münzenberger [69]; see also section 4.2.1), AUTOSAR uses the term VFB Trace Event to denote both kinds of entities. VFB Trace Events are atomic in the sense as they refer to a single place or action that can be monitored by means of hook functions provided by the VFB Tracing Mechanism.

The VFB Trace Events events have direct counterparts in the source code of the RTE of an AUTOSAR-compliant ECU system. These counterparts are the APIs which are offered by the RTE to the application software components in order to

operate on data that is managed by the RTE. Due to their resemblance with ECU software code, the events can be associated with a precise semantics.

AUTOSAR defines four different types of VFB Trace Events: *RTE API events*, *RunnableEntity events*, *COM events* and *OS events*. In the following, these concrete VFB Trace Events are described. Figure 4.25 shows an overview of the VFB Trace Events in the architecture of an AUTOSAR system.

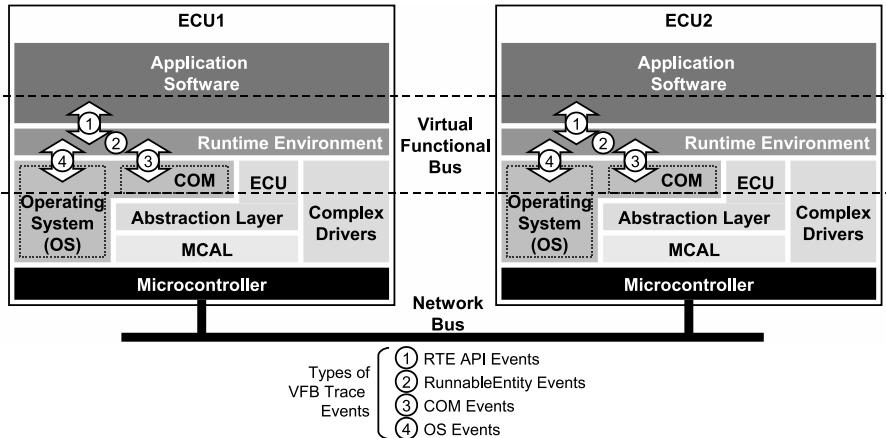


Figure 4.25: Overview of the VFB Trace Events in the architecture of an AUTOSAR system

Runtime Environment (RTE) API Events

“RTE API trace events occur when an AUTOSAR software-component interacts with the generated RTE API [4].” By means of the VFB Tracing Mechanism and the RTE API events, all interaction of software components with the RTE can thus be monitored. For each RTE API function, the VFB Tracing Mechanism defines two hook functions, a *start hook function* and a *return hook function*. The start hook function is automatically invoked when a RunnableEntity makes a call to a specific RTE API function. The return hook function is automatically invoked just before an RTE API function call returns control to a RunnableEntity.

The rationale for having two hook functions for what is meant to be a single event (i.e., the occurrence of an RTE API call event) is that an RTE API function implements an action that is composed of one or more instructions in the code. The execution of the latter does take a specific amount of time. The start hook function thus marks the start of the RTE API call action, i.e. the start event occurring at the instant just prior to the invocation of an RTE API call. The return hook function marks the end of the RTE API call action, i.e. the end event occurring at the instant just after the return of the RTE API call to its calling context.

4 Foundations

The API of the VFB Trace hooks for RTE API Events is defined in line with the RTE API itself. I.e., a VFB Trace hook function has the same base signature as the corresponding RTE API function, however, with slight modifications. It starts with the pattern `Rte_<API>Hook...` where `<API>` denotes an infix for the respective communication pattern, e.g. `Write` for an explicit send action or `IRead` for implicit read action. To unambiguously identify the start hook from the return hook, the VFB Trace hook functions are either suffixed with `...Start` or `...Return`.

Runnable Entity Events

The VFB Tracing Mechanism defines two hook functions per RunnableEntity which are invoked either just before the start of a RunnableEntity, thus denoting the start of a RunnableEntity, or immediately after the termination of a RunnableEntity, thus denoting the termination of a RunnableEntity. By means of these two hook functions, the life cycle of a RunnableEntity can be monitored.

The signatures of the two hook functions take the form `Rte_Runnable_<SWCType>_<RE>_Start()` for the start hook function, and `Rte_Runnable_<SWCType>_<RE>_Return()` for the return hook function. The infix `SWCType` denotes the AtomicSoftwareComponentType which contains the RunnableEntity `RE`. Different instances of the same ComponentType are distinguished with an instance handle that is given as a parameter to the hook function. This is not shown above.

Communication (COM) Events

The VFB Tracing Mechanism defines several hook functions to monitor the interaction of software components which are deployed onto distinct ECUs. The hook functions resemble events corresponding to the transmission and reception of signals and signal groups via the remote communication services that are abstracted by the RTE and implemented by the COM stack. The hook functions which enable the monitoring of remote communication are hook functions at the interface of the RTE to the COM module.

In order to monitor the successful transmission and reception of a signal or signal group between two ECUs it is sufficient to monitor the events when the signal or signal group is sent on the sending ECU side, and when the signal or signal group is received on the receiving ECU side. Other aspects such as signal invalidation or acknowledgement both on sender or receiver ECU side belong to the remote communication protocol realized by the COM stacks and are not of interest for our purposes.

Table 4.5 provides an overview of the relevant COM hook functions for signal and signal group communication.

A detailed description of the hook functions and the sequence of actions and events to be monitored for a successful signal or signal group transmission between two ECUs is provided in Appendix D.

	Sender ECU	Receiver ECU
System Signals	Rte_ComHook_<signalname>.SigTx	Rte_ComHook_<signalname>.SigRx
System Signal Groups	Rte_ComHook_<signalgroup>.SigTx	Rte_ComHook_<signalgroup>.SigRx

Table 4.5: VFB Tracing COM hook functions for signal and signal group communication

Operating System (OS) Events

Within an AUTOSAR ECU system, the OS is responsible for managing all resources and for invoking the defined tasks according to their configuration. The OS also provides the mechanisms and services to run an RTE-based AUTOSAR system as the RTE employs the mechanisms of the OS to activate the respective RunnableEntities. To monitor the interaction of the OS with the RTE, the VFB Tracing Mechanism defines hook functions which are called

- immediately prior to the activation of an OS-task by the RTE: This hook function thus resembles the event when an OS-task is being activated.
- when an OS-task which is dispatched by the RTE starts execution: This hook function thus resembles the event when an OS-task is being dispatched.
- immediately before an OS-task attempts to send an OS-event: This hook function thus resembles the event when an OS-task sends an OS-event.
- when an OS-task attempts to wait for an OS-event: This hook function thus resembles the event when an OS-task is waiting for an OS-event.

Figure 4.26 depicts the state transition diagrams for basic tasks and extended tasks as defined in OSEK/VDX OS [66] and AUTOSAR OS [7].

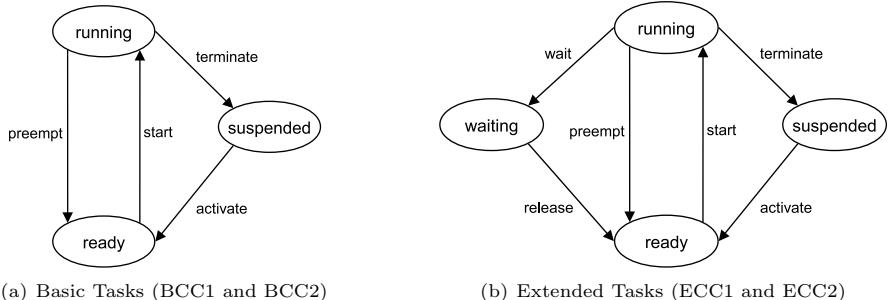


Figure 4.26: State transition diagrams for tasks in OSEK/VDX and AUTOSAR compliant RTOS

4 Foundations

The OS-specific hook functions provided by the VFB Tracing Mechanism are thus invoked at transitions activate (1) and start (2) in the state transition model. Note that the VFB Tracing Mechanism does not support a hook function which is invoked at termination of a task or when a task is preempted.

4.4 Summary and Overview of Work

4.4.1 Summary to Control Theory

The detailed review of control theory in section 4.1 has shown that discrete-time control applications are inherent real-time applications which assume that certain timing properties are continuously maintained during their runtime. These timing properties are all oriented on one or more important feedback paths within the control application such that timing requirements can only be adequately formulated with the help of a precise signal path description.

Table 4.6 summarizes the timing requirements and their formal representation together with a description of their origin.

Timing Requirement	Formal Representation	Origin
Equidistant input sampling	$h_{sampling} \equiv const$	Discretization of continuous-time signals
Equidistant output production	$h_{actuation} \equiv const$	Analogy to first timing requirement
Minimization of feedback path delay	$\tau_{SCA} \equiv 0$	Minimization of deteriorating effects of control delays
Maintenance of constant feedback path delay	$\tau_{SCA} \equiv const$	Ease of compensation of inevitable control delays
Synchronization of sampling actions	$\forall i \in \{1..m\} : d_{Sensor(i) \rightarrow Controller} \leq d_{inputsync}^{\max}$	Temporal consistent and synchronized measurement signal acquisition for control signal computation
Synchronization of actuation actions	$\forall j \in \{1..n\} : d_{Controller \rightarrow Actuator(j)} \leq d_{outputsync}^{\max}$	Temporal consistent and synchronized effectuation of control signals

Table 4.6: Timing requirements for discrete-time, linear time-invariant control applications

The work in this thesis targets the description of these timing requirements for control applications that are realized as part of an AUTOSAR system. The main prerequisite for a description of these timing requirements is the unambiguous description of signal paths in AUTOSAR systems such that the timing requirement specifications can refer to them. For control applications, it is important to describe the feedback path such that timing requirements can be adequately expressed.

4.4.2 Summary to Timing Models

The foundations of timing models for parallel and distributed computer systems are based on the notion of events and actions that can be observed during the runtime of the computer system. This can be used in simulation- or monitoring-based approaches to evaluate the performance of the computer system and to determine timing properties. For the latter, several aspects need to be considered:

- Instances of events and actions can be ordered according to the temporal order relation and the causal order relation. While a causal order implies a temporal order on a set of instances of events, the inverse implication does not hold. This is important for the determination of timing properties based on sets of events as these only make sense for the case where the temporal order on the set of events is consistent with the causal order in which they occurred. Logical clocks such as scalar or vector clocks can be used to determine a partial or total logical temporal order on the set of event instances. However, logical clocks cannot be used to determine timing properties. It is thus required to use real-time clocks for the provision of time stamps for instances of events and actions. These clocks must be synchronized in an adequate (see work of Hofmann [43]) way such safe statements about temporal distances between instances of events that occurred in different computational nodes can be made.
- A formal notion of time is required in order to firstly determine safe statements about the temporal order of instances of events and actions that occur in the computer system and that can be observed, and secondly to have a definite temporal context to which timing requirements can refer to. A formal notion of time can be gained through the precise characterization of clocks which act as a measurement device for time. Clocks can be formally characterized by means of concepts that have been introduced in related work by Münzenberger et al. [69].

Timing properties can be determined by means of simulation or monitoring-based approaches through the observation of the dynamic runtime behavior of a computer system. Simulation-based approaches are applicable already during the development in order to evaluate and plan the timing behavior. Monitoring-based approaches can be employed when the target system or a prototype thereof is available in order to validate the planned and expected timing behavior.

4.4.3 Summary to AUTOSAR

The AUTOSAR initiative targets a rigorous approach for automotive E/E systems development through the provision of standardized concepts in several aspects:

- a standardized, layered ECU software architecture with standardized APIs for the basic software modules and a middleware-layer, the RTE, which separates the application software from the platform software,

4 Foundations

- a standardized software component technology which allows the specification of the structure and behavior of embedded applications,
- a methodology which describes how AUTOSAR-compliant systems are built.

AUTOSAR provides a comprehensive infrastructure with standardized concepts in all relevant areas in order to successfully implement automotive embedded applications. Technically, an AUTOSAR system can be perceived as potentially distributed embedded real-time computer system. The algorithms of the real-time applications, e.g. control applications, are realized by the application software of an AUTOSAR system. How such applications are designed, i.e., how they are structured in terms of software components, what communication patterns are employed, and how the individual behavioral entities are triggered, is however left to the developer of the application. There is thus a large degree of freedom with numerous possibilities how to design an application and how to configure an AUTOSAR system. On the other hand, all these design and configuration decisions are relevant for the application-specific timing properties of an AUTOSAR system.

Many automotive applications are real-time applications, e.g. control applications, which are associated with application-specific timing requirements. As AUTOSAR abstracts from concrete applications being realized as AUTOSAR systems, AUTOSAR has not considered the different kinds of timing requirements so far.

The requirements towards a Timing Model for AUTOSAR are that it must be possible

- to precisely specify application-specific timing requirements, especially those of control applications
- to determine timing properties that correspond to the timing requirements such that the degree of fulfillment of the latter can be evaluated.

For real-time applications that have timing requirements which are oriented on signal paths, AUTOSAR does not provide an adequate concept for the precise description of the latter. Consequently, it is not possible to adequately describe application-specific timing requirements.

With the Virtual Function Bus Tracing Mechanism, AUTOSAR provides a means to monitor interesting events that occur during the runtime of an AUTOSAR system.

The Virtual Function Bus Tracing Mechanism, however, has originally not been developed for a monitoring- or simulation based performance analysis of real-time applications realized in an AUTOSAR system. Although it is not described in the AUTOSAR specification documents, it can be assumed that the Virtual Function Bus Tracing Mechanism should provide a means to measure the data that is communicated over the ports of software components in the value domain (i.e., not in the time domain). The Virtual Function Bus Trace Events, however, offer a suitable and sufficient means for a behavioral abstraction of an AUTOSAR system such that a monitoring- or simulation-based performance analysis can be conducted. This is

the principal foundation of the approach presented in this thesis for the integration of specification-based performance analysis into the AUTOSAR standard.

While AUTOSAR thus already defines a set of possible events occurring within an AUTOSAR-based ECU system [4], AUTOSAR does not reflect these events within its meta-model specification. Consequently, AUTOSAR does not provide a concept to describe relations among events, e.g. in order to describe a signal path. They are also not part of one of the templates which are used during development of an AUTOSAR system. Application-specific timing requirements, especially those of control applications, refer to one or more signal paths from system inputs to system-outputs. It is thus required to precisely specify signal paths in AUTOSAR systems. For this, two or more events need to be considered in relation to each other. The latter, however, is not supported by the Virtual Function Bus Tracing Mechanism. Developers of AUTOSAR systems can thus not express application-specific timing requirements by means of these already existing events.

4.4.4 Overview of Work

Figure 4.27 provides an overview of the work presented in this thesis towards a Timing Model for AUTOSAR.

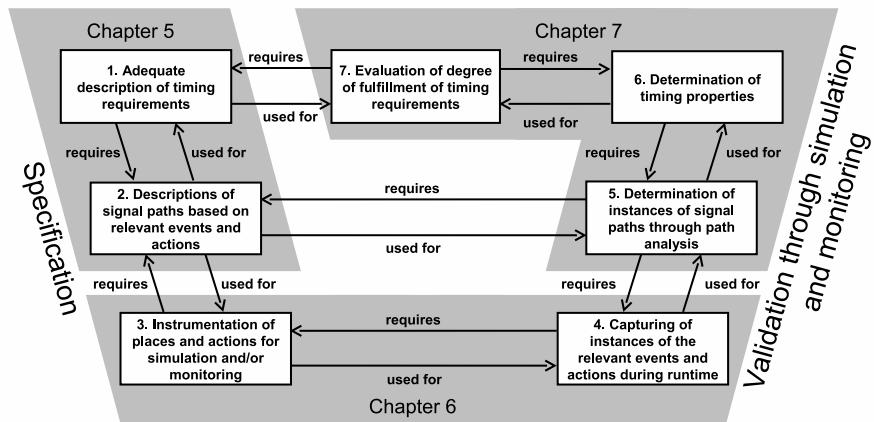


Figure 4.27: Overview of the work presented in this thesis towards a the Timing Model for AUTOSAR

The figure resembles the V-development cycle which is widely employed in the automotive industry. However, the focus of this thesis is not on the development of automotive embedded systems itself. The objective is to provide concepts for the

4 Foundations

adequate specification application-specific timing requirements (1) and the determination of corresponding timing properties by means of simulation- and monitoring-based approaches (6) such that the degree of fulfillment of the timing requirements can be evaluated (7).

The adequate description of application-specific timing requirements (1), especially those of control applications, requires concepts for the description of signal paths (2). Concepts for these two aspects are introduced in chapter 5. As simulation- and monitoring-based approaches shall be employed for the determination of timing properties, the description of signal paths must be based on relevant events and actions that lie on the signal path. From such signal path descriptions, an instrumentation of the specific places where instances of events and actions can be observed can be derived (3). The instrumentation can then be used for the capturing of instances of events and actions during the runtime of the system in simulation- and monitoring-based approaches (4). The necessary steps to obtain an instrumentation is described in chapter 6. Based on the captured instances of relevant events and actions, instances of signal paths can be determined through the analysis of temporal and causal relations between instances of events and actions (5). Based on the instances of signal paths, timing properties can be determined (6) such that the degree of fulfillment of the timing requirements can be evaluated (7). The concepts that finally enable the latter are introduced and described in chapter 7.

4.4.5 An Integrated Example

Throughout the thesis, we employ the example of a single-input-single-output (SISO) control application that is realized in terms of an AUTOSAR system. Figure 4.28 provides an overview of the considered generic SISO control application. The timing requirements of the SISO control application are informally annotated to the feedback path between the sensor and the actuator.

The SISO control application is then realized in terms of an AUTOSAR system as shown in figure 4.29. The logical software architecture of the AUTOSAR system consists of three AtomicSoftwareComponents, **SensorSWC**, **ControllerSWC** and **ActuatorSWC**, which each contain a single RunnableEntity. Each AtomicSoftwareComponent and its contained RunnableEntity realizes one of the three processes **Sensor**, **Controller** and **Actuator**. Note that we assume that the evaluation of the error signal $e(k)$ based on the setpoint signal $w(k)$ and the input signal $y(k)$ is performed by the controller processes and thus included in the **ControllerRE**. Each RunnableEntity is triggered by a TimingEvent. The periods of the TimingEvents are not fixed but part of different configuration scenarios discussed in the thesis.

For the example AUTOSAR system of the SISO control application, we

- describe the application-specific timing requirements based on formal path specifications (chapter 5)
- explain the generation of an instrumentation for simulation- or monitoring-based determination of timing properties (chapter 6)

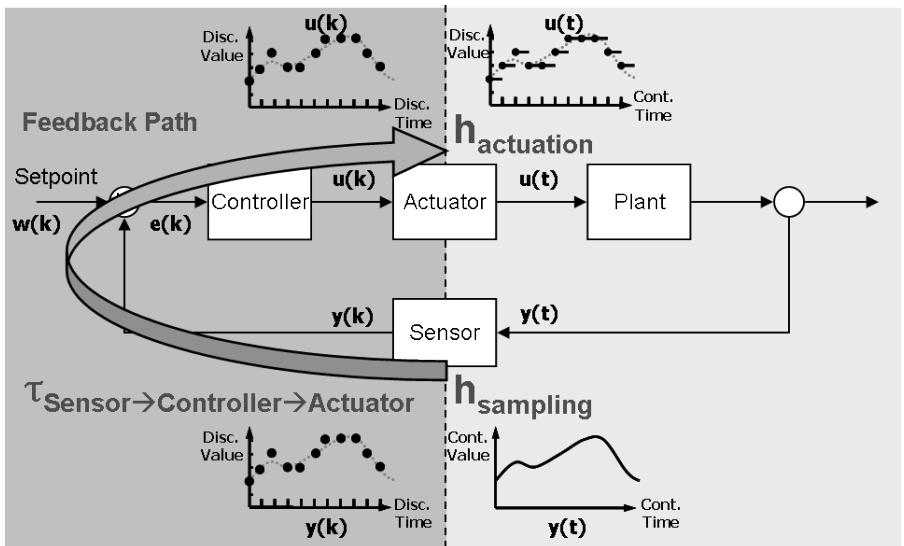


Figure 4.28: Example closed-loop single-input-single-output (SISO) control application with timing requirements associated with the feedback path

- explain the determination of timing properties and their adequate interpretation in the form of runtime-oriented diagrams (chapter 7)

Informally, we assume that the following timing requirements are given:

- The feedback path delay from sensor data acquisition to actuator output data effectuation should be negligible and constant, i.e. $\tau_{SCA} = 0$ ms. A deviation of 2 ms is acceptable.
- The effective sampling rate at the sensor should be constant as well, i.e. $h_{sampling} = 10$ ms. A deviation of 1 ms is acceptable.
- Furthermore, the effective actuation rate at the actuator should be constant, i.e. $h_{actuation} = 10$ ms. A deviation of 1 ms is acceptable again.
- If multiple sensors are employed for the acquisition of input signals from the same physical quantity, the data acquisition actions should be synchronized within an interval of 2 ms.

4 Foundations

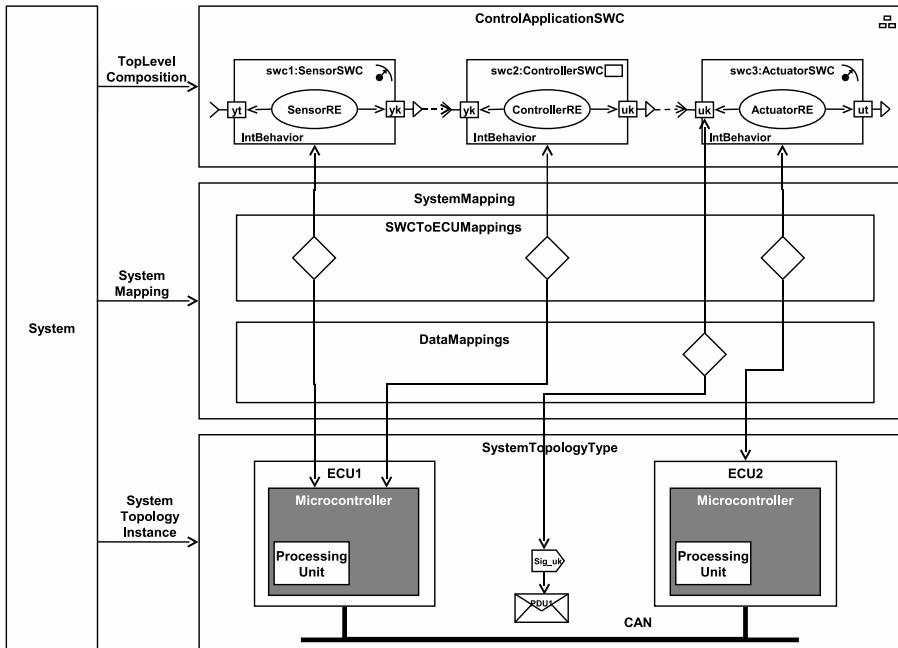


Figure 4.29: Example AUTOSAR system for a simple single-input-single-output (SISO) control application

5 Specification of Timing Requirements based on Formal Path Specifications

5.1 Introduction

In this chapter, the necessary concepts for the adequate description of timing requirements for real-time applications based on signal path specification are introduced. The concepts are described for systems built according to the AUTOSAR standard. The underlying principles, however, are valid also in a broader sense and can, with some effort, be adapted to other approaches.

In order to adequately specify timing requirements for real-time applications, especially those of control applications as described in section 4.1, it is required that descriptions of signal paths are provided to which the timing requirements can refer. For the determination of their corresponding timing properties, event-based approaches such as discrete-event simulation or monitoring are employed. These approaches require that relevant events and actions in a system are identified such that their instances can be observed in a simulation or by means of monitoring during the runtime of the system. Consequently, to enable the determination of timing properties by means of such approaches, signal path descriptions must be based on events and actions that lie on the signal path such that it is clear what needs to be monitored. Such a signal path description can be perceived as a chain of events and actions that pairwise stand in a direct causal relation. For the construction of such an event chain, however, it is first necessary to identify what events and actions can be observed in a system at all.

As described in section 4.3.7, the Virtual Functional Bus Tracing Mechanism provides a lightweight instrumentation of the RTE of an AUTOSAR ECU system to monitor the interaction of software components with the RTE at runtime and to monitor the occurrence of other interesting events. The so-called Virtual Functional Bus Trace Events, however, do not have a counterpart in the AUTOSAR meta-model yet. Thus, it is not possible to use these Trace Events as a means to specify points of interest. Consequently, they can also not yet be used to specify any causal relations among them, e.g., in order to precisely describe a signal path.

In the following, the Virtual Functional Bus Trace Events are formally introduced into the AUTOSAR meta-model (section 5.2) such that AUTOSAR-specific event and action classes can be marked as points of interest in an AUTOSAR system.

In order to give instances of event and action classes a definite context, a formal notion of time is introduced into AUTOSAR (section 5.3). This is achieved through the adaptation of the formal clock model concepts of Münzenberger et al. [56]. This enables the characterization of the behavior of any real-time clock in an AUTOSAR

system as well as the binding of the AUTOSAR-specific event and action classes in the different development phases.

Then, a concept is introduced to define order relations between these AUTOSAR-specific event and action classes (section 5.4). The introduced concept allows the concatenation of event and action classes that can be identified in the logical software architecture of an AUTOSAR system to event chains in a hierarchical way. This consequently enables the precise description of event chains for the top-level component of an AUTOSAR system which finally, when constructed correctly, denote signal path specifications (section 5.5).

In order to describe application-specific timing requirements, especially those of control applications, concepts are introduced that allow to express them based on the formal signal path descriptions (section 5.6).

Section 5.7 describes the integration of the previously described concepts with other AUTOSAR aspects such as the Virtual Functional Bus view and the System view, the development methodology and the Templates defined by AUTOSAR.

Throughout the chapter, we provide examples based on the simple control application example to demonstrate the practical applicability of the concepts of our Timing Model for AUTOSAR. These describe how AUTOSAR-specific event and action classes are modeled, how they are related to each other to event chains and signal path specifications, respectively, and how the latter are associated with application-specific timing requirements.

As described in section 4.3.1, the AUTOSAR initiative develops its concepts using a UML2-based meta-modeling approach. The meta-model specifications are important as they represent specifications of AUTOSAR concepts on the basis of a formal semantics (UML2 class diagrams). They can be used as a source for tool developers to implement the defined AUTOSAR concepts. The AUTOSAR meta-model is complemented by textual descriptions in the different specification documents. To introduce the concepts for the description of timing requirements based on formal path specifications, we adapt the UML2 meta-modeling approach. Especially, we provide meta-model specifications for AUTOSAR-specific event and action classes that resemble the Virtual Functional Bus Trace Events. Furthermore, we provide meta-model specifications for the concepts to relate these to event chains. To keep the definition of the new concepts succinct, their meta-model specifications and their integration with the existing AUTOSAR meta-model specifications can be found in Appendix E.

5.2 Events and Actions

The basis for the description of signal paths in an AUTOSAR system is the identification of relevant event and action classes that are part of the signal path, i.e. that lie on the signal path, and which stand in a direct causal relation. As described in section 4.3.7 the Virtual Functional Bus Tracing Mechanism defined by AUTOSAR provides adequate means to observe relevant events during the runtime

of an AUTOSAR system. Furthermore, the Virtual Functional Bus Tracing Mechanism also implicitly defines what kinds of Trace Events can be observed. These are RunnableEntity events, RTE API events, COM events and OS events.

The Virtual Functional Bus Trace Events are in principal all AUTOSAR-specific specializations of event classes. This especially holds for RunnableEntity events, OS events and COM events as there is one corresponding Virtual Functional Bus Trace hook function for the monitoring of each such Trace Event. RTE API events always go pairwise for each RTE API function: one event class marks the start of the action (corresponding to the start hook function) and one event class marks the end of the action (corresponding to the return hook function). Consequently, to monitor the interaction of software components with the RTE of an AUTOSAR ECU system, we introduce the notion of RTE API *actions* which are an AUTOSAR-specific specialization of the general concept of action classes as described in the foundations of timing models (section 4.2). An RTE API action implicitly refers to the two event classes that mark the start and end of an RTE API function call as their start event and end event.

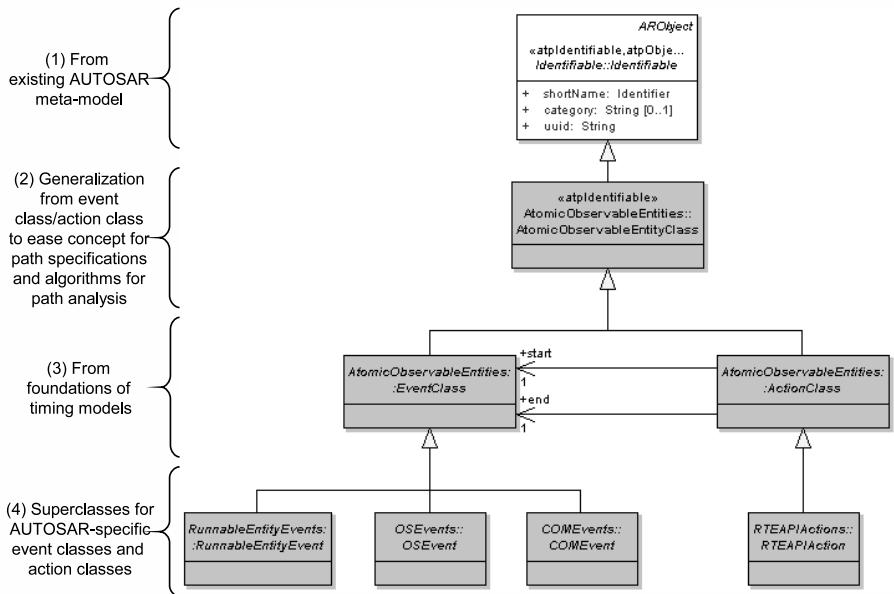


Figure 5.1: Meta-model specification for event and action classes in AUTOSAR

In order to better integrate the AUTOSAR-specific event and action classes with the existing AUTOSAR concepts, we extend the meta-model for event and action classes from the foundations of timing models (see section 4.2). Figure 5.1 depicts the extended meta-model specification for event and action classes in AUTOSAR.

which is based on the core concepts from the foundations of timing models (3). As both event and action classes are perceived as observable entities which are atomic, we introduce the notion of an abstract observable entity class (class `AbstractObservableEntityClass`) which is an abstract superclass for `EventClass` and `ActionClass` (2). This entails several benefits for our further work: Firstly, for the construction of signal paths it allows the description of an order on both event and action classes at the same time in an elegant way as these can both be referred to by referencing their common superclass. Secondly, the algorithms that we later introduce for the determination of instances of signal paths are less complex as the different treatment of instances of event and action classes only needs to be considered once. The abstract superclass `AtomicObservableEntityClass` inherits from the AUTOSAR class `Identifiable` (1). This means that any AUTOSAR-specific event or action class can be identified via its AUTOSAR shortname in an AUTOSAR system description¹. From the abstract classes `EventClass` and `ActionClass`, different superclasses are derived: `RunnableEntityEvent`, `OSEvent` and `COMEvent` and `RTEAPIAction` (4). Under these, the concrete AUTOSAR-specific event and action classes are further organized. These are described in the following.

5.2.1 Runnable Entity Events

As described in section 4.3.7, according to the Virtual Functional Bus Tracing Mechanism, there are two concrete events that can be observed with respect to a `RunnableEntity`. These are the events corresponding to the start and termination of a `RunnableEntity`. In our Timing Model for AUTOSAR, these are defined as follows:

Definition 12

- A ***RunnableEntityStartEvent*** is an event class that marks the *start* of a `RunnableEntity`.
- A ***RunnableEntityEndEvent*** is an event class that marks the *end* of a `RunnableEntity`.

The meta-model specification for `RunnableEntityEvents` can be found in Appendix E.2.

Example: Figure 5.2 depicts an example `AtomicSoftwareComponentType` with `RunnableEntityEvent` specifications for the contained `RunnableEntity`. In the graphical representation, the `RunnableEntityEvents` which are specified for an `AtomicSoftwareComponentType` and its `InternalBehavior` are visualized through the handles (line with filled circle) that are attached to the AUTOSAR entities they refer to. In the example, these relate to the `RunnableEntity`. □

¹This is the AUTOSAR-specific implementation of the demand stated by Klar et al. [48] that event classes must have a unique name by which they can be unambiguously identified.

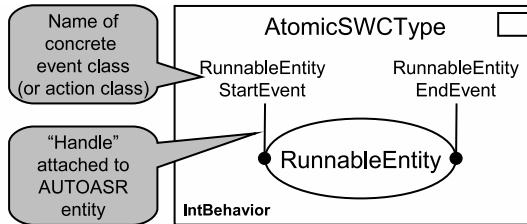


Figure 5.2: Example `AtomicSoftwareComponentType` with specification of `RunnableEntityEvents`

5.2.2 Runtime Environment API Actions

In this section, we formally define the concrete AUTOSAR-specific action classes which relate to the RTE API of an `AtomicSoftwareComponentType` and its `InternalBehavior`. Through this, all points of interaction of a `RunnableEntity` within an `AtomicSoftwareComponentType` with the RTE can be marked as interesting actions where the communication between the `RunnableEntity` and the RTE shall be observed. As a starting point, we define an abstract superclass `RTEAPIAction` which serves as basis for all concrete RTE API action classes.

Definition 13 *An `RTEAPIAction` is an action class where instances can be observed when a `RunnableEntity` makes an API call to an RTE function or macro.*

The `RTEAPIAction` is the abstract superclass for all concrete RTE API actions for Sender/Receiver communication, Interrunnable communication and Client/Server communication. `RTEAPIActions` are defined in the context of the `InternalBehavior` of an `AtomicSoftwareComponentType`, meaning that they can be only defined for an `AtomicSoftwareComponentType` with a concrete `InternalBehavior`². When a different `InternalBehavior` is defined for an `AtomicSoftwareComponentType`, the `RTEAPIActions` need to be redefined as well. The meta-model specification can be found in Appendix E.2.

As described in section 4.3, AUTOSAR defines three different communication patterns which each have further specializations. In the following, the definitions for the `RTEAPIActions` which are involved in Sender/Receiver communication, Client/Server communication and Interrunnable communication are introduced.

²This is a main difference to existing approaches towards a Timing Model for AUTOSAR (e.g., efforts by the AUTOSAR initiative or efforts of the TIMMO project). In existing approaches, events refer to data elements in the ports and thus only to the structural interface of a software component. This is, however, not sufficient for the generation of an instrumentation for simulation or monitoring.

RTE API Actions for Sender/Receiver Communication

For Sender/Receiver communication, we first define two abstract action classes that allow us to monitor the sending/writing and receiving/reading actions of data elements performed by a RunnableEntity. These are the SendDataAction and the ReceiveDataAction.

Definition 14

- The **SendDataAction** represents an RTE API action where instances can be observed when a RunnableEntity performs a **sending or writing action** on a DataElementPrototype.
- The **ReceiveDataAction** represents an RTE API action where instances can be observed when a RunnableEntity performs a **receiving or reading action** on a DataElementPrototype.

As outlined in section 4.3.6, there are two concrete specializations of Sender/Receiver communication: implicit and explicit Sender/Receiver communication. This requires also two specializations each of the abstract RTEAPIActions defined above. Table 5.1 provides an overview on the concrete RTEAPIActions for Sender/Receiver communication.

Concrete RTEAPIAction	Semantics	Referenced AUTOSAR entity
ReceiveDataImplicitAction	Implicit read action	DataReadAccess
SendDataImplicitAction	Implicit write action	DataWriteAccess
ReceiveDataExplicitAction	Explicit read or receive action	DataReceivePoint
SendDataExplicitAction	Explicit write or send action	DataSendPoint

Table 5.1: Concrete RTEAPIActions for Sender/Receivr communication

Note that each concrete RTEAPIAction for Sender/Receiver communication has an explicit reference to an AUTOSAR element which describes how the concrete data access of a RunnableEntity to a DataElementPrototype is specified. Through this, the location of the action class is unambiguously defined. As the RTEAPIActions are defined in the context of the InternalBehavior of an AtomicSoftwareComponentType, simple references (in contrast to instance references) are sufficient to unambiguously identify the referenced AUTOSAR entity. The meta-model specification for the RTEAPIActions of Sender/Receiver communication can be found in Appendix E.2.

Furthermore note that with respect to explicit Sender/Receiver communication, we do not distinguish between queued communication (i.e., event semantics) and unqueued communication (i.e., data semantics) in the concrete RTEAPIActions. This distinction is already made as part of the respective DataElementPrototype specification to which a RunnableEntity specifies its access. For queued communication, the length of the queue must be specified as part of the so-called PortAnnotation, more

precisely the SenderReceiverAnnotation. The information is thus already available in an AUTOSAR model.

Example: Figure 5.3 depicts two example AtomicSoftwareComponentTypes with RTEAPIActions modeled for implicit and explicit Sender/Receiver communication.

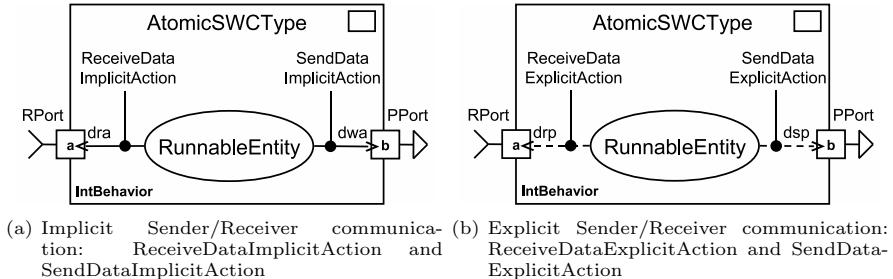


Figure 5.3: Example AtomicSoftwareComponentTypes with specification of RTEAPIActions for Sender/Receiver communication

In the example, the handles of the RTEAPIActions point to the port access specifications of the RunnableEntity, i.e. to a DataReadAccess (dra), DataWriteAccess (dwa), DataReceivePoint (drp) and DataSendPoint (dsp). \square

RTE API Actions for Interrunnable Communication

In order to model the AUTOSAR-specific action classes that can be monitored in Interrunnable communication, we define concrete RTEAPIActions for Interrunnable communication in analogy to the concrete RTEAPIActions for Sender/Receiver communication.

The definition of the RTEAPIActions for Interrunnable communication, however, requires a change of the original AUTOSAR meta-model specification. Figure 5.4 depicts the relevant excerpt from the original AUTOSAR meta-model.

With the original AUTOSAR concepts, it is not possible to define a reference to the reading/writing data access specification of a RunnableEntity to an InterRunnableVariable (“writtenVariable”, “readVariable”) as an InterRunnableVariable is not an Identifiable element in the AUTOSAR sense³. This, however, is required to adequately define the concrete AUTOSAR-specific action classes for InterRunnable communication and to integrate the latter with the AUTOSAR concepts.

Thus, for our purposes, the original AUTOSAR meta-model specification needs to be changed. We define two classes IrvDataWriteAccess and IrvDataReadAccess that

³I.e., in the original AUTOSAR meta-model, the class InterRunnableVariable does not inherit from the superclass Identifiable and thus does not have the attribute “shortname” by which it could be identified.

5 Specification of Timing Requirements based on Formal Path Specifications

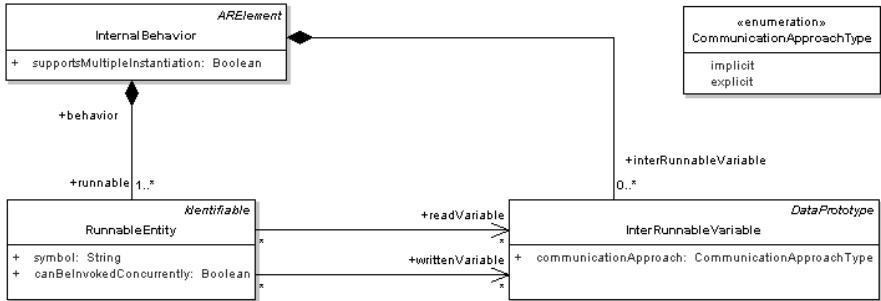


Figure 5.4: Specification of InterRunnableVariable and concepts for specifying reading/writing data access of a RunnableEntity to it (excerpt from AUTOSAR meta-model [6])

both have a reference to an InterRunnableVariable. Figure 5.5 depicts the changed meta-model excerpt. The grayish classes are newly introduced and denote the data access specifications which can now be referenced by concrete RTEAPIActions of Interrunnable communication. Furthermore, an inheritance relation to the AUTOSAR superclass Identifiable has been added.

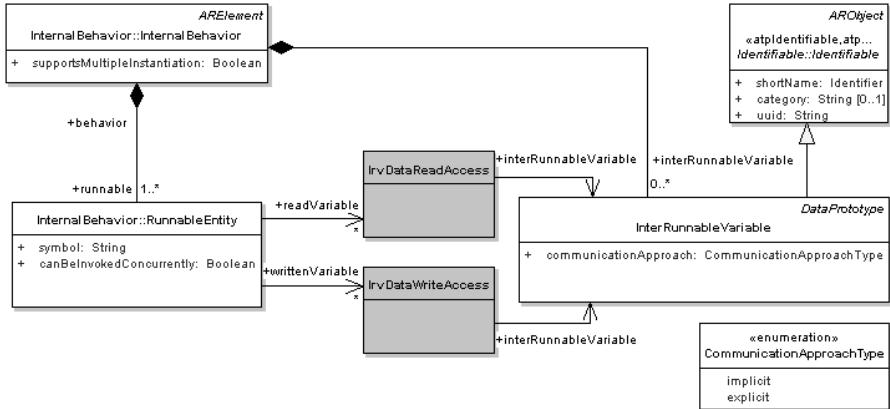


Figure 5.5: Specification of InterRunnableVariable and changed concepts for specifying reading/writing access of a RunnableEntity to it

We then define two abstract RTEAPIActions that allow the monitoring of the writing and reading actions of Interrunnable communication.

Definition 15

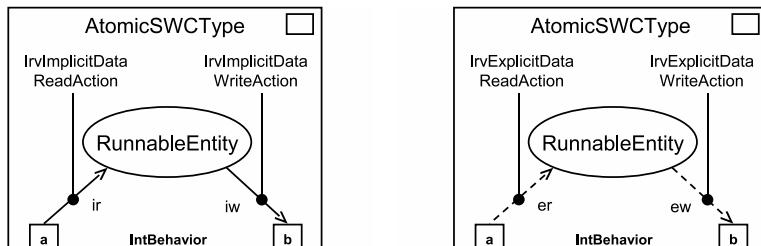
- The **IrvDataReadAction** represents an RTE API action where instances can be observed when a RunnableEntity performs a **reading action** on an **InterRunnableVariable**.
- The **IrvDataWriteAction** represents an RTE API action where instances can be observed when a RunnableEntity performs a **writing action** on an **InterRunnableVariable**.

As outlined in section 4.3.6, there are two specializations of Interrunnable communication: implicit and explicit Interrunnable communication. Thus, there are also two specializations required for each of the abstract RTE API actions defined above. Table 5.2 provides an overview of the concrete RTEAPIActions for Interrunnable communication.

Concrete RTEAPIAction	Semantics	Referenced AUTO-SAR entity
IrvImplicitDataWriteAction	Implicit read action	IrvDataReadAccess
IrvImplicitDataReadAction	Implicit write action	IrvDataWriteAccess
IrvExplicitDataWriteAction	Explicit read action	IrvDataReadAccess
IrvExplicitDataReadAction	Explicit write action	IrvDataWriteAccess

Table 5.2: Concrete RTEAPIActions for Interrunnable communication

Example: Figure 5.6 depicts two example AtomicSoftwareComponentTypes with RTEAPIAction specifications for implicit and explicit Interrunnable communication.



(a) Implicit Interrunnable communication: IrvImplicitDataReadAction and IrvImplicitDataWriteAction
(b) Explicit Interrammable communication: IrvExplicitDataReadAction and IrvExplicitDataWriteAction

Figure 5.6: Example AtomicSoftwareComponentTypes with specification of RTEAPIActions for Interrunnable communication

The handles of the RTEAPIActions point to the AUTOSAR entities IrvDataReadAccess (ir, er) and IrvDataWriteAccess (iw, ew) that we introduced as part of the necessary AUTOSAR meta-model modification. □

The meta-model specification for the concrete RTEAPIActions for Interrunnable communication can be found in Appendix E.2.

RTE API Actions for Client/Server Communication

Client/Server communication in general involves two threads of execution: a client side which requests or calls a service, and a server side which provides this service by implementing an operation. Between the client and the server, data is exchanged in the form of in and out arguments as specified by the signature of the operation. To monitor how Client/Server communication is established and consequently how the values of the output arguments are produced based on the values of the input arguments, the involved actions and events on both the client and server side can be monitored. For this purpose, we define two abstract superclasses, each for one side: the ServerCallAction and the OperationEvent.

Definition 16

- The *ServerCallAction* represents an RTE API action where instances can be observed on the *client side* of a Client/Server communication when a RunnableEntity acting as a client performs a server call action.
- The *OperationEvent* represents an RTE API event where instances can be observed on the *server side* of a Client/Server communication when a RunnableEntity implementing a service starts or terminates.

In Client/Server communication, the two cases of synchronous and asynchronous Client/Server communication need to be further distinguished. In synchronous Client/Server communication, the thread of control is handed over to the server after the client call, and returned to the client after the server has terminated. During the execution of the server, the client is blocked. Due to this, there is in principle only one relevant RTEAPIAction involved in synchronous Client/Server communication, i.e. the synchronous server call action. In asynchronous Client/Server communication, the thread of control of the RunnableEntity calling the service continues after the call has been made. Upon notification by the RTE (AsynchronousServerCall>ReturnsEvent), the notified RunnableEntity fetches and further processes the result provided by the server. In this case, there are two RTE API actions, one for the asynchronous call of the operation by the client RunnableEntity, and one for the collection of the result by the RunnableEntity that has been marked as responsible for this (activated through an AsynchronousServerCall>ReturnsEvent by the RTE).

In the following, we provide definitions for these concrete RTEAPIActions where instances can be observed on the client side.

Client Side

As outlined in section 4.3.6, there are two concrete specializations of Client/Server communication: synchronous and asynchronous Client/Server communication. In

order to account for their differences in simulation- and monitoring-based approaches for the determination of timing properties, we define two specializations of the abstract RTE API actions defined above.

Definition 17

- The *SynchronousServerCallAction* represents the RTE API action where instances can be observed when a client RunnableEntity performs a synchronous server call action on an OperationPrototype.
- The *AsynchronousServerCallAction* represents the RTE API action where instances can be observed when a client RunnableEntity performs an asynchronous server call action on an OperationPrototype.

Note that the SynchronousServerCallAction is already a concrete RTEAPIAction while the AsynchronousServerCallAction is an abstract superclass for two further specializations, the AsynchronousServerCallStartAction and the AsynchronousServerCallReturnAction. The latter need to be distinguished due to the fact that in asynchronous Client/Server communication, the thread of control continues after the server call (non-blocking) whereas in synchronous Client/Server communication, the client is blocked.

Table 5.3 provides an overview of the concrete RTEAPIActions for Client/Server communication for the client side. The meta-model specifications can be found in Appendix E.2.

Concrete RTEAPI-Action	Semantics	Referenced AUTO-SAR entity
SynchronousServerCall-Action	Start and return of a synchronous server call	SynchronousServer-CallPoint
AsynchronousServerCall-StartAction	Start of an asynchronous server call	AsynchronousServer-CallPoint
AsynchronousServerCall-ReturnAction ⁴	Return of an synchronous server call	AsynchronousServer-CallPoint

Table 5.3: Concrete RTEAPIActions for Client/Server communication (client side)

Example: Figure 5.7(a) depicts an example AtomicSoftwareComponentType with a client RunnableEntity and the specification of a SynchronousServerCallAction. A typical process of a synchronous Client/Server communication is shown in figure 5.7(b).

After the server is called the RunnableEntity ClientRE is blocked until the operation is terminated and the result is returned. Note that the implicit order between the start event and the end event of the SynchronousServerCallAction complies with the causal order between input and output arguments of the operation. This is important for the description of signal paths that include Client/Server communication. □

5 Specification of Timing Requirements based on Formal Path Specifications

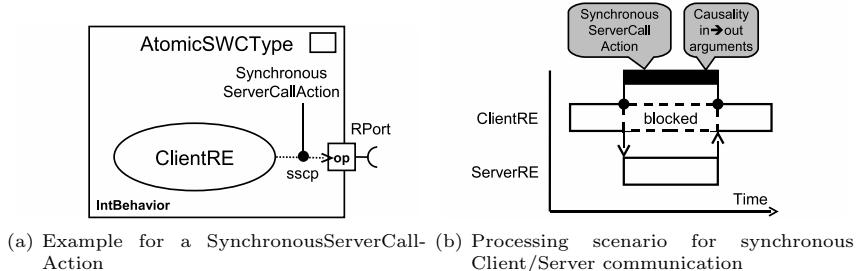


Figure 5.7: Example AtomicSoftwareComponentType with a SynchronousServerCallAction

Example: Figure 5.8(a) depicts an example AtomicSoftwareComponentType with a client RunnableEntity and the specification of an AsynchronousServerCallStart- and ReturnAction. Figure 5.8(b) shows the process of an asynchronous Client/Server communication where the result is processed by the same RunnableEntity that invoked the service.

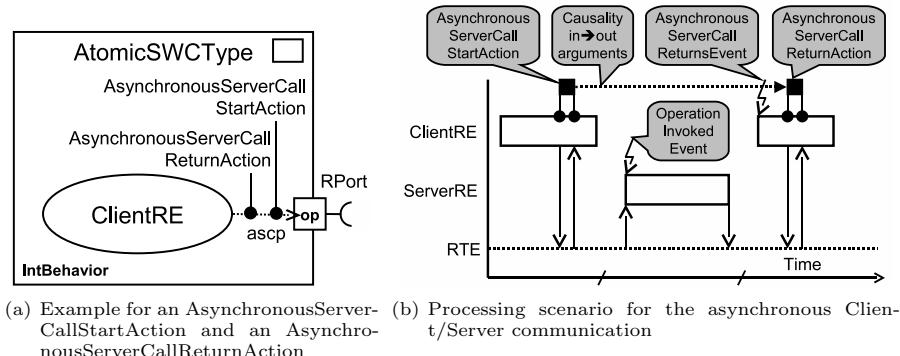


Figure 5.8: Example AtomicSoftwareComponentType with an AsynchronousServerCallStartAction and an AsynchronousServerCallReturnAction

Note that in figure 5.8(a), the handles of the RTEAPIActions on the client side both refer to the same AsynchronousServerCallPoint. An AsynchronousServerCallPoint is used to denote both the start point and the end point of a service invocation.

After the server is called by the RunnableEntity ClientRE, its thread of control continues until it terminates. The RTE activates the ServerRE (by means of an Operation-InvokedEvent) which implements the operation. The ServerRE is provided with the input arguments from the server call and returns its results to the RTE. After the

ServerRE has terminated, the ClientRE is invoked by an AsynchronousClientServerCallReturnsEvent by the RTE. It can then fetch the result which can be observed by the AsynchronousServerCallReturnAction. The order of the two action classes, however, must be described explicitly in a signal path description. \square

Example: An aspect that further distinguishes asynchronous Client/Server communication from synchronous Client/Server communication is the fact that in asynchronous Client/Server communication, the RunnableEntity collecting the result from a service invocation does not need to be the same RunnableEntity that also called the service. However, it must belong to the InternalBehavior of the same AtomicSoftwareComponentType. Figure 5.9(a) depicts an example AtomicSoftwareComponentType with such a scenario. Figure 5.9(b) shows the process of an asynchronous Client/Server communication where the result is processed by a different RunnableEntity than where the service was invoked.

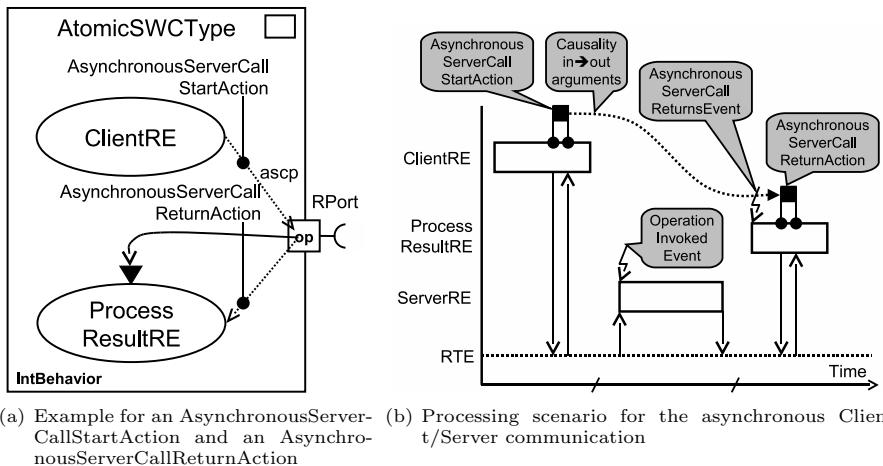


Figure 5.9: Example AtomicSoftwareComponentType with an AsynchronousServerCallStartAction and an AsynchronousServerCallReturnAction belonging to different RunnableEntities

The RunnableEntity ClientRE specifies an AsynchronousServerCallPoint that references the OperationPrototype **op** in the RPortPrototype. The server call can be observed by modeling an AsynchronousServerCallStartAction. When the server operation has finished, the RTE activates the RunnableEntity ProcessResultRE by means of the AsynchronousServerCallReturnsEvent. The RunnableEntity ProcessResultRE can then explicitly collect the result. This is observed by the AsynchronousServerCallReturnAction. Again, the causality between input and output arguments complies with the order of the two RTEAPIActions. This must be captured explicitly by a signal path description. \square

Server Side

A server RunnableEntity can in general only implement a single operation. This means that the server RunnableEntity *is* the operation. The operation starts when the RunnableEntity starts and finishes when the RunnableEntity terminates. These start and termination events of a RunnableEntity can be monitored by means of RunnableEntityEvents.

Definition 18

- The ***OperationStartEvent*** represents an event class that marks the *start* of a server RunnableEntity.
- The ***OperationEndEvent*** represents an event class that marks the *end* of a server RunnableEntity.

Both OperationEvents have a reference to a server RunnableEntity that implements the operation. This is provided by inheriting these two events from the abstract superclass RunnableEntityEvent. Note that these two events are thus no RTEAPIActions and are explained here only for the sake of completeness. For the description of signal paths that includes Client/Server communication, they are also not important because from the client perspective, the execution of the operation is handled transparently by the RTE of an AUTOSAR ECU system. The meta-model specification for the concrete OperationEvents can be found in Appendix E.2.

Example: Figure 5.10 depicts an example AtomicSoftwareComponentType with a server RunnableEntity and an OperationStartEvent and an OperationEndEvent.

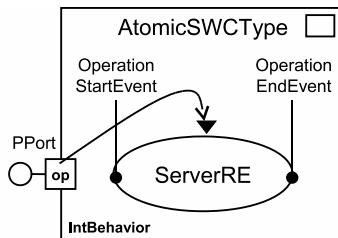


Figure 5.10: Example AtomicSoftwareComponentType with OperationEvents

□

5.2.3 Communication Events

In AUTOSAR, inter-ECU communication is transparently performed via the RTE of the involved ECUs by using the services provided by the COM modules. The COM module of an AUTOSAR ECU provides communication services in terms of

an API to the RTE such that the RTE itself is independent of any concrete hardware and network technology.

The Virtual Functional Bus Tracing Mechanism provides several hook functions which are invoked when a signal or signal group is transmitted or is received via the COM module. These have been described in section 4.3.7. In order to specify that instances of these events shall be monitored in an AUTOSAR system model, we define COMSignalEvents for system signals and COMSignalGroupEvents for system signal groups to mark the relevant places by means of a concrete event class. In the following, these event classes are introduced into the AUTOSAR meta-model.

COM Events for Transmission and Reception of System Signals

To monitor the transmission or reception of system signals, COMSignalEvents are introduced.

Definition 19 A *COMSignalEvent* represents an event class where instances can be observed when a system signal is transmitted to or received from the COM module by the RTE.

A COMSignalEvent has a reference to the respective system signal it refers to.

A system signal can be either transmitted or received. To distinguish the transmission from the reception, the following two specializations are introduced.

Definition 20

- A *COMSendSignalEvent* represents a COM signal event where instances can be observed when a **system signal is transmitted** to the COM module by the RTE.
- A *COMReceiveSignalEvent* represents a COM signal event where instances can be observed when a **system signal is received** from the COM module by the RTE.

Note that there is no graphical representation for these event classes as they are not intended to be used for specification of signal paths in an AUTOSAR system by a modeler. Rather, they are introduced in the course of the transition from the planning phase (Virtual Functional Bus view) to the realization phase (System view). The definition of COMEvents for signal groups is analogous and can be found in Appendix E.2.

5.2.4 Operating System Events

By means of the Virtual Functional Bus Tracing Mechanism, Trace Events can be monitored which are related to OS objects. However, Trace Events are defined only for a subset of OS-objects such as OS-tasks and OS-events. Events to monitor other OS-objects such as OS-resources are not defined. Also, no events are defined to monitor the preemption and restart of an OS-task by means of the Virtual Functional

Bus Tracing Mechanism. Preemptions are managed by the operating system and are transparent to OS-tasks being preempted. For our purposes, however, it is also not important to monitor preemptions as these are transparent and do not have an impact on the causality between events and actions that lie on a signal path.

Figure 5.11 depicts the state transition models for basic tasks and extended tasks as defined in OSEK-OS [66] and AUTOSAR OS [7].

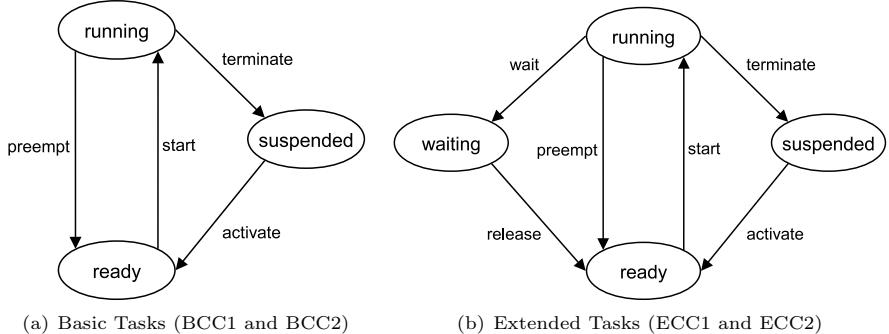


Figure 5.11: State transition models for basic and extended tasks in OSEK/VDX and AUTOSAR compliant real-time operating systems

For our Timing Model for AUTOSAR, we define the following OSEvents as AUTOSAR-specific event classes that allow the monitoring of the life-cycle of OS-tasks.

Definition 21

- A **TaskActivateEvent** represents an event class where instances can be observed when an OS-task is activated. This corresponds to the edge labeled **activate** in figure 5.11.
- A **TaskDispatchEvent** represents an event class where instances can be observed when an OS-task starts. This corresponds to the edge labeled **start** in figure 5.11.
- A **TaskTerminateEvent** represents an event class where instances can be observed when an OS-task terminates. This corresponds to the edge labeled **terminate** in figure 5.11.

As shown in Figure 5.11, compared to basic tasks, extended tasks do have an additional state **waiting** which allows the synchronization of OS-tasks by means of OS-events.

- A **TaskWaitEvent** represents an event class where instances can be observed when an OS-task waits for an OS-event. This corresponds to the edge labeled **wait** in figure 5.11.

- A *TaskReleaseEvent* represents an event class where instances can be observed when an OS-task is released by an OS-event. This corresponds to the edge labeled *release* in figure 5.11.

OSEvents are defined in the scope of an ECU instance with references to the OS-objects defined in the ECU configuration for the ECU instance. The meta-model specification can be found in Appendix E.2

Note that the TaskTerminateEvent cannot be monitored by the Virtual Functional Bus Tracing Mechanism as no hook function is defined. For our purposes of determining timing properties by means of simulation- and monitoring-based approaches, we extend the Virtual Functional Bus Tracing Mechanism and introduce such a hook function that is called just prior to the termination of an OS-task. For the determination of instances of signal paths in simulation- or monitoring-based approaches, only the TaskDispatchEvent and the TaskTerminateEvent are of interest, and this also only in the special cases of implicit Sender/Receiver or Interrunnable communication. This is described further in chapter 6.

5.2.5 Overview on Event and Action Classes

In this section, concrete AUTOSAR-specific event and action classes which can be identified and marked in an AUTOSAR system description and which can be monitored by means of the Virtual Functional Bus Tracing Mechanism have been introduced. The concrete AUTOSAR-specific event and action classes denote interesting places and actions where instances can be monitored with respect to

- the life cycle of RunnableEntities (RunnableEntityEvents)
- the interaction of RunnableEntities and the RTE (RTEAPIActions)
- the life cycle of OS objects (OSEvents)
- the transmission and reception of system signals and system signal groups in inter-ECU communication (COMSignalEvents and COMSignalGroupEvents).

The concrete AUTOSAR-specific event and action classes are the basis for precisely and unambiguously specifying signal paths in AUTOSAR systems.

Tables E.1 to E.4 in Appendix E provide an overview on the concrete AUTOSAR-specific event and action classes.

5.3 Clocks

With respect to time, instances of event and action classes that can be monitored in an AUTOSAR system occur at points in time (events) or between two points in time (actions). Time itself is measured by means of measurement instruments, i.e. clocks. Furthermore, for the specification of timing requirements based on events as well as for the determination of timing properties by means of event-based simulation

or monitoring in parallel and distributed computer systems, it is necessary that the temporal context in which event classes are defined and instances of these event classes occur is defined. This is required in order to unambiguously specify to which time base the time values of instances of event classes refer such that safe reasoning between events denoting causes and effects can be performed. This ultimately allows that timing requirements can be determined and compared with timing properties such that the degree of fulfillment of the former can be evaluated.

The AUTOSAR standard does not yet provide a formal notion of time. For this, it is necessary to characterize the real-time clocks that are present in an AUTOSAR system description and to bind all event classes to a clock such that the time values of their instances have a defined temporal context. To achieve this, we adapt the concepts of the formal clock model by Münzenberger et al. [56]. This also allows the specification of timing requirements in a defined temporal context and enables the determination of timing properties based on events monitored in parallel and distributed computer systems.

For the development of AUTOSAR-compliant electric/electronic systems, the AUTOSAR standard defines two phases: a planning phase and a realization phase. With respect to the AUTOSAR development methodology there are, however, specific challenges for the introduction of a formal notion of time based on clocks. The problem is that in the planning phase, no real-time clock exists in the logical software architecture to provide a temporal context to what event classes can be bound in the AUTOSAR system description or to which the time values of timing requirements could refer to. Real-time clocks only exist in the ECU hardware platforms which are part of the system topology on which the application software architecture is later being mapped. The problem then becomes how a binding of event classes to existing real-time clocks in the ECUs is established in the realization phase. In the following, a formal notion of time is introduced into the concepts of the AUTOSAR standard that also addresses these specific problems.

5.3.1 Planning Phase (Virtual Functional Bus View)

In order to give event and action classes that can be identified in a logical software architecture of an AUTOSAR system a definite temporal context, and in order to characterize the real-time clocks that are present in the electronic control units of the system topology of an AUTOSAR system, we introduce the notion of a global clock that can be accessed by all entities in the AUTOSAR system and which has the characteristics of an ideal clock (see foundations of timing models, section 4.2). For every AUTOSAR system, there exists exactly one globally visible ideal clock to which all other clocks such as the real-time clocks in the single ECUs of an AUTOSAR system can refer as reference clock. In the following, we describe how the ideal clock is used in the context of the logical software architecture and in the system topology description.

Logical Software Architecture: Binding of Event and Action Classes to the Globally Visible, Ideal Clock

In the planning phase, the event and action classes which can be identified in the logical software architecture of an AUTOSAR system are implicitly bound to the ideal clock. By implicit we mean that no explicit references by the event and action classes to the ideal clock of the AUTOSAR system are required.

Example: Figure 5.12 shows the logical software architecture of the example AUTOSAR system for the simple control application with the RTEAPIActions that have been identified and the globally visible ideal clock.

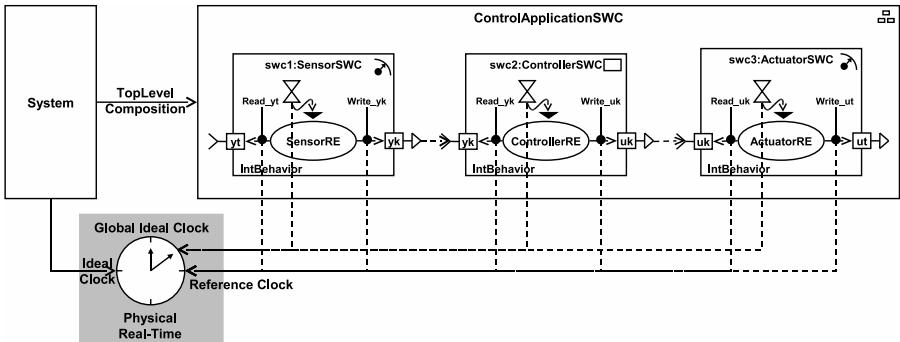


Figure 5.12: Planning Phase: Binding of event and action classes from the logical software architecture to the system-wide ideal clock

In the figure, the implicit references are indicated by the dashed lines. □

Note that the time values of the periods specified in the TimingEvents that determine the rates at which the referred RunnableEntities should be activated can also refer to the ideal clock. In figure 5.12, this is indicated by the dashed lines going from the TimingEvents to the global ideal clock of the AUTOSAR system. The ideal clock thus establishes a global time base based on which the time-triggered activation of RunnableEntities by means of TimingEvents can be also be planned.

System Topology Description: Characterization of Real-Time Clocks based on Globally Visible Ideal Clock

In AUTOSAR-compliant electric/electronic systems, each ECU has its own real-time clock that dictates the speed by which the code of the application software deployed onto the ECU is executed by the processing unit.

As the real-time clocks of different ECUs have in principal different physical characteristics, their readings can become out of sync such that clock drift effects can

5 Specification of Timing Requirements based on Formal Path Specifications

occur. This can have negative effects on the timing properties of a real-time application such that their timing requirements are violated. Furthermore, in monitoring-based approaches, time-stamps which are obtained as readings of the real-time clocks from different ECUs are in general not directly comparable as they refer to different time bases established by the different real-time clocks of the ECUs. To account for such problems during development, it is required that the properties of the real-time clocks are adequately specified.

The characterization of the real-time clocks of an AUTOSAR system can be performed through the adaptation of the formal clock model concepts from Münzenberger et al [69]. The characteristics of the real-time clock of an ECU can be specified through parameters such as the *drift*, *granularity* and *step width* of the real-time clock as described in the foundations of timing models in section 4.2. These parameters implicitly refer to the globally visible ideal clock of the AUTOSAR system.

Example: Figure 5.13 shows the system topology of the example AUTOSAR system for the simple control application.

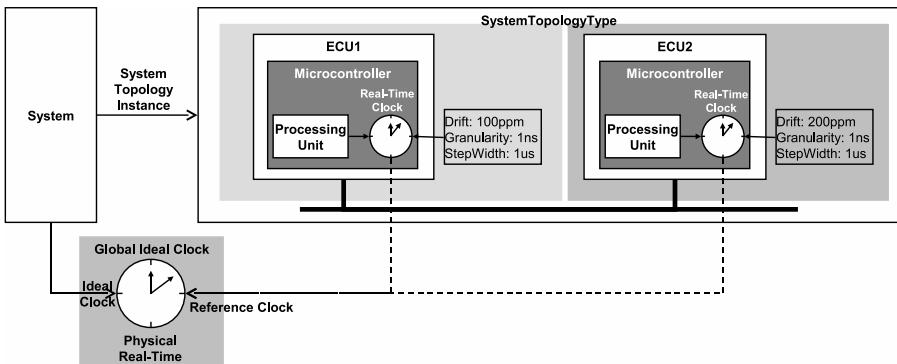


Figure 5.13: Planning Phase: Characterization of real-time clocks in the ECUs

Each ECU has a real-time clock which dictates the speed at which instructions are processed by the processing unit. The real-time clocks are formally characterized by the parameters drift, granularity and step width. While the granularity and the step width are the same, the drift parameters differ. This has influence on the timing properties of distributed real-time applications as the two time bases which are established by the two real-time clocks, and on which time-triggered activations are performed, can get out of sync. \square

The described parameters for the characterization of the behavior of the real-time clock of an ECU need to be integrated with the existing concepts for describing the hardware architecture of an AUTOSAR-compliant ECU and the concepts for describing their properties. The place where these parameters thus need to be added is the ECU Resource Template provided by AUTOSAR.

In simulation-based approaches, the concrete values of these parameters can be used to specify the behavior clocks which underly “virtual” ECUs. This allows an analysis of effects such as clock drift on the timing properties of the real-time applications being realized by the AUTOSAR system. In monitoring-based approaches, the parameters allow the determination of the required synchronization of the clocks such that safe statements can be made with respect to the temporal order of events.

5.3.2 Realization Phase (System View)

At the transition from the planning phase (VFB view) to the realization phase (System view) of an AUTOSAR system development, the logical software architecture is integrated with the system topology. Information on this mapping is aggregated in the System Mapping. The essential step which is of importance from a timing perspective is the mapping of instances of atomic software components on the leaf level of the logical software architecture to the distinct ECU instances of the system topology.

In line with the System Mapping, the event and action classes which mark the relevant places and actions within the atomic software components of the logical software architecture are bound to a definite, concrete temporal context in which instances of the event and action classes occur at runtime. Through the mapping of an atomic software component instance to an ECU, all event and action classes owned by the InternalBehavior of that atomic software component are automatically bound to the real-time clock of the ECU.

Through the binding of event and action classes to real-time clocks of ECUs, the time values of instances of these event classes and action classes become readings of the respective real-time clocks.

Example: Figure 5.14 shows the example AUTOSAR system of the simple control application after the system mapping, the role of the clocks (globally visible ideal clock and real-time clocks of the ECUs) as well as the resolved binding of event and action classes to the real-time clocks.

Through the system mapping decisions, the time domains of the real-time clocks in the ECUs spreads into the logical software architecture □

5.4 Event Chains

In order to reason about causes and effects in systems in which events and actions can be observed, the causal relation between these events and actions must be described. For this purpose, we introduce a concept to relate the event and action classes that mark the places and actions in an AUTOSAR system, or more generally speaking the atomic observable entity classes, to each other to so-called called *event chains*.

An event chain describes the causal order in which instances of event and action classes must appear such that it can be safely assumed that an instance of an event

5 Specification of Timing Requirements based on Formal Path Specifications

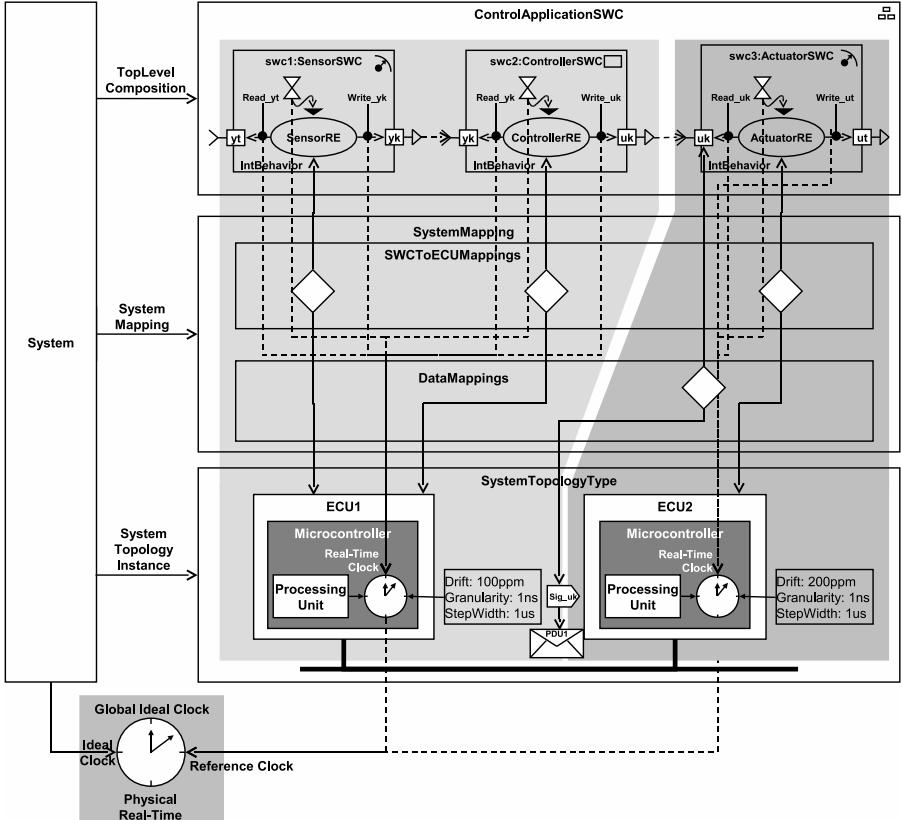


Figure 5.14: Realization Phase: Binding of event and action classes from instances of atomic software components to real-time clocks in ECU instances

or action class that denotes an effect is really affected by another instance of an event class or action class that denotes its cause.

A simple and straight-forward solution would be to describe such an event chain as flat ordered set where the order of the contained event and action classes corresponds to the causal order between these. However, the concepts of the software component technology in AUTOSAR impose specific challenges which hinder the application of such simple solutions⁵. The problem is that the description of the system is not flat but hierarchical. A system is rather composed of independent components which are aggregated in multiple levels of hierarchy. The events that can be observed are

⁵We think that this could also be a main reason why a feasible solution has not yet been found for describing event chains in AUTOSAR.

on the leaf-level of the component hierarchy and are not directly accessible from higher levels of the component hierarchy. This also means that chains of cause-and-effect and consequently signal paths are segmented which makes their description not trivial.

The AUTOSAR software component technology employs the so-called type/prototype concept to establish component hierarchies: AtomicSoftwareComponentTypes are instantiated in the context of a CompositionType to ComponentPrototypes. In order to define event chains which can seamlessly be integrated with the concepts for such component or function hierarchies, the type/prototype concept must also be applied.

Based on the concrete AUTOSAR-specific event and action classes we introduced into AUTOSAR, we introduce a concept to define hierarchical event chains for AUTOSAR systems. This then enables us to precisely and unambiguously describe signal paths in AUTOSAR systems.

5.4.1 Basic Concepts

The AUTOSAR software component technology employs the type/prototype concept to establish component hierarchies: AtomicSoftwareComponentTypes are instantiated in the context of a CompositionType to ComponentPrototypes. CompositionTypes themselves can be instantiated to ComponentPrototypes as well such that the nesting of components becomes hierarchical. In order to define event chains which seamlessly integrate with the component hierarchy of the logical software architecture of an AUTOSAR system, the type/prototype concept must also be applied to the concepts for describing event chains. This allows the definition of event chains which directly complement the specification of component types and their instances on all levels of a component hierarchy.

As a starting point for our hierarchical event chain concept, we define an abstract superclass *EventChainType* from which two specializations are derived: the *AtomicEventChainType* as the base case, and the *CompositeEventChainType* as recursive or hierarchical case. An *AtomicEventChainType* defines a total order on a set of event and action classes. In order to establish hierarchical event chains from previously defined event chains, the type/prototype concept is applied. This leads to the specification of the class *EventChainPrototype*. An *EventChainPrototype* is an instance of a previously defined *EventChainType* and is defined in the context of a *CompositeEventChainType*. To define an order between the *EventChainPrototypes* that are instantiated in the context of a *CompositeEventChainType*, *EventChainConnectorPrototypes* are introduced.

Definition 22

1. An *EventChainType* specifies a total order on a set of event and action classes. An *EventChainType* can be either an *AtomicEventChainType* or a *CompositeEventChainType*.

2. An **AtomicEventChainType** specifies a total order on a set of event and action classes which are directly referred by the **AtomicEventChainType**.
3. A **CompositeEventChainType** specifies a total order on a set of event and action classes which are indirectly referred by the instances of specified **EventChainTypes**. For this, previously defined **EventChainTypes** are instantiated in the context of the **CompositeEventChainType**. The instances are referred to as **EventChainPrototypes**. An **EventChainPrototype** has a reference to a **ComponentPrototype** which also belongs to the **CompositionType**. The order between **EventChainPrototypes**, and thus on the AUTOSAR-specific event and action classes on their leaf level, is established via **EventChainConnectorPrototypes**.

Note that, in contrast to the concrete AUTOSAR specific event and action classes which are defined as Identifiable objects and which can only “live” within the context of the InternalBehavior of an **AtomicSoftwareComponentType**, **EventChainTypes** are defined as AUTOSAR objects and thus can “live” and be defined stand-alone.

The **AtomicEventChainType** has a reference to the **InternalBehavior** of the **AtomicSoftwareComponentType** it belongs to. Thus, an **AtomicEventChainType** is described in the context of an **AtomicSoftwareComponentType** with a concrete **InternalBehavior**.

The **CompositeEventChainType** has a reference to a **CompositionType** it belongs to. This integrates **CompositeEventChainTypes** with **CompositionTypes**. Through the reference of an **EventChainPrototype** to a **ComponentPrototype** it is unambiguously specified which event and action classes are referred to on the leaf level. This is important in the case of multiple instantiation of a **ComponentType** within the same **CompositionType**. In this case, it is not sufficient to identify only the **AtomicSoftwareComponentType** in which the event or action class is defined, but to identify the concrete instance of the **AtomicSoftwareComponentType** as this is the context where the event or action class is also instantiated in.

The meta-model specification of the concepts for hierarchical event chains and their integration with the existing AUTOSAR concepts can be found in Appendix E.2.

Hierarchical event chains are constructed in a two stage process: first, **AtomicEventChainTypes** are defined which directly refer to concrete event and action classes, and which define an order on them. Then, the **AtomicEventChainTypes** are instantiated to **EventChainPrototypes** in the context of a **CompositeEventChainType** and connected via **EventChainConnectorPrototypes**. This establishes a total order on the set of event and action classes on the leaf-level.

In the following, several examples are provided for how the concepts are practically applied and how atomic and composite event chains are constructed.

First, several examples are provided for how an order on RTEAPIActions within **AtomicSoftwareComponentTypes** can be specified whereby the three different communication patterns defined by AUTOSAR are employed. Then, it is shown how hierarchical event chains can be specified within **CompositionTypes**. The concept

of hierarchical event chains supports multiple and also mixed component hierarchy levels for which examples are provided. It is also shown that the special case of multiple instantiation of an AtomicSoftwareComponentType is supported through the hierarchical event chain concepts. Consequently, this means that signal paths can precisely and unambiguously be described by means of hierarchical event chains.

5.4.2 Atomic Event Chains

The following examples describe AtomicEventChainTypes which are constructed from RTEAPIActions for Sender/Receiver communication, Interrunnable communication and Client/Server communication. An AtomicEventChainType defines an order on a set of RTEAPIActions that can be identified within the context of the InternalBehavior of an AtomicSoftwareComponentType. Note that in the examples, only a single communication pattern is considered at a time. Communication patterns can, however, also be employed in a mixed form within a RunnableEntity, depending on the RTE API function calls being made within the implementation of the respective RunnableEntities. For example, a RunnableEntity can perform a synchronous server call where data is first fetched, perform an operation on the data and then send it via an explicit write action to one or more other components. The communication patterns between software components, i.e. according to which the communication is established by the RunnableEntities performing the respective RTE API actions, must be consistent as described in section 4.3.

Example: Figures 5.15(a) and 5.15(b) show how order relations can be described based on the RTEAPIActions involved in Sender/Receiver communication.

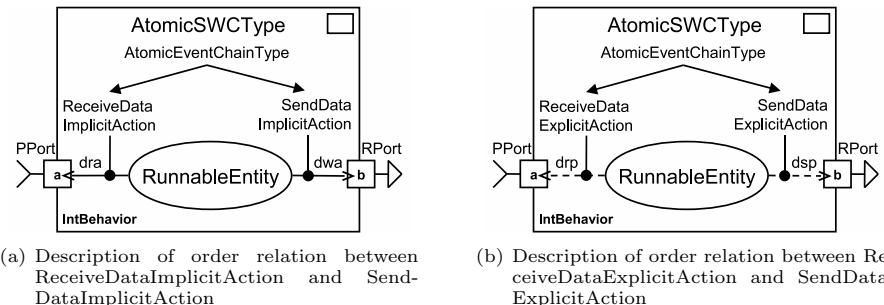


Figure 5.15: Description of order relations by means of AtomicEventChainTypes for Sender/Receiver communication

In both examples, the AtomicEventChainType specifies that the ReceiveDataAction comes before the SendDataAction. Note that in principle, implicit and explicit read and write actions can also be mixed. \square

Example: The definition of an order relation based on the RTEAPIActions involved in Interrunnable communication is shown in figure 5.16.

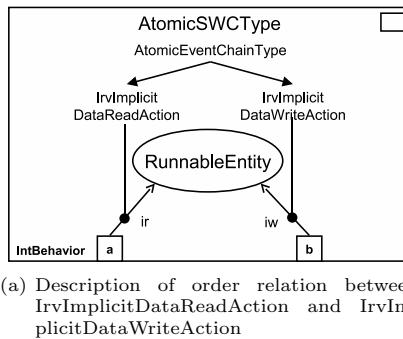


Figure 5.16: Description of order relations by means of an AtomicEventChainTypes for Interrunnable communication

In the example, the AtomicEventChainType specifies that the IrVImplicitDataReadAction on InterrunnableVariable a comes before the IrVImplicitDataWriteAction on InterrunnableVariable b. Order relations on RTEAPIActions for explicit Interrunnable communication can be described analogously. \square

In Client/Server communication, the thread of control goes from the client to the server and back. As the important thread of control is on the client side where input arguments are provided and the result in the form of output arguments are expected, it is sufficient to only describe the order between the RTEAPIActions on the client side. The event classes on the server side are transparent for the client.

Example: An example AtomicSoftwareComponentType for the client side of a synchronous Client/Server communication is shown in figure 5.17.

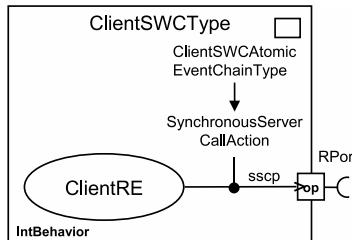


Figure 5.17: Example AtomicEventChainType for a synchronous Client/Server communication

The client RunnableEntity invokes the service by calling the operation `op`. As synchronous Client/Server communication is blocking, the thread of control continues on the server side. When the server has finished, the result is returned to the client and the thread of control continues there. \square

Example: Figure 5.18 depicts two example AtomicSoftwareComponentTypes for the client side of an asynchronous Client/Server communication. The client RunnableEntity invokes the service by calling the operation `op`. In the case of blocking asynchronous Client/Server communication, the client RunnableEntity waits in a WaitPoint. It is reactivated by an AsynchronousServerCallReturnsEvent when the server has processed the request and returned the result. In the case of non-blocking asynchronous Client/Server communication, the client RunnableEntity continues and subsequently terminates after invoking the operation. When the server has processed the request and returned the result, the RTE activates the RunnableEntity that subscribed for processing the results through an AsynchronousServerCallReturnsEvent. In both cases, the result is collected through an explicit RTE API call by the ClientRE which is resembled by the AsynchronousServerCallReturnAction.

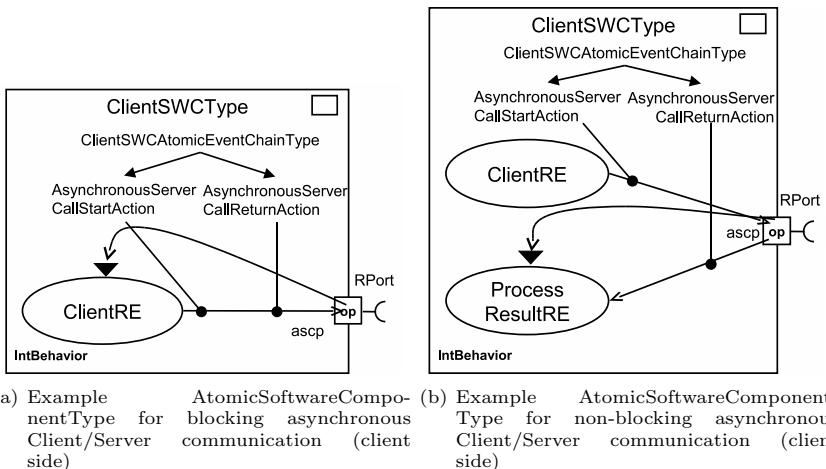


Figure 5.18: Two example AtomicEventChainTypes for asynchronous Client/Server communication (client side)

5.4.3 Composite Event Chains

The following examples show how different CompositeEventChainTypes are constructed. At first, we consider how composite event chains for one, two and mixed

hierarchy levels can be constructed for Sender/Receiver communication. Composite event chains for Interrunnable communication are in principle in the same way. Then, we show how composite event chains for Client/Server communication are constructed. Here, the two cases of synchronous and asynchronous Client/Server communication are distinguished.

Example: An example CompositeEventChainType for a CompositionType with one hierarchy level is shown in figure 5.19.

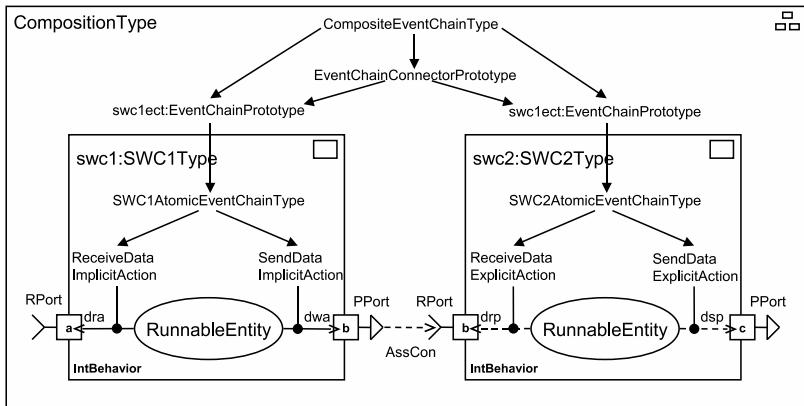
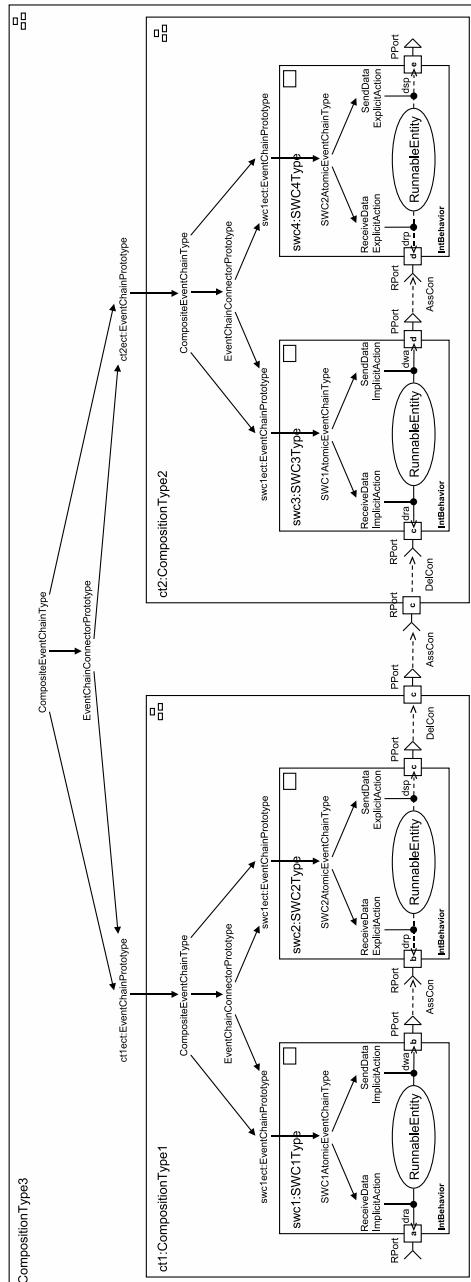


Figure 5.19: CompositeEventChainType for a CompositionType (with one hierarchy level)

The CompositeEventChainType is constructed by instantiating the previously defined AtomicEventChainTypes to EventChainPrototypes. The order of event and action classes of the EventChainPrototypes is then specified by the EventChainConnectorPrototype. □

Example: Figure 5.20 depicts an example CompositeEventChainType with two hierarchy levels. The example shows that the instantiation of EventChainTypes to EventChainPrototypes allows the establishment of event chain hierarchies which are directly in line with the component hierarchy of an AUTOSAR software architecture. □

Example: The example depicted in figure 5.21 shows that the hierarchy levels of event chains can effectively resemble the potentially mixed hierarchy levels of an AUTOSAR software architecture. □

Figure 5.20: CompositeEventChainType for a `CompositionType` (with two hierarchy levels)

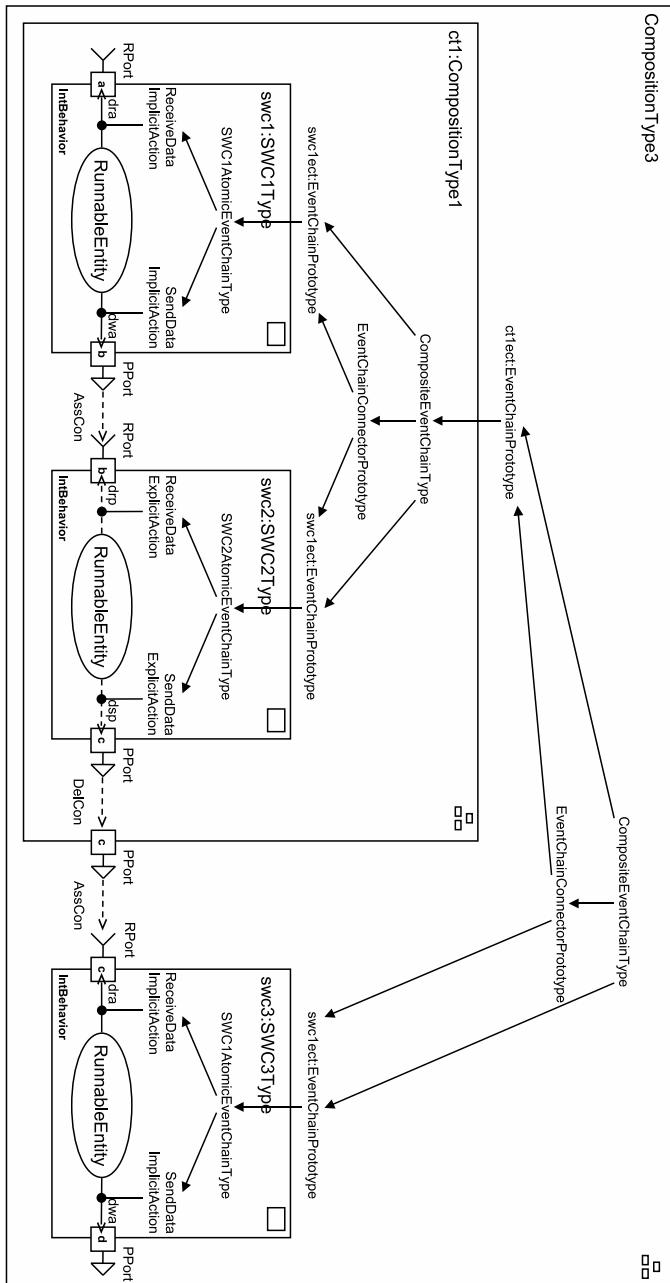


Figure 5.21: CompositeEventChainType for a CompositionType (with mixed hierarchy levels)

Example: An example CompositeEventChainType for a synchronous Client/Server communication is shown in figure 5.22. □

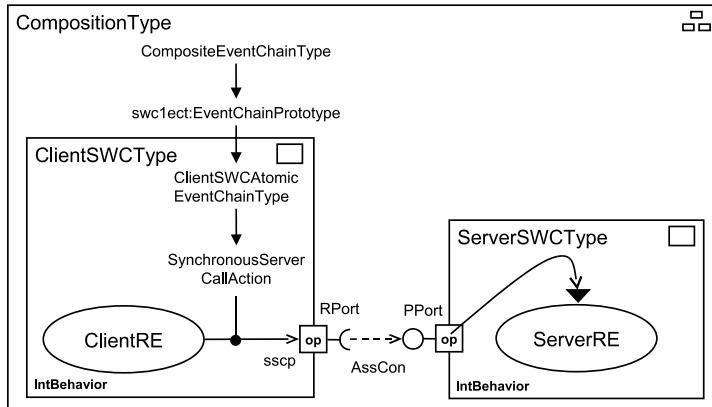


Figure 5.22: CompositeEventChainType for a CompositionType with synchronous Client/Server communication (one hierarchy level)

Example: CompositeEventChainTypes for asynchronous Client/Server communication are constructed in a similar way. Here, however, an AtomicEventChainType that refers to the AsynchronousServerCallStartAction and AsynchronousServerCallReturnAction must be constructed first which is then instantiated in the next hierarchy level. Figure 5.23 depicts an example. □

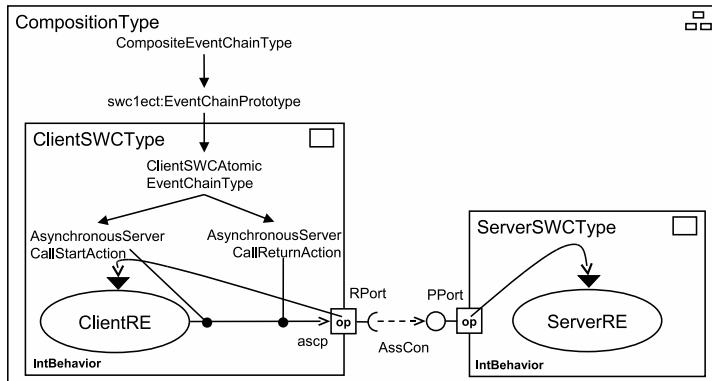


Figure 5.23: CompositeEventChainType for a CompositionType with asynchronous Client/Server communication (one hierarchy level)

5.4.4 Multiple Instantiation of Components and Event Chains

As described in section 4.3.5, in the InternalBehavior of an AtomicSoftwareComponentType it can be specified that the AtomicSoftwareComponentType can be instantiated multiple times in the software architecture of an AUTOSAR system. Multiple instantiation induces additional complexity as the RTEAPIEvents of the instances of AtomicSoftwareComponentTypes need to be unambiguously identified in a hierarchical event chain. This is achieved by instantiating the same ComponentType to distinct ComponentPrototypes. For event chains, the difficulty is then to refer to the event and action classes belonging to a specific ComponentPrototype on the leaf level of an event chain hierarchy. The following example demonstrates how this is handled through our hierarchical event chains concept. The reference from an EventChainPrototype to a ComponentPrototype allows us to unambiguously identify the correct event and action classes on the leaf level (see meta-model specification in figure E.13).

Example: Figure 5.24 depicts a SensorActuatorSoftwareComponentType which contains a single RunnableEntity, **SensorRunnable**, within its InternalBehavior.

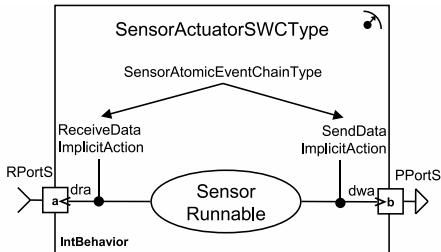


Figure 5.24: Specification of an AtomicEventChainType for multiply instantiable SensorActuatorSoftwareComponentType

The RunnableEntity transforms a value of DataElementPrototype **a** in the RPortPrototype **RPortS** to a value of DataElementPrototype **b** in PPortPrototype **PPortS**. In the InternalBehavior it is specified that the SensorActuatorSoftwareComponentType can be instantiated multiple times (not directly shown in the figure). The RTEAPIActions specify the interesting action classes, i.e., they refer to the implicit read and write actions. An AtomicEventChainType is specified that describes the order of the RTEAPIActions complying to the causal order of a data transformation from **a** to **b** can be observed.

In figures 5.25 and 5.26, the SensorActuatorSoftwareComponentType is instantiated twice to the ComponentPrototypes **Sensor1** and **Sensor2**.

The PPortPrototypes of both ComponentPrototypes are connected to respective RPortPrototypes of an instantiated AtomicSoftwareComponentType **Voter**. The latter contains a single RunnableEntity in its InternalBehavior which has defined access

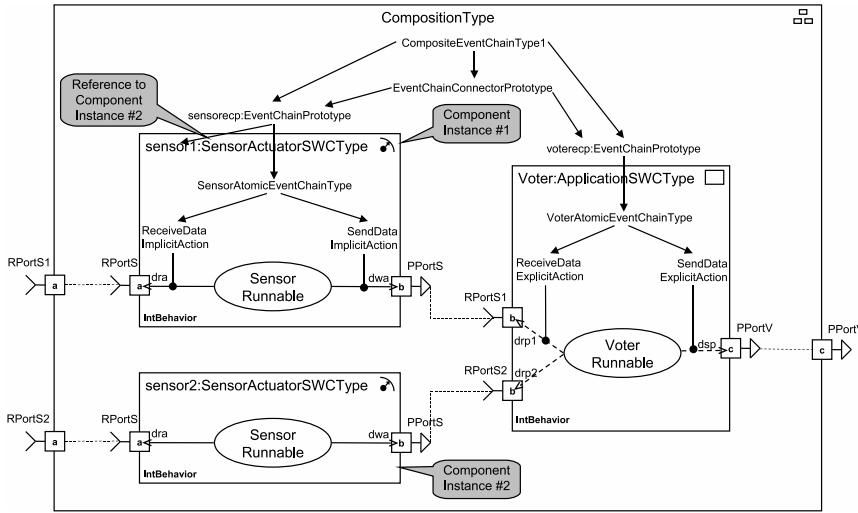


Figure 5.25: CompositeEventChainType specifying the order of RTEAPIActions for ComponentPrototypes Sensor1 and Voter

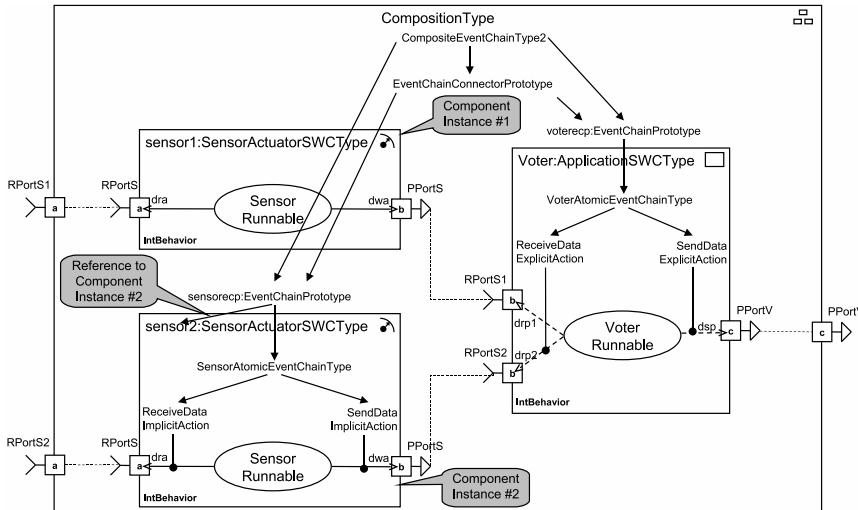


Figure 5.26: CompositeEventChainType specifying the order of RTEAPIActions for ComponentPrototypes Sensor2 and Voter

to the DataElementPrototypes in both RPortPrototypes **RPortS1** and **RPortS2** according to the explicit Sender/Receiver communication pattern. The purpose of the RunnableEntity **VoterRunnable** is to read both values of the DataElementPrototypes in **RPortS1** and **RPortS2**, to compute a voted value based on both read values, and to write the result to DataElementPrototype **c** in PPortPrototype **PPortV**.

In order describe the order of RTEAPIActions which allows the monitoring of a data transformation of a DataElementPrototype **a** on the RPortPrototypes of one of the instantiated ComponentPrototypes **Sensor1** or **Sensor2** to a resulting value of DataElementPrototype in PPortPrototype **c** of the ComponentPrototype **Voter**, two distinct CompositeEventChainTypes are required that reflect the causal order of the involved RTEAPIActions. These CompositeEventChainTypes are constructed by instantiating the AtomicEventChainType which has been defined for the SensorActuatorSoftwareComponentType to distinct EventChainPrototypes **sensorecp**. These are used together with an instantiated AtomicEventChainType of VoterAtomicEventChainType in two different CompositeEventChainTypes, **CompositeEventChainType1** and **CompositeEventChainType2**. Each of the two CompositeEventChainTypes defines the order of the RTEAPIActions on its leaf-level that must be observed such that the data transformations can be monitored.

□

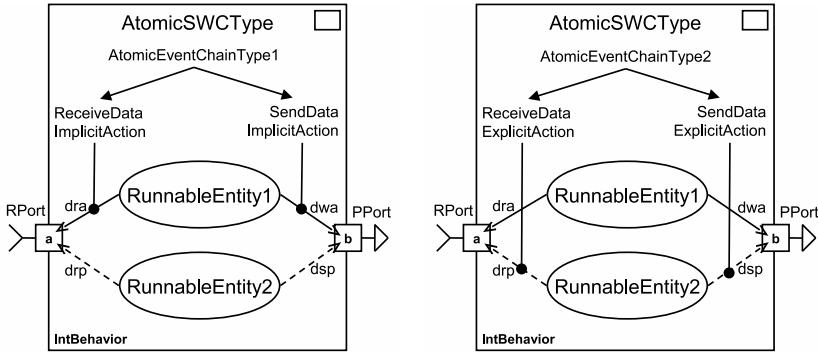
The example shows that through the application of the type/prototype concept to hierarchical event chains and through the reference from an EventChainPrototype to a ComponentPrototype, an unambiguous identification of event and action classes on the leaf level of the component hierarchy is supported.

5.4.5 Event Chains for Alternative Cases

With the event chain concept it is also possible to describe multiple alternative event chains for a ComponentType where data communicated over the same ports. This, for example, is the case when two RunnableEntities which are active in different ECU modes read and write the same data elements. Consequently, when describing the signal path through such a component, both alternatives need to be adequately handled.

Example: Figure 5.27 depicts an AtomicEventChainType with two RunnableEntities. Both RunnableEntities perform a data transformation of DataElementPrototype **a** to DataElementPrototype **b**.

Figures 5.27(a) and 5.27(b) also show two different AtomicEventChainTypes, **AtomicEventChainType1** and **AtomicEventChainType2**, which describe the concrete order of RTEAPIActions that must be observed such that a data transformation can be monitored in both cases. The alternative AtomicEventChainTypes can be used in different CompositeEventChainTypes, e.g., to monitor the signal path from **a** to **b** in different system modes (modes are not shown in the figures). □



(a) AtomicEventChainType for RunnableEntity RunnableEntity1 employing implicit Sender/Receiver communication (b) AtomicEventChainType for RunnableEntity RunnableEntity1 employing explicit Sender/Receiver communication

Figure 5.27: Example for specification of distinct AtomicEventChainTypes for the InternalBehavior of an AtomicSoftwareComponentType

5.5 Top-Level Event Chains and Path Specifications

In an AUTOSAR system, the so-called top-level composition is the CompositionType that represents the logical software architecture of the AUTOSAR system. The top-level composition contains the instances of all software components that make up the application software realized by the AUTOSAR system.

Event chains that are specified for the top-level composition of an AUTOSAR system are special: they can refer to all relevant event and action classes on the leaf level of the event chain hierarchy that mark places or actions in an overall system description, reaching from a system input signal acquisition to a system output signal production. Event chains that are specified for the top-level composition are so-called *TopLevelEventChains*.

Definition 23 A *TopLevelEventChain* in an AUTOSAR system is an *EventChainType* that is specified based on the event and action classes that are visible for the top-level composition of an AUTOSAR system.

The event and action classes on the leaf level of a TopLevelEventChain can denote the relevant places and actions of a signal path, i.e. they can be used to observe how a system input signal is transformed to a system output signal. If the order information encoded in the TopLevelEventChain captures the causal order relations between the referred event and action classes correctly, then the TopLevelEventChain describes a signal path from a system input to a system output. The TopLevelEventChain is then termed a *PathSpecification*.

Definition 24 A **PathSpecification** in an AUTOSAR system is a **TopLevelEventChain** where the event and action classes on the leaf level all denote RTEAPIActions, RunnableEntityEvents, COMEvents and OSEvents that are part of the transformation of a system input signal to a system output signal. The order of the event and action classes is the causal order according to which firstly input data is transformed into output data by RunnableEntities and secondly according to which data is communicated between them according to any of the three communication patterns.

A PathSpecification is also referred to as *end-to-end signal path* where the ends denote event or action classes that mark places or actions related to a input data acquisition at a system input (sensor) and a output data production at a system output (actuator).

For PathSpecifications to be valid, they must be correct and complete:

Correct: The order of the event and action classes which is described by means of an EventChainType that denotes a PathSpecification must comply to the causal order of events that can be observed in the system. Otherwise, the PathSpecification is incorrect as it does not describe the true causal relations between events in the system.

Complete: All relevant event and action classes must be contained in the EventChainType. Otherwise, the PathSpecification is incomplete.

This means that

- not every TopLevelEventChain is a PathSpecification, but
- every PathSpecification is a TopLevelEventChain.

Example: The logical software architecture of the example AUTOSAR system for the simple control application is shown in figure 5.28.

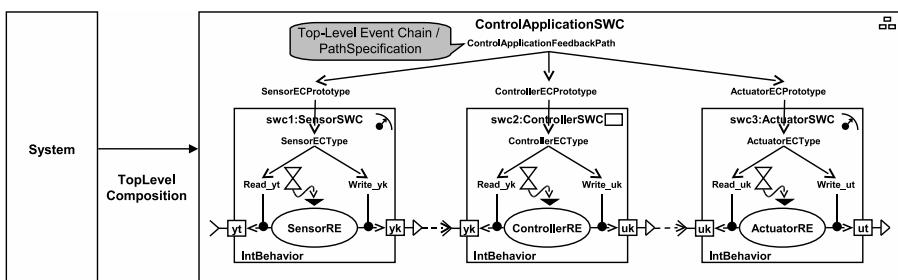


Figure 5.28: Example AUTOSAR system with a TopLevelEventChain that is a PathSpecification

The **CompositionType ControlApplicationSWC** is the top-level composition of the software architecture. The signal path from the **DataElementPrototype** *yt* to the **DataElementPrototype** *ut* is specified by means of a **CompositeEventChainType ControlApplicationFeedbackPath**. The **CompositeEventChainType** is a **TopLevelEventChain** as the **CompositionType** for which it is specified is the top-level component of the AUTOSAR system. The **TopLevelEventChain** is a **PathSpecification** as the order of the concrete RTEAPIActions encoded in the **TopLevelEventChain** complies with the causal order according to which the **DataElementPrototype** *yt* affects the **DataElementPrototype** *ut*. The **PathSpecification** precisely describes the feedback path of the simple control application.

Note that in the example, the RTEAPIActions as well as the TimingEvents that specify the periodic activation of the RunnableEntities are bound to the globally visible ideal clock. With the help of the latter, the time-triggered activation of the RunnableEntities can be planned based on a global time base. □

5.6 Timing Requirements

The previously introduced concepts allow the precise and unambiguous description of signal paths in AUTOSAR systems based on AUTOSAR-specific event and action classes. This is the basis for the introduction of concepts for the description of application-specific timing requirements. The origin of the latter have been described in chapter 4.1 on foundations of control theory.

We distinguish three different types of timing requirements that are all described based on formal path specifications:

End-to-End Delay Requirements End-to-end delay requirements express that the latency between an input data acquisition and a related output data production that is based on the input data should be bound by a certain deadline (reactive real-time applications), or that the delays should have a specific constant nominal value whereby some tolerances in the form of a jitter value is allowed (control applications).

Interval Delay Requirements Interval delay requirements specify that the latency between two effective input data acquisition actions or between two effective output data production actions should be bound by a certain nominal value. This type of timing requirement specifically targets the continuous timing behavior of control applications.

Synchronization Requirements Synchronization timing requirements express that effective instances of certain event or action classes should occur in a synchronized temporal relation to each other, i.e. within a certain interval. Synchronization timing requirements are specified for event or action classes where the system directly interacts with its environment, e.g. when multiple sensors acquire a new input value from the same plant process or multiple actuators effectuate a new value on the same plant process.

5 Specification of Timing Requirements based on Formal Path Specifications

The meta-model specification for these abstract types of timing requirements can be found in Appendix E.

To specify timing requirements for an AUTOSAR system, they need to be expressed in the context of the AUTOSAR system. For this, we introduce a new container for timing requirements which aggregates all information on application-specific timing requirements and which belongs to an AUTOSAR system description.

Example: Figure 5.29 depicts an example AUTOSAR system with the timing requirements container and three placeholder timing requirements. The general idea is that attributes which characterize the concrete timing requirements refer to the globally visible ideal clock such that timing requirements are expressed in a definite temporal context. This is shown by the dashed lines in figure 5.29.

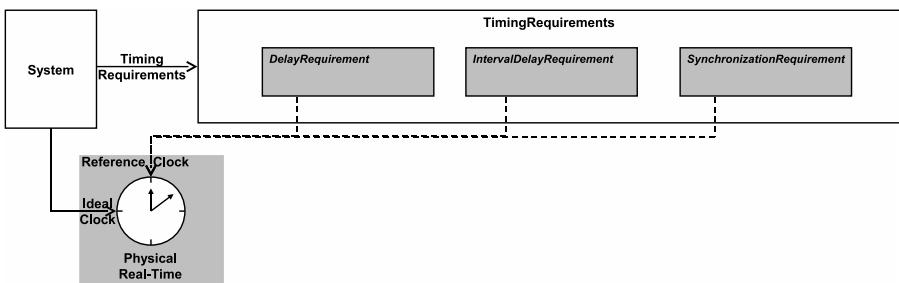


Figure 5.29: Integration of timing requirements with AUTOSAR system description

□

Section 5.6.1 discusses the notion of end-to-end delay requirements in the context of reactive real-time applications and control applications and introduces the means to describe them in relation to a signal path specification. Section 5.6.2 then introduces the concepts to describe interval delay requirements in relation to a signal path specification. For the description of synchronization timing requirements, multiple PathSpecifications and specific event and action classes to which they refer must be considered in relation to each other. Concepts to express such timing requirements are introduced in section 5.6.3.

5.6.1 End-to-End Delay Requirements

A single PathSpecification can be associated with a timing requirement which constrains the delays of an effective data transformation of an input signal to an output signal along a signal path.

In the different engineering disciplines which deal with real-time applications such as reactive real-time systems or control engineering, different perceptions exist for how timing requirements are expressed. This results in different concepts used to

describe such timing requirements. We thus distinguish two concrete kinds of end-to-end delay requirements: the *ReactionTimeDelayRequirement* for reactive real-time applications and the *PathDelayRequirement* for control applications.

Reaction Time Delay Requirement for Reactive Real-Time Applications

Reactive real-time applications express timing requirements in terms of a *deadline* or *response time* between a stimulus at a system input and a response on a system output. The *ReactionTimeDelayRequirement* is a timing requirement on the minimum and maximum reaction time that can be measured between effective instances of the first and last event classes of a PathSpecification.

To relax the notion of a strict absolute deadline we allow the specification of a *minimum* and *maximum* value, meaning that the reaction time must fall within a certain bounded range of time values. Often, the minimum value is set to zero, and only the maximum value is specified as a time value greater than zero which denotes the absolute deadline.

Path Delay Requirement for Control Applications

Control applications express timing requirements on signal paths in terms of *feedback path delays* between system inputs acquired at a sensor and system outputs effectuated at an actuators. As discussed in section 4.1.7, in discrete-time, time-invariant control applications, the feedback path delay should either be negligibly small compared to the sampling rate, or constant such that it can be adequately incorporated in the design of the control application. A PathSpecification describing a feedback path can thus be associated with a *PathDelayRequirement* which expresses that a specific *nominal* value should be maintained for the feedback path delay. Relaxations of the notion of a strict nominal value for the feedback path delay are expressed in terms of an additionally allowed tolerance that is termed *jitter*. The jitter denotes the maximum positive or negative deviation from the nominal value and thus expresses a margin around the nominal value in which the end-to-end delay is still considered as being valid.

Example: Figure 5.30 depicts the logical software architecture of the example AUTOSAR system for the simple control application. A PathSpecification is used to describe the feedback path from the SensorSWC over the ControllerSWC to the ActuatorSWC. The PathSpecification is associated with a PathDelayRequirement, expressing that the nominal feedback path delay should be 5 ms with an allowed deviation of 2 ms.

□

5.6.2 Interval Delay Requirements

Discrete-time control applications require that certain timing properties are maintained continuously during runtime. As discussed in section 4.1.7, input signals must

5 Specification of Timing Requirements based on Formal Path Specifications

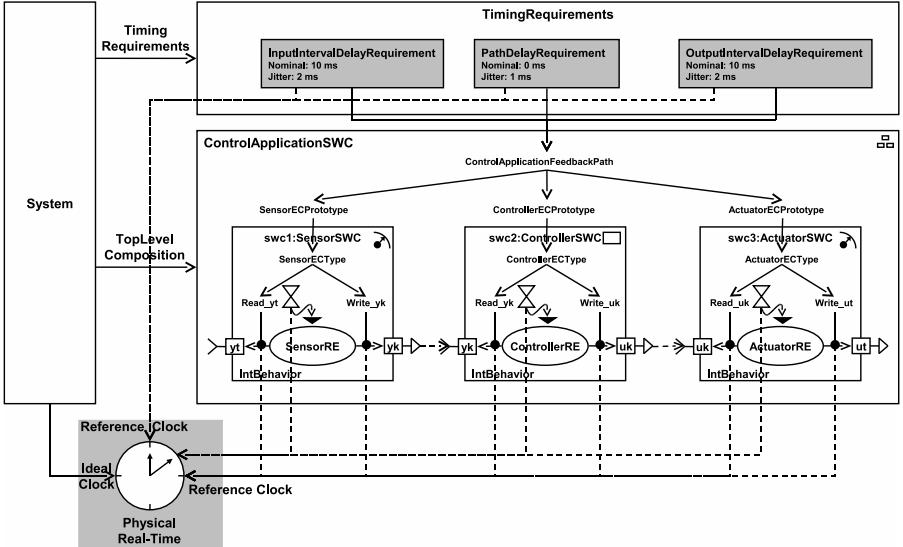


Figure 5.30: Example for an AUTOSAR software architecture with a PathSpecification and an associated PathDelayRequirement

be sampled at a constant nominal rate, and output signals must also be actuated at a constant nominal rate. In both cases, certain deviations are acceptable. In order to express such timing requirements, *IntervalDelayRequirements* are employed.

An IntervalDelayRequirement specifies that the latency between two effective consecutive instances of one and the same event or action class should be constant whereby an allowed deviation is acceptable. The size of the IntervalDelayRequirement is specified in terms of a *nominal* value, and the deviation is expressed in terms of a *jitter* value.

With respect to a PathSpecification that denotes the feedback path of a control application, the most relevant event or action classes are its first and its last ones. For the former, we refer to the specific kind of IntervalDelayRequirement as *InputIntervalDelayRequirement* as the first event or action class points to where new input is acquired (e.g., the start of a sampling action). For the latter, we refer to it as *OutputIntervalDelayRequirement* as the last event or action class points to where new output is provided (e.g., the end of an actuation action). As these relevant event or action classes can be directly identified for a PathSpecification as it is in principal an ordered set of event and action classes, it is sufficient to refer to the PathSpecification rather than the concrete event or action class on its leaf level.

Example: The previous example shown in figure 5.30 also depicts two IntervalDelayRequirement that are associated with the feedback path of the simple control

application. The InputIntervalDelayRequirement specifies that input data should be acquired at a nominal rate of 10 ms whereby a deviation of 1 ms is acceptable. The same holds for the OutputIntervalDelayRequirement to describe the timing requirement on the effective actuation rate. \square

5.6.3 Synchronization Requirements

Control applications interact with the environment through input and output devices. If multiple input and output devices are employed as in multiple-input-multiple-output (MIMO) control applications, timing requirements arise where the input data acquisition of multiple sensors need to be synchronized, and the output data effectuation of multiple actuators need to be synchronized.

In applications with multiple inputs and outputs, in principle, there are multiple signal paths from a single input to a single output (see foundations of control theory in section 4.1.7). Conceptually, these can be described by multiple independent PathSpecifications.

When multiple PathSpecifications are considered in relation to each other, they can overlap and thus intersect each other. This is the case if the PathSpecifications refer to event and action classes on their leaf level which are common to all PathSpecifications. In general, the common event and action classes can be interpreted as *IntersectionPoints* then.

An intersection point can be a *JoinPoint* or a *ForkPoint*, or even both. This depends on the fact whether there are common event and action classes in the PathSpecifications before (*ForkPoint*) or after (*JoinPoint*) the intersection point. Due to the distinction of intersection points into join points and fork points, the related path segments can also be distinguished into *JoinPathSegments* and *ForkPathSegments*.

The synchrony of event or action classes which are referred to by the PathSpecifications can only be considered with respect to a common intersection point. The intersection point serves as common reference from which the size of the synchronicity is measured.

In order to express timing requirements on the synchronization of multiple event or action classes, we have introduced a specific type of timing requirement, the *SynchronizationRequirement*.

A SynchronizationRequirement expresses that instances of a specific set of event and action classes which are part of distinct PathSpecifications must occur within a given interval measured with respect to the instances of a common event or action class which is part of all PathSpecifications, i.e. the *IntersectionPoint*. For this, a SynchronizationRequirement specifies a nominal value as the size of the interval in which the event or action classes which should be synchronized to each other must occur.

As outlined above, intersection points can be distinguished into JoinPoints and ForkPoints. With respect to the SynchronizationRequirement, this leads to the distinction of the *InputSynchronizationRequirement* and the *OutputSynchronizationRequirement*.

Input Synchronization Requirement

In the case of an InputSynchronizationRequirement, a common event or action class must exist which is a JoinPoint of multiple JoinPathSegments leading to the JoinPoint. Each JoinPathSegment must refer to a different PathSpecification. The event or action class which acts as the JoinPoint is identified by a reference⁶ which points to the event or action class on the leaf level of the event chain hierarchy. The event or action class which shall be synchronized with event or action classes of other join path segments is referred to as *StartPoint* and also referenced by an instance reference. A detailed meta-model specification of the instance references can be found in Appendix E.

Output Synchronization Requirement

The OutputSynchronizationRequirement is very similar to the InputSynchronizationRequirement and only distinguishes certain details. A common event or action class must exist which is a ForkPoint of multiple ForkPathSegments commencing from the ForkPoint. Each ForkPathSegment must refer to a different PathSpecification. The event or action class which shall be synchronized with event or action classes of other ForkPathSegments is referred to as *EndPoint*. As in the case for the InputSynchronizationRequirement, the ForkPoint and the EndPoint are referenced by the ForkPathSegment through instance references. Detailed meta-model specifications for the instance references can be found in Appendix E.

Example: Figure 5.31 depicts an excerpt of an example software architecture of an AUTOSAR system with two sensor software components (**swc1**, **swc2**) and a voter software component (**swc3**). The idea is the sensor software components acquire data (**yt**) from the same plant process which is then send to the voter software component. Here, the values are compared to each other to check if they are plausible and consequently merged into a single data element (**yk**) as output of the voter software component).

The signal paths from the two input signals of the sensor software components (**yt**) to the voted signal provided by the voter software component (**yk**) are described by two distinct PathSpecifications. An InputSynchronizationRequirement is modeled to express that the two sensor data acquisition actions should be synchronized with each other. The InputSynchronizationRequirement has two JoinPathSegments that each refer to a common JoinPoint (**Write_yk** in the voter software component) and two distinct StartPoints (**Read_yt** in the two sensor software components). The InputSynchronizationRequirement expresses that the reading actions of the two RunnableEntities **SensorRE** must be synchronized with the writing action of the RunnableEntity **VoterRunnable** within an interval of 3 ms. □

⁶Note that this is actually an instance reference in the AUTOSAR sense. The instance reference is necessary to unambiguously identify the event or action class by capturing its context in terms of the event chain hierarchy down to the leaf level.

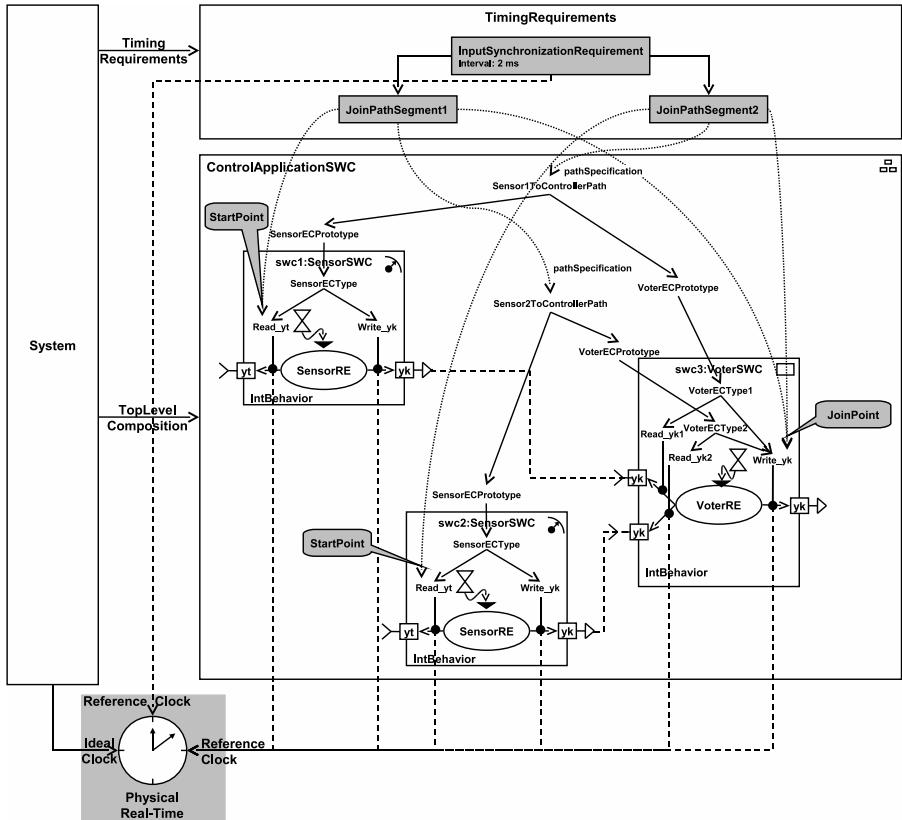


Figure 5.31: Example for an AUTOSAR software architecture with two CompositeEventChainTypes and an associated InputSynchronizationRequirement

5.7 Integration with Views, Development Methodology and Templates

In order to integrate the introduced concepts of the AUTOSAR-specific event and action classes, the concepts to describe paths specifications by means of hierarchical event chains based on the latter, and the concepts to express timing requirements based on precise path specifications seamlessly with the existing AUTOSAR concepts, several other aspects which go beyond the definitions and meta-model specifications need to be considered. These are the integration of the concepts with the AUTOSAR views for the planning and the realization phase (Virtual Functional

Bus view and System view), the AUTOSAR development methodology and the organization of concepts in the existing AUTOSAR templates. In the following, these aspects are discussed and their integration is described.

5.7.1 Views

As described in section 4.3.4, AUTOSAR defines two views for the different phases of an AUTOSAR system development: the planning phase with a logical system perspective (VFB view), a realization phase with a technical perspective (System view).

Table 5.4 shows which of AUTOSAR-specific event and action classes are available for the description of PathSpecifications in the two development phases, i.e. under the two distinct views, respectively.

	Virtual Functional Bus view	System view
RTEAPIActions	✓	✓
RunnableEntityEvents	✓	✓
OSEvents	—	✓
COMEvents	—	✓

Table 5.4: Availability of AUTOSAR-specific event and action classes under the specific AUTOSAR view

In the System view, all information which is required to realize an AUTOSAR system is available. This especially means that the mapping-specific information such as the SWC-to-ECU mapping information and the information about the data-mappings are available (system configuration). Furthermore, information on the basic software configuration such as the OS-tasks and their priorities is available (ECU configuration). This implies that all AUTOSAR-specific event and action classes that we defined for AUTOSAR are available in the System view. These are the RTEAPIActions, RunnableEntityEvents, OSEvents and COMEvents.

In the VFB view, the information stemming from system configuration and ECU configuration is abstracted, thus only a subset of the AUTOSAR-specific event and action classes that have been defined for AUTOSAR are available. These are the RTEAPIActions and the RunnableEntityEvents. OSEvents and COMEvents are only available on the System view.

The availability of the different types of AUTOSAR-specific event and action classes has implications on what kinds of signal paths can be described in the planning phase (VFB view) and the realization phase (System view). Due to that, we introduce the notion of *logical* PathSpecifications and *concrete* PathSpecifications.

Definition 25 A *logical PathSpecification* is a PathSpecification that is defined for the logical software architecture of an AUTOSAR system (VFB view). A *logical*

PathSpecification can refer to RTEAPIActions and RunnableEntityEvents on its leaf level.

RTEAPIActions and RunnableEntityEvents can thus be used during the planning phase for the description of logical PathSpecifications.

OSEvents and COMEvents are transparent from the VFB view and must be introduced as a consequence of the mapping decisions that lead to the System view. These event classes do not need to be explicitly considered during the planning phase they can be automatically inserted at the correct places into the concrete order of event and action classes. Logical PathSpecifications which are augmented by OSEvents and COMEvents are termed *concrete* PathSpecifications.

Definition 26 A *concrete PathSpecification* is a PathSpecification that is defined for the technical realization of an AUTOSAR system (System view). A *concrete PathSpecification* can refer to RTEAPIActions, RunnableEntityEvents, COMEvents and OSEvents on its leaf level.

By means of the application-specific timing requirements that have been integrated with the concept to describe event chains it is possible to describe concrete timing requirements for the application software on the VFB view. This is independent from concrete configuration information of the system mapping (SWC-to-ECU mappings, signal-mappings).

5.7.2 Development Methodology

The integration of the timing model concepts and AUTOSAR views has implications for how the definition of concrete event and action classes and the definition of event chains can be integrated into the AUTOSAR development methodology. In the following, we describe how a seamless integration is achieved.

RTEAPIActions and RunnableEntityEvents as well as AtomicEventChainTypes are to be modeled in the course of the specification of the structure and behavior of individual AtomicSoftwareComponentTypes. The order of event and action classes, especially RTEAPIActions, specified in terms of an AtomicEventChainType can be considered as part of the contract for the implementor of the AtomicSoftwareComponentType. In the implementation of the RunnableEntities of an AtomicSoftwareComponentType, the RTE API functions must be invoked in the order as defined by the AtomicEventChainType. In the course of the establishment of a component hierarchy, composite event chains then need to be introduced. This allows the reuse of information on the order of RTEAPIActions whenever the corresponding AtomicSoftwareComponentType is reused.

The concrete event classes for OS objects (OSEvents) and inter-ECU communication (COMEvents) are then introduced after the SWC-to-ECU and data mappings as well as the basic software module configurations for the basic software modules OS, RTE and COM are established. This can be performed automatically as all required information is available: the logical PathSpecifications from VFB view plus the system mapping and ECU configuration information.

5 Specification of Timing Requirements based on Formal Path Specifications

Figure 5.32 depicts the overview of the first part of the AUTOSAR methodology (compare to section 4.3.3) where our Timing Model concepts are integrated.

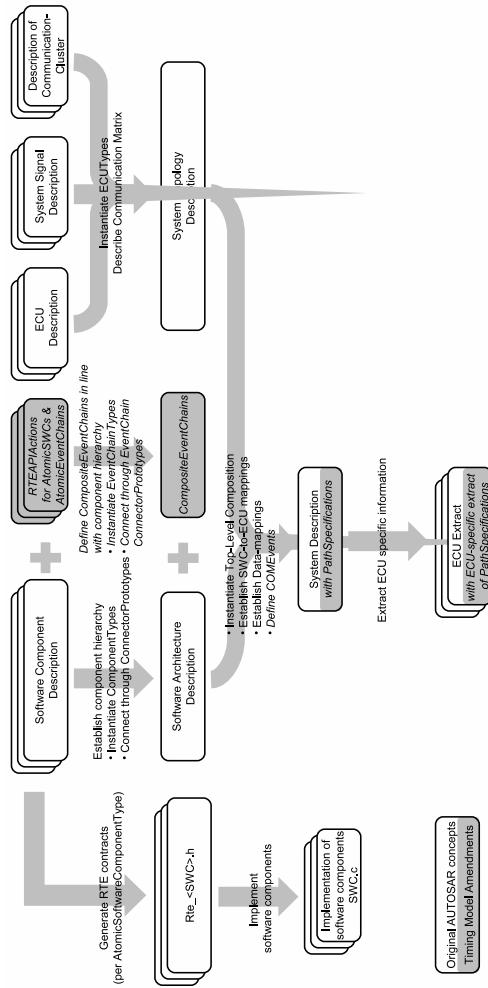


Figure 5.32: Overview of AUTOSAR methodology with integrated timing model concepts

The SystemDescription can be complemented with application-specific TimingRequirements which are defined for the logical PathSpecifications.

5.7.3 Templates

As described in section 4.3.1, the AUTOSAR concepts are organized in so-called templates that contain different sets of information of the model of an AUTOSAR system. The templates conceptually are related to different phases of a development and are thus integrated with the AUTOSAR development methodology. As described in section 5.7.1, only a subset of the AUTOSAR-specific event and action classes are available on Virtual Functional Bus view and thus available in the planning phase of an AUTOSAR system. Thus, a reasonable assignment of the AUTOSAR-specific event and action classes to the templates is as follows:

- The RTEAPIActions and RunnableEntityEvents are integrated in the SoftwareComponentTemplate. They can be used in both the planning and the realization phase.
- The OSEvents, COMEvents are integrated in the SystemTemplate. They are only available realization phase, after the system configuration and the ECU basic software configuration have been performed.
- The concepts for hierarchical event chains and logical path specifications are integrated with the Generic Structure.
- The concepts for expressing application-specific timing requirements are also integrated with the SystemTemplate. They can be used in the planning and realization phase.

Figure 5.33 depicts the relation of the AUTOSAR templates as presented in section 4.3.1 with the integration of the timing model concepts.

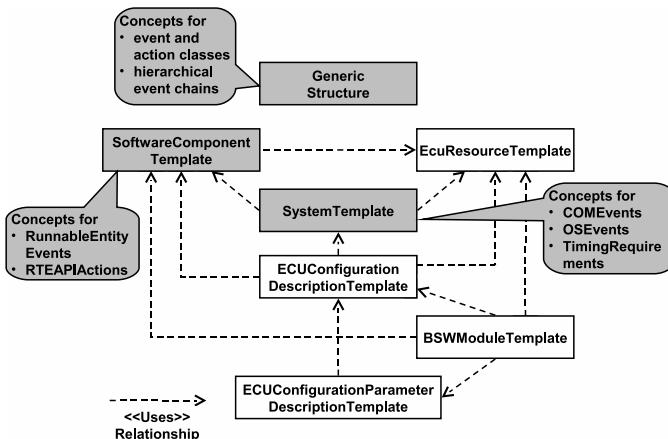


Figure 5.33: AUTOSAR Templates with extensions for AUTOSAR timing model

6 Specification-Based Instrumentation of AUTOSAR Systems for Simulation and Monitoring

6.1 Introduction

In order to determine the timing properties of real-time applications by means of simulation- and monitoring-based approaches, it is required to capture instances of event and action classes that characterize its behavior. For monitoring approaches that employ software instrumentations, an instrumentation of the code making up the technical real-time system that realizes the application is required to capture instances of events and actions. Other approaches such as hardware-based or hybrid monitoring where an event-detection hardware is used could be employed as well, however, they require more effort to be realized and do not straight-forward integrate with the existing AUTOSAR concepts such as the Virtual Functional Bus Tracing Mechanism. As we are interested in timing requirements that relate to signal paths, such an instrumentation can be derived from signal path specifications by which the relevant event and action classes that need to be monitored are referred. For AUTOSAR systems, the concepts to describe logical PathSpecifications on the basis of the relevant RTEAPIActions have been introduced in chapter 5. To monitor instances of an event or action class, instrumentation code must be placed at the right places of the code of the real-time application. For the instrumentation of an AUTOSAR system, the Virtual Functional Bus Trace hook functions that resemble the respective AUTOSAR-specific event and action classes already denote the right places for the monitoring of their instances. The hook functions need to be adequately implemented such that instances of the latter are captured in an event trace that can then be further analyzed.

The question that is addressed in this chapter is how an instrumentation can be derived from the PathSpecifications that are described for an AUTOSAR system. These PathSpecifications are first described as logical PathSpecifications that capture logical causal relations based on the RTEAPIActions that lie on the signal path within the logical software architecture of the AUTOSAR system. In order to have all required information for the generation of an instrumentation at hand, at first, a concrete PathSpecification that adequately captures the concrete causal relations between all relevant event and action classes needs to be derived. For that, two steps need to be performed: At first, the logical PathSpecification which is a hierarchical event chain specified in the context of the top-level composition of an AUTOSAR system needs to be converted to a flat ordered set of RTEAPIActions

6 Instrumentation of AUTOSAR Systems for Simulation and Monitoring

that are referred to on the leaf-level of the event chain hierarchy. Then, additional OSEvents and COMEvents need to be introduced into the flat ordered set of RTEAPIActions to adequately address the consequences of the System Mapping and ECU configuration decisions on the concrete signal paths, i.e. to account for the specifics of intra-task, inter-task and inter-ECU communication. This step is required to ensure direct causal relations between all event and action classes of the concrete PathSpecification. The generation of the software instrumentation of an AUTOSAR system based on a set of concrete PathSpecifications can then be conducted by implementing the respective Virtual Functional Bus Trace hook functions with calls to an additional event logging software.

Figure 6.1 provides an overview of the approach for the derivation of instrumentations for simulation and monitoring from the PathSpecifications of an AUTOSAR system.

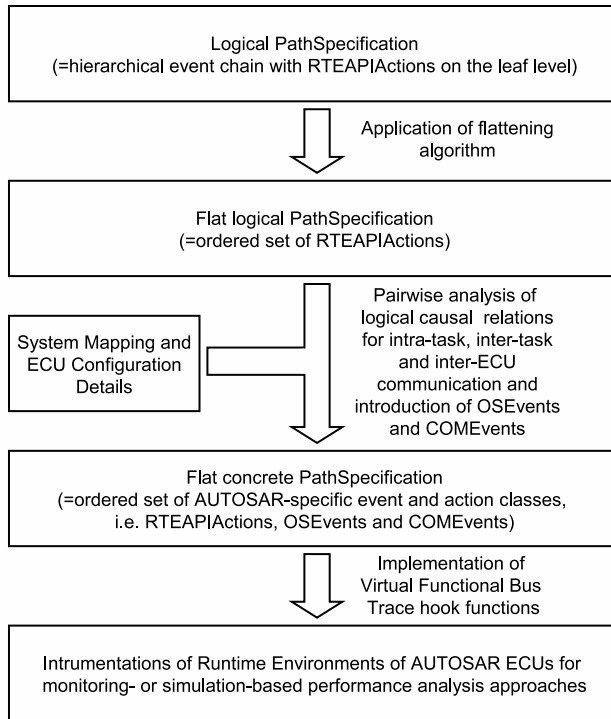


Figure 6.1: Overview on specification-based instrumentation of AUTOSAR systems for simulation and monitoring

In the following, we first describe how a flat ordered set of RTEAPIActions can be derived from a logical PathSpecification (section 6.2). Then, the determination of

concrete PathSpecifications from logical PathSpecifications is described (section 6.3). The generation of an instrumentation is described for an example simulation tool (chronSim from INCHRON GmbH) and an example monitoring tool (RTA-TRACE from ETAS GmbH). This is explained in section 6.4.

6.2 Derivation of Flat Ordered Sets of Runtime Environment API Actions from Logical Path Specifications

In order to derive a flat ordered set of RTEAPIActions from a logical PathSpecification of an AUTOSAR system, the corresponding CompositeEventChainType that is specified for the top-level composition of the AUTOSAR system needs to be analyzed. A CompositeEventChainType encodes the order information on its referred event and action classes in a hierarchical way. This order information needs to be extracted and for this, a special algorithm is employed. A recursive in-order traversal algorithm that works on the tree-shaped data structure of an hierarchical event chain is introduced as shown in Listing 6.1.

6 Instrumentation of AUTOSAR Systems for Simulation and Monitoring

```

eventChainPrototypesOrdered :=
    [eventchainType.eventChainConnectorPrototypes[i].precedingEvents] +
    eventChainPrototypesOrdered
31     }
32 }
33 }
34 }
35 #To 2:
#For each event chain prototype referred by the composite event chain type, add the
RTEAPIActions on its leaf level to the result.
36 #For this, recursively call the algorithm on the event chain type of the event chain
prototypes.
37 eventChainPrototypesOrdered.each {
38     |eventChainPrototype|
39         result +:= deriveOrderedSetOfRTEAPIActionsFromEventChainType(
40             eventChainPrototype.eventChainType)
41     }
42 }
43 return result
44 end
45

```

Listing 6.1: Algorithm to derive a flat ordered set of RTEAPIActions from a logical PathSpecification

To extract the RTEAPIActions on the leaf level in the encoded order, the algorithm performs a recursive in-order traversal on the tree-shaped data structure of the given EventChainType. Depending on the type of the given event chain, either the base case (atomic event chain) or the recursive case (composite event chain) of the algorithm is considered. In the base, i.e. the event chain is an AtomicEventChainType, the ordered set of RTEAPIActions is the ordered set of RTEAPIActions which are directly referred by the AtomicEventChainType. In the recursive case, i.e. the event chain is a CompositeEventChainType, the contained EventChainPrototypes need to be traversed in the order determined by the EventChainConnectorPrototypes, and for each contained EventChainPrototype, the RTEAPIActions on its leaf level need to be determined and appended to the result. This is performed by recursively calling the algorithm. The algorithm finally returns the ordered set of RTEAPIActions. In the following, we will use the term *flat logical PathSpecification* also synonymously for the flat ordered set of RTEAPIActions that can be determined from a logical PathSpecification.

Example: Figure 6.2 depicts the logical software architecture of the AUTOSAR system for the simple control application. The feedback path of the control application is specified by the PathSpecification `ControlApplicationFeedbackPath`. Note that in contrast to previously shown figures, the EventChainConnectorPrototypes are explicitly depicted.

To derive the flat ordered set of RTEAPIActions that equivalently describes logical PathSpecification, the algorithm from Listing 6.1 is applied to the CompositeEventChainType `FeedbackPathSpecification`. As there are two EventChainConnectorPrototypes, these are evaluated first, and the order of the EventChainPrototypes

6.2 Derivation of Flat Ordered Sets of Runtime Environment API Actions

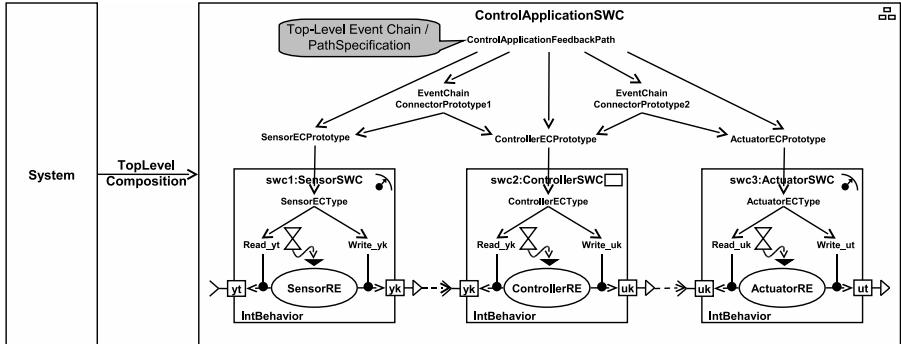


Figure 6.2: Logical software architecture of the AUTOSAR system for the control application example

that the CompositeEventType consists of is determined. This results in an ordered set where **SensorECPrototype** comes before **ControllerECPrototype** which in turn comes before **ActuatorECPrototype**. This set is then traversed in order and for each of the EventChainPrototypes, their event chain type is determined and from that, the flat ordered set of RTEAPIActions on their leaf-level is determined. As these are all AtomicEventChainTypes, the ordered set of the RTEAPIActions is directly returned. During the recursive ascendance of the algorithm, this is then used to construct the flat ordered set of RTEAPIActions denoted by the logical PathSpecification. This complies to [Read_yt, Write_yk, Read_yk, Write_uk, Read_uk, Write_ut] for the example. □

As an alternative graphical notation to the textual description of the flat ordered set of RTEAPIActions, we employ a simple directed graph where the nodes correspond to the RTEAPIActions and the edges to the logical causal relations between the RTEAPIActions. For the logical PathSpecification of the example AUTOSAR system, the flattened version is shown in figure 6.3.

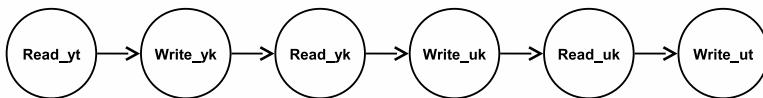


Figure 6.3: Flat logical PathSpecification as directed graph for the example AUTOSAR system

6.3 Derivation of Concrete Path Specifications based on Flat Logical Path Specifications

For the instrumentation of all event and action classes that characterize a signal path in an AUTOSAR system, the logical causal relations that are encoded in the logical PathSpecifications on the basis of the logical software architecture need to be translated into concrete causal relations that also hold in the technical realization of the AUTOSAR system.

As described in the foundations of AUTOSAR, the decisions taken during system configuration and ECU basic software configurations have an influence on the concrete realization of the communication patterns in an AUTOSAR system implementation. This is shown in figure 4.24 in section 4.3.6. For the three different cases of intra-task, inter-task and inter-ECU communication, the concrete realizations of the communication patterns are discussed.

It is thus required to adequately treat the special cases of intra-task and inter-task communication as well as inter-ECU communication. Furthermore, special care must be taken for implicit Sender/Receiver and Interrunnable communication. In both cases, the logical causal relations between two RTEAPIActions do not necessarily comply with the concrete causal relations in the technical realization. Implicit communication is realized through a specific implementation concept where copies of the actual data is provided and need to be adequately respected.

The concrete realizations of the communication patterns are further discussed on the basis of simple examples. Due to the large number of cases that need to be considered, we only provide a discussion for Sender/Receiver communication here. Interrunnable communication can be considered as a special case of Sender/Receiver communication and is explained in Appendix E. Client/Server communication is treated special in the case of intra-task communication and inter-task communication. The treatment of Client/Server communication is not explicitly explained in detail in this thesis as there are many cases that would need to be distinguished and where the level of detail would go beyond the scope. The different cases discussed for the Sender/Receiver communication example make clear what distinguishes intra-task, inter-task and inter-ECU communication with respect to the concrete realization of the logical communication and the concrete causal relations between event and action classes.

Figure 6.4 depicts an example logical software architecture for an AUTOSAR system with a logical Sender/Receiver communication between two RunnableEntities. RE1 performs a write action on data element *a* that is marked by a SendDataAction *Write_a*¹. RE2 performs a read action on data element *a* that is marked by a ReceiveDataAction *Read_a*². The example is used in the following to explain how the concrete causal relations between the two RTEAPIActions *Write_a* and *Read_a* are determined based on the mapping and configuration decisions that lead from the

¹Note that this can either be a SendDataExplicitAction or a SendDataImplicitAction, depending on the fact whether implicit or explicit Sender/Receiver communication is employed.

²Note that this can be either a ReceiveDataExplicitAction or a ReceiveDataImplicitAction.

6.3 Derivation of Concrete Path Specifications based on Flat Logical Path Specifications

logical software architecture to the technical system. These are the design decisions manifested in the System Mapping and ECU Configurations of the single ECUs.

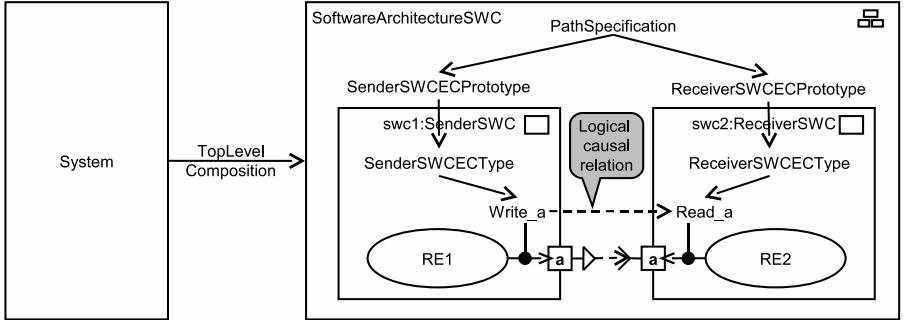


Figure 6.4: Example logical Sender/Receiver communication

6.3.1 Intra-Task Communication

Intra-task communication takes place when two RunnableEntities that communicate with each other according to the Sender/Receiver or Interrunnable communication pattern are assigned to the same OS-task on the same ECU. This requires that the two instances of the software components are mapped to the same ECU. The question is then how this communication is realized and what the concrete causal relations between the RTEAPIActions are.

In implicit Sender/Receiver communication, for each DataElementPrototype for which an implicit data access is specified (i.e., either a DataReadAccess or DataWriteAccess), a variable is provided that is local to the OS-task where the RunnableEntity that declared the implicit data access is executed. This local variable is termed a *task buffer*. The provision of OS-task-local copies is a well-known concept to ensure data-consistency in parallel programming, especially in the automotive domain (see message copy concept in [72]). The value of the task buffer variable is initialized with the latest value from the RTE right after the start of the OS-task that contains the RunnableEntity. It is written back to the RTE right before the termination of an OS-task. The copying of values between the task buffer and the RTE is performed in a thread-safe environment, i.e. interrupts are disabled during the copying process. Due to the fact when the copying takes place, namely right after the start of an OS-task before any RunnableEntity is executed, and right before the termination of an OS-task, after all RunnableEntities have finished, the start and termination of an OS-task can be employed as an alibi place for marking the copy processes.

Figure 6.5 depicts the AUTOSAR system in the realization phase with the configuration decisions (System Mapping, ECU Configuration) that lead to the fact that

6 Instrumentation of AUTOSAR Systems for Simulation and Monitoring

the logical communication is realized in terms of intra-ECU, intra-task communication. In the OS-configuration, both RunnableEntities RE1 and RE2 are assigned to the same OS-task.

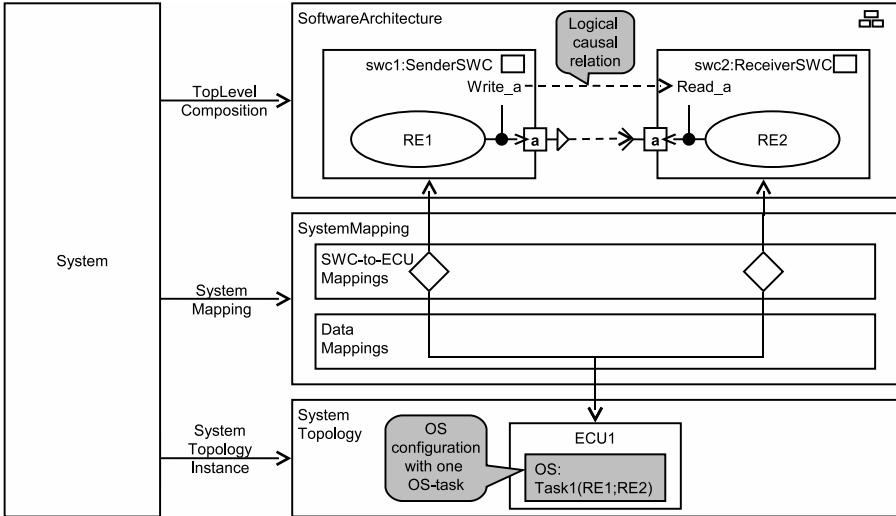


Figure 6.5: System Mapping and ECU configuration decisions leading to intra-ECU, intra-task communication of a logical Sender/Receiver communication

Figures 6.6 and 6.7 provide example execution scenarios for two possible combinations of implicit and explicit write and read actions that can be performed to establish a Sender/Receiver communication with intra-task communication.

Figure 6.6(a) depicts an execution scenario for intra-task Sender/Receiver communication with explicit read and write actions.

The communication is established directly over the RTE. RunnableEntity RE1 writes to and RunnableEntity RE2 reads from the data element **a** that is contained in the Component Instance Data Structures of the involved SenderSWC and ReceiverSWC. As shown in figure 6.6(a), there is thus a concrete causal relation between the two RTEAPIActions that mark the explicit reading and writing actions.

Figure 6.7(a) depicts an execution scenario for intra-task Sender/Receiver communication with implicit read and write actions.

The communication is established over the task buffer for DataElementPrototype **a** rather than directly over the component instance data structures of the software components that are managed by the RTE. Note that the copying of data between the task buffer and the RTE after the start of the OS-task and before the termination of the OS-task is not important in the case of intra-task communication as the communication is not established over the boundary of the OS-task. There is thus

6.3 Derivation of Concrete Path Specifications based on Flat Logical Path Specifications

a concrete causal relation between the two RTEAPIActions that mark the implicit read and write actions. This is shown in figure 6.7(b).

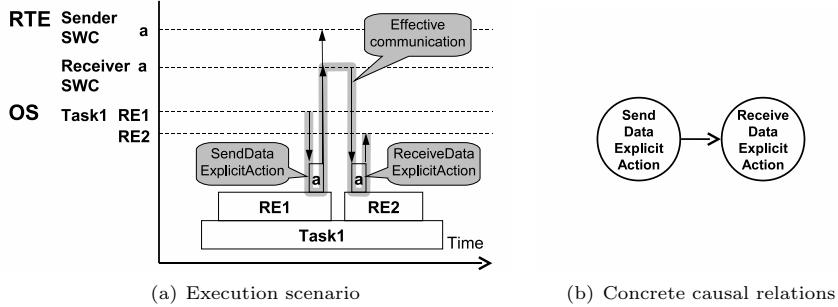


Figure 6.6: Intra-task Sender/Receiver communication: Explicit write and explicit read actions

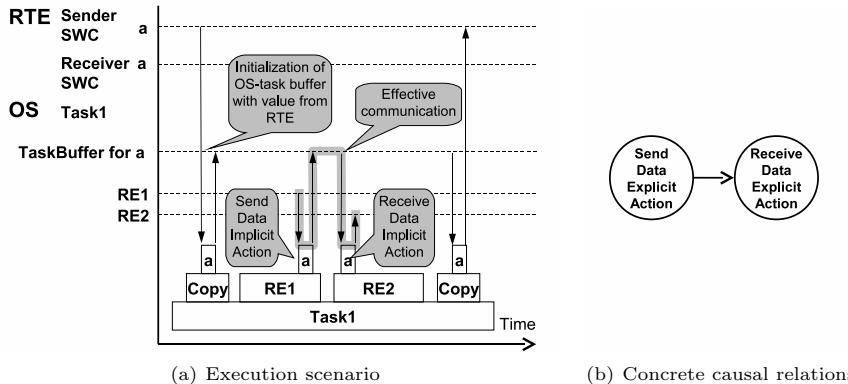


Figure 6.7: Intra-task Sender/Receiver communication: Implicit write and implicit read actions

Two special cases are when implicit and explicit read and write actions are used in combination with each other. In these cases, the copying of data between the RTE and task buffer must be adequately considered for the side where implicit communication is employed. As these two special cases are expected to occur rather rarely as it can be assumed that the read and write accesses are not in general employed in such an inconsistent way, the execution scenarios and concrete causal relations are shown in Appendix E.

6.3.2 Inter-Task Communication

Inter-task communication takes place when two RunnableEntities that communicate with each other according to the Sender/Receiver or Interrunnable communication pattern are assigned to distinct OS-tasks on the same ECU. Again, the question is then how this communication is realized and what the concrete causal relations between the RTEAPIActions are.

Figure 6.8 depicts the AUTOSAR system in the realization phase with the configuration decisions that lead to the fact that the logical communication is realized in terms of intra-ECU, inter-task communication. In the OS-configuration, the two RunnableEntities RE1 and RE2 are assigned to distinct OS-tasks, Task1 and Task2.

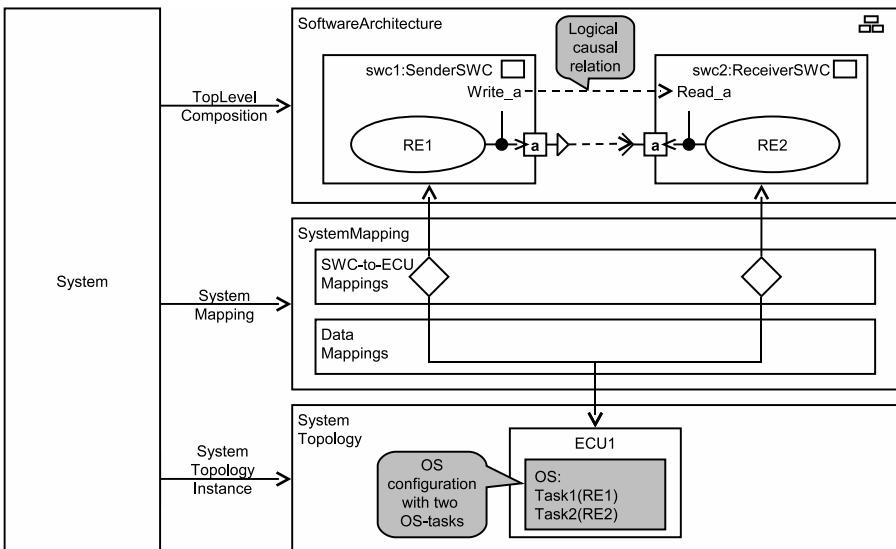


Figure 6.8: System Mapping and ECU configuration decisions leading to intra-ECU, inter-task communication of a logical Sender/Receiver communication

Figure 6.9(a) depicts an execution scenario for inter-task Sender/Receiver communication with explicit read and write actions.

The communication is established directly over the RTE by writing and reading to the DataElementPrototype **a** that is contained in the Component Instance Data Structures of the involved component instances of the **SenderSWC** and the **ReceiverSWC**. Note that the concrete causal relations shown in figure 6.9(b) are the same as for intra-task communication.

Figure 6.10(a) depicts an execution scenario for inter-task Sender/Receiver communication with implicit read and write actions.

6.3 Derivation of Concrete Path Specifications based on Flat Logical Path Specifications

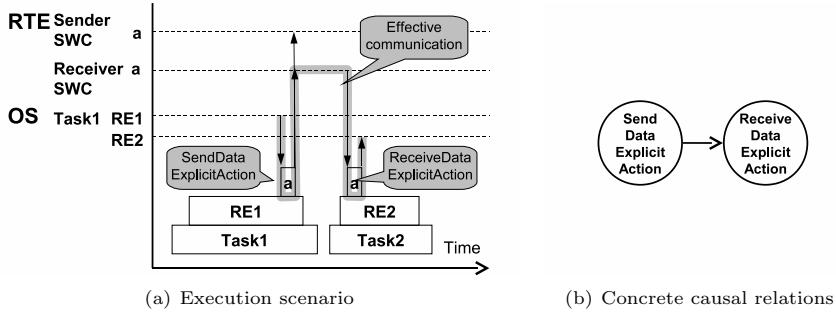


Figure 6.9: Inter-task Sender/Receiver communication: Explicit write and read actions

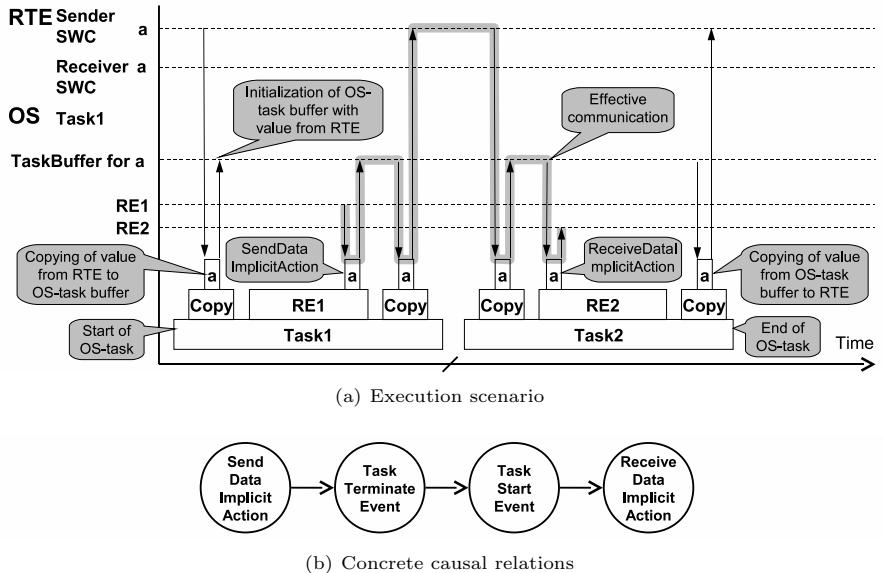


Figure 6.10: Inter-task Sender/Receiver communication: Implicit write and read actions

The communication is established over the task buffer for DataElementPrototype a. Note that the copying of data between the task buffer and the RTE after the start of the OS-task and before the termination of the OS-task is important. As the communication is established over the boundaries of the OS-tasks, the concrete

causal relations must include a TaskTerminateEvent and a TaskDispatchEvent as alibi events for the copy processes. The concrete causal relations are thus as shown in figure 6.10(b).

For inter-task communication, there are also two special cases where implicit and explicit read and write actions are combined. Again, as these are special cases that rarely occur but which require an adequate treatment, example execution scenarios and the concrete causal relations between the relevant event and action classes are shown in Appendix E.

6.3.3 Inter-ECU Communication

Inter-ECU communication takes place when the software components of two RunnableEntities that communicate with each other are mapped to distinct ECUs. Their RunnableEntities are consequently assigned to the distinct OS-tasks on the different ECUs. The question is then how this inter-ECU communication is realized and what the concrete causal relations between the RTEAPIActions are. Inter-ECU communication requires a special treatment due to the communication over a physical network. The transmission and reception of system signals is performed with the help of the communication services of the involved AUTOSAR-compliant ECUs. The cases where implicit data access is involved in the communication also need special treatment.

Figure 6.11 depicts the AUTOSAR system in the realization phase with the configuration decisions that lead to the fact that the logical communication is realized in terms of inter-ECU, inter-task communication.

The two software component instances for the SenderSWC and the ReceiverSWC are mapped to the two distinct ECUs ECU1 and ECU2 (SWC-to-ECU Mapping). The DataElementPrototype *a* which is communicated between the two RunnableEntities is mapped to a system signal *Sig_a* which is send over the vehicle network with the help of the communication services (Data Mapping). In the OS-configuration of each ECU, the RunnableEntities RE1 and RE2 are assigned to distinct OS-tasks, **Task1** and **Task2**, respectively.

Figures 6.12 and 6.13 provide example execution scenarios for two possible combinations of implicit and explicit write and read actions that can be performed to establish a Sender/Receiver communication with inter-ECU communication.

Figure 6.12(a) depicts an execution scenario for inter-ECU Sender/Receiver communication with explicit read and write actions.

The communication is over the Runtime Environments and Communication Services of the involved ECUs. The RunnableEntity RE1 of the SenderSWC performs an explicit writing action on DataElementPrototype *a*, marked by the SendDataExplicitAction, where the written value is also directly handed over to the communication service by invoking the COMSendSignal function. This leads to the transmission of the system signal *Sig_a* to which the DataElementPrototype *a* is mapped. The latter is marked by a COMSendSignalEvent and can consequently be observed during simulation or monitoring. When the system signal *Sig_a* is received on the receiver

6.3 Derivation of Concrete Path Specifications based on Flat Logical Path Specifications

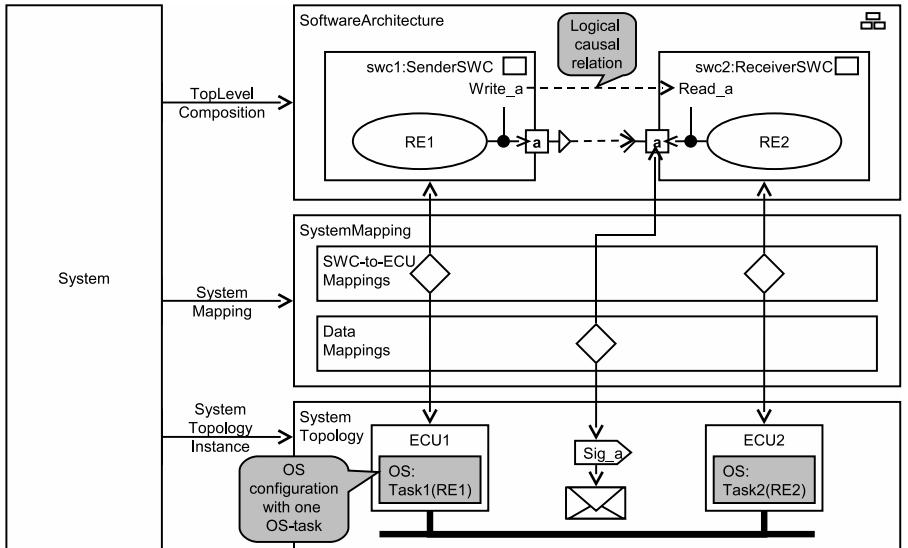
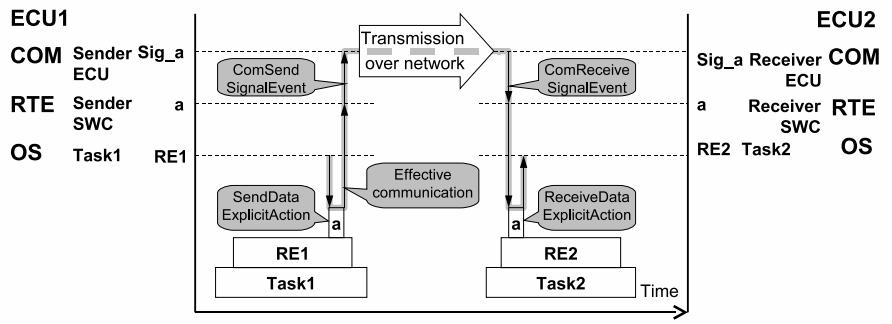
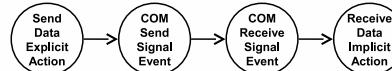


Figure 6.11: System Mapping and ECU configuration decisions leading to inter-ECU, inter-task communication of a logical Sender/Receiver communication



(a) Execution scenario



(b) Concrete causal relations

Figure 6.12: Inter-ECU Sender/Receiver communication: Explicit write and read actions

side by the respective communication service, the value of `Sig_a` is automatically provided to the RTE. When the RunnableEntity `RE2` of the `ReceiverSWC` performs a reading action (marked by the `ReceiveDataExplicitAction`), the latest value of the system signal `Sig_a` that has been received via the communication services is fetched and returned. The fetching is marked by a `COMReceiveSignalEvent`. Figure 6.12(b) depicts the chain of event and action classes that precisely characterize the concrete causal relations for this inter-ECU communication.

Figure 6.13(a) depicts an execution scenario for inter-ECU Sender/Receiver communication with implicit read and write actions.

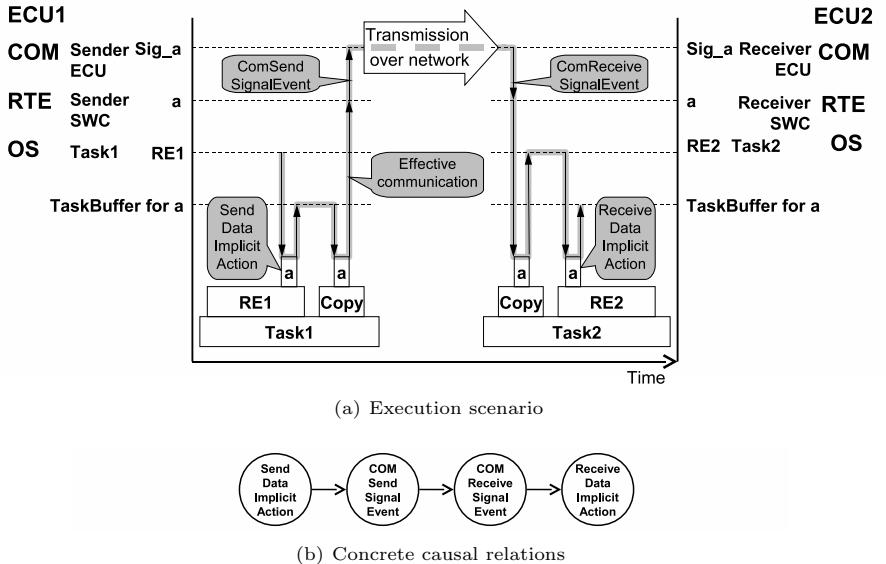


Figure 6.13: Inter-ECU Sender/Receiver communication: Implicit write and read actions

Compared to the example with explicit read and write actions, the communication is not directly established over the RTE, but over the task buffers for `DataElementPrototype a` and the Communication Services of the ECUs. The RunnableEntity `RE1` of `SenderSWC` performs an implicit writing action on `DataElementPrototype a`, marked by the `SendDataImplicitAction`. The transmission of the system signal `Sig_a`, however, is delayed until the value of the task buffer is copied to the RTE just before the end of the OS-task. This also initiates the transmission of the system signal via the communication services. The transmission of the system signal `Sig_a` is marked by a `COMSendSignalEvent` that can be observed during monitoring and simulation. When the system signal `Sig_a` is received on the receiver side by the

6.3 Derivation of Concrete Path Specifications based on Flat Logical Path Specifications

respective communication service of ECU2, this value is automatically provided to the RTE. Right after the start of the OS-task that contains RunnableEntity RE2, the most recent value of the system signal `Sig_ut` that has been received by the communication service on ECU2 is copied to the task buffer of Task2 on ECU2. The fetching of the value from the system signal is marked by a COMReceiveSignalEvent. The RunnableEntity RE2 then performs the implicit reading action that is marked by the ReceiveDataImplicitAction.

For inter-ECU communication, there are also two special cases where implicit and explicit read and write actions are combined. Again, as these are special cases that rarely occur but which require an adequate treatment, example execution scenarios and the concrete causal relations between the relevant event and action classes are shown in Appendix E.

6.3.4 Concrete Path Specifications for the Example AUTOSAR System

Figure 6.14 depicts the example AUTOSAR system for the simple control application with the relevant details on the ECU configuration (OS configuration).

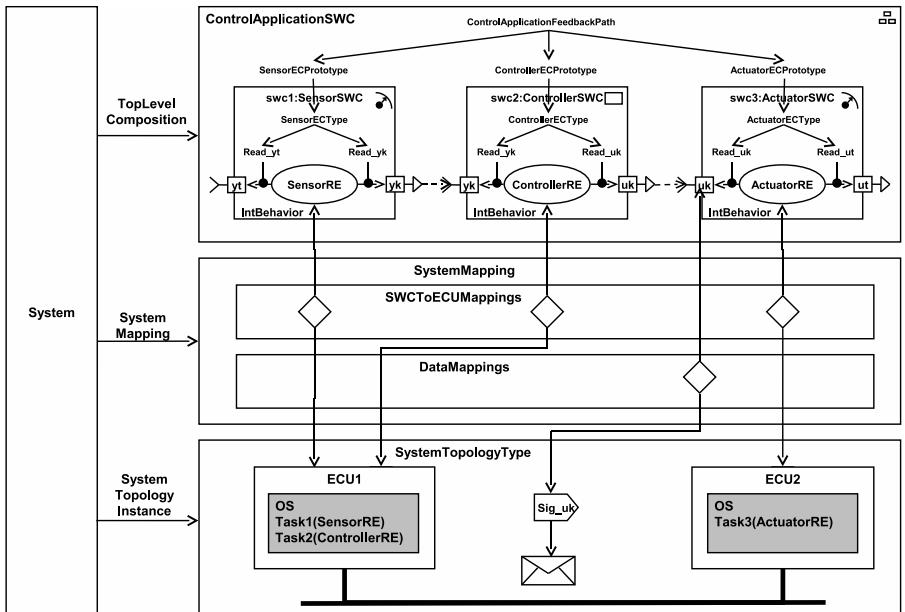
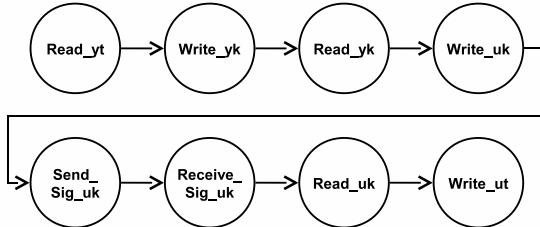


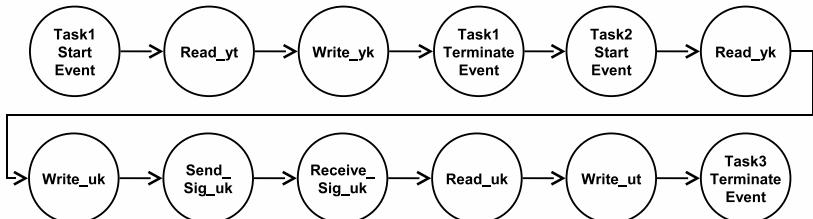
Figure 6.14: Example AUTOSAR system for the simple control application with relevant ECU configuration details (OS configuration)

6 Instrumentation of AUTOSAR Systems for Simulation and Monitoring

From the PathSpecification for the feedback path of the control application, its flat logical PathSpecification has been derived through the application of the flattening algorithm. The flat logical PathSpecification is shown in figure 6.3. From the flat logical PathSpecification, a flat concrete PathSpecification can be derived through the pairwise analysis of the logical causal relations between the RTEAPIActions and the introduction of OSEvents and COMEvents, depending on whether intra-task, inter-task or inter-ECU communication is present. The latter can be determined based on the decisions of the System Mapping and the relevant details from the ECU configurations of the single ECUs. Figure 6.15 depicts two possible flat concrete PathSpecifications that could be derived.



(a) Explicit Sender/Receiver communication



(b) Implicit Sender/Receiver communication

Figure 6.15: Flat concrete PathSpecification for the example AUTOSAR system of the simple control application

Figure 6.15(a) depicts the flat concrete PathSpecification where it has been assumed that the read and write actions performed by the RunnableEntities are all made according to the explicit Sender/Receiver communication pattern. Here, only additional COMEvents for the system signal `Sig.uk` need to be introduced to account for the inter-ECU Sender/Receiver communication. Figure 6.15(b) depicts the flat concrete PathSpecification where it has been assumed that the read and write actions are performed according to the implicit Sender/Receiver communication pattern. Additionally to the COMEvents for monitoring the transmission and reception of the system signal `Sig.uk`, also OSEvents need to be introduced in order adequately to account for the copying of values between the DataElementPrototypes

that are managed by the Runtime Environments and the task buffers belonging to the OS-tasks in which the RunnableEntities are executed.

6.4 Generation of Software Instrumentations for Simulation and Monitoring

In order to capture instances of event and action classes in simulation- and monitoring-based approaches, a software instrumentation is required that logs these instances with time-stamps that denote the points in time when the events occurred. For an AUTOSAR system, such a software instrumentation can be obtained by implementing the the trace hook functions provided by the Virtual Functional Tracing Mechanism with calls to an event logging mechanism.

As the Virtual Function Bus Tracing Mechanism is independent of any specific simulation or monitoring-based approach, the places in the source code that need to be instrumented are the same. In this section, we describe how the Trace hook functions can be implemented to log the instances of the different types of AUTOSAR-specific event and action classes.

As the event logging mechanisms of simulation and monitoring-based approaches are in principle similar, but distinct for specific simulation and monitoring tools, a general event logging mechanism is described first. As a concrete event logging mechanism for a monitoring-based approach, the event logging mechanism of the monitoring tool RTA-TRACE from ETAS GmbH is described and then used to log instances of event and action classes. RTA-TRACE was employed in several monitoring experiments during the development of our timing model concepts and also in the case study that we conducted. As a concrete event logging mechanism for a simulation-based approach, the event logging mechanism of the real-time simulation tool chronSim from INCHRON GmbH is then described. chronSim was employed in simulation experiments to show the transferability of our concepts to simulation-based performance evaluation. The generic event logging mechanism as well as the event logging mechanisms of RTA-TRACE and chronSim are described in section 6.4.1. Section 6.4.2 then describes the implementation of the Virtual Function Bus Trace hook functions that correspond to the AUTOSAR-specific event and action classes. In the implementations, calls to the logging functions of the generic event logging mechanism and the specific event logging mechanisms of RTA-TRACE and chronSim are made. This exemplifies how the trace hook functions need to be implemented. Section 6.4.3 then describes how the specification-based generation process for an instrumentation integrates with the AUTOSAR development methodology and how the source code making up the instrumentation integrates with the AUTOSAR ECU software build process.

6.4.1 Event-Logging Mechanisms

In order to log the instances of event and action classes to an event trace file that can then be further analyzed, an event logging mechanism is required. Such an event

6 Instrumentation of AUTOSAR Systems for Simulation and Monitoring

logging mechanism must be capable to detect when an event instance has occurred and store the relevant data for this in an event trace file. The relevant data is in general referred to as an event record. Among the relevant data of an event record is the unique name³ by which the event class to which the detected event instance belongs can be identified, and a time value that denotes at what point in time the event instance occurred⁴. In simulation and monitoring-based approaches, the latter is provided as the reading of a clock such that timing properties, e.g. temporal distances, can be calculated.

In simulation and monitoring-based approaches, multiple clocks of with different characteristics can be present in a system. In principle, each of these clocks can be used for the provision of a time value for an event instance. However, in order to determine a valid temporal order on a set of event instances that have potentially occurred in different computational nodes of a parallel and distributed system, the time values must be comparable to each other with respect to the same or a common time base.

In real-time simulation-approaches, a common time base can be established by a globally visible ideal clock based on what all other reference clocks in the system are defined. In the timing model concepts that we introduced for AUTOSAR in chapter 5, this is the globally visible ideal clock of an AUTOSAR system to which all event and action classes are implicitly bound during the planning phase.

Figure E.29 in Appendix E⁵ depicts an overview on the logging of event instances in simulation-based approaches. When an instance of an event is logged by the software instrumentation on an ECU, it is associated with a time value which is the reading of the globally visible ideal clock.

In monitoring-based approaches, there are different possibilities for providing a time value for an event instance. This depends on the technical realization of the real-time system either as a system with a single computational node with a processing unit that is driven by a single real-time clock, or as a parallel and distributed system with multiple computational nodes and consequently processing units driven by multiple real-time clocks. In the case of a real-time system with a single processing unit, the real-time clock that drives the processing unit can be taken as a real-time clock from which the time values of event instances are taken as readings. An example for such a monitoring approach is shown in figure E.30 in Appendix E.

In the case of a parallel and distributed real-time system, there are in principle two alternatives. The first alternative assumes that the real-time clocks that drive the single processing units are adequately synchronized with a sufficiently small precision such that when instances of events are associated with readings of these clocks it can clearly be distinguished in which temporal order the event instances occurred. Figure E.31 in Appendix E shows an example for such a monitoring approach.

³This is also referred to as an *identifier* or *label* in different approaches.

⁴A third attribute that is mentioned by is the location that describes where this attribute occurred. For our purposes, we assume that the location is clear from the event or action class that marks a place or action in a system and that it can be determined based on the unique name of the event or action class.

⁵Note that due to the size of the figures they are provided in the Appendix.

6.4 Generation of Software Instrumentations for Simulation and Monitoring

The second alternative is to employ a distributed monitoring system where each ECU is attached with a monitoring unit that brings its own real-time clock. In this case, the real-time clocks of the distributed monitoring system need to be synchronized with a sufficiently small precision. When an event is being logged by the instrumentation, the time value it is associated with is determined by the reading of the real-time clock of the monitoring unit of the ECU. Figure E.32 in Appendix E depicts an example for such a monitoring approach.

A detailed description of clock synchronization mechanisms can be found in the thesis of Hofmann [43]. Hofmann describes a clock synchronization mechanism that enables safe temporal relations between instances of events such that a performance analysis of parallel and distributed real-time systems with meaningful and trustworthy statements can be performed.

For our purposes, we assume that an adequate simulation or monitoring-based approach is chosen such that the event trace that is obtained by the instrumentation of the potentially parallel and distributed system is temporally consistent. Our focus is on the analysis of event traces for the determination of application-specific timing properties and their adequate visualization.

Generic Event Logging Mechanism

We introduce a generic event logging mechanism which logs the occurrence of an event instance with the unique name of its corresponding event class and automatically associates this with the time value of a clock such that the time values of all recorded event instances are directly comparable, i.e. they refer to the same time base. The generic event logging mechanism provides a software function (or macro) that when being called at a place in the source code performs the event record construction and logging.

Signature of Logging Function	Description
<code>LogEventInstance(eventname)</code>	Log the instance of an event class with a globally valid time value

Table 6.1: Overview of generic event logging function

Simulation

chronSim [44] from INCHRON GmbH is a real-time simulation and analysis software for parallel and distributed embedded systems. The tool allows the simulation and analysis of the dynamic timing behavior of such systems based on a virtual prototype that is implemented in C-code is based on one of the several simulation kernels that the tool provides. It supports the simulation of OSEK-OS based embedded real-time systems as they are predominant in the automotive domain. Although AUTOSAR-OS is not directly supported, AUTOSAR systems can also be simulated based on the

6 Instrumentation of AUTOSAR Systems for Simulation and Monitoring

OSEK-OS kernel such that the dynamic timing behavior of the AUTOSAR system can be analyzed.

chronSim allows the logging of arbitrary stand-alone events and events that are part of a chain of events by means of placing specific markers into the source code of the real-time application. The markers are macros that make calls to an event logging function provided by a library of the chronSim real-time simulator. Each event marker is associated with a unique label to identify an event.

A specific concept is the logging of events that are part of a chain of events. For this, the chronSim event logging library provides a special macro (**EVENTCHAIN**) that requires to pass in a step number that denotes the position of the event in the event chain. The step number is static (i.e., it does not change at runtime) and can be determined from the flat concrete PathSpecification. The step numbers for all event and action classes of a PathSpecification can thus be provided as macros and stored in a central header file that is part of the instrumentation. As the markers of an event chain can be passed through multiple times during the dynamic runtime of the simulated real-time system, an instance number must additionally be provided. The instance number determines what instances of events are part of the instance of an event chain. It must be increased adequately in the source code in order to clearly distinguish multiple instances of event chains in the analysis.

Table 6.2 provides an overview on the event logging functions provided by chron-Sim.

Signature of Logging Function	Description
<code>EVENT(label)</code>	Logging of an event
<code>EVENT1(label, int val)</code>	Logging of an event and associated value
<code>EVENTCHAIN(label, int instance, int step)</code>	Logging of an event that is part of an event chain

Table 6.2: Overview of event logging functions provided by chronSim

Further details on the event logging functions and macros can be found in [39].

For the implementation of the Virtual Function Bus Trace hook functions for the simulation-based performance analysis with chronSim, the **EVENTCHAIN** macro is best to be used. This allows an automatic analysis of event chain instances by the chron-Sim tool. For the latter, it is required that **EVENTCHAIN** macro is used adequately in the code. For each event and action class in a concrete flat PathSpecification, the position is determined and used as step number in the **EVENTCHAIN** macro. If this is not the case, then the instances of event classes are not analyzed in the correct order. Furthermore, the event chain instance counter must be increased after each pass through an event chain (i.e., when an instance of an event class that is the last event class in a concrete flat PathSpecification of an AUTOSAR system has occurred). Otherwise, it is not clear how instances of event chains can be distinguished.

Monitoring

RTA-TRACE [26] from ETAS GmbH is a monitoring tool for OSEK-based ECU systems. The tool directly integrates with the real-time operating systems of the ERCOSEK [23] and RTA-OSEK [24] families from ETAS GmbH, but also allows the instrumentation of other OSEK-compliant RTOS from other vendors and even non-OSEK compliant schedulers. RTA-TRACE compliant RTOS are directly instrumented with logging functions to record the instances of interesting events. During runtime of the instrumented system, RTA-TRACE then records instances of the events with their time-stamps. Standard events that can be recorded refer to the life cycle of OS-tasks and interrupt service routines (ISRs), i.e., their activation, start, preemptions and termination. The tool also allows the recording of instances of user-defined trace-points and intervals that mark places or actions in the source code of an application. The latter is used for our purposes to perform RTE Tracing experiments.

RTA-TRACE allows the logging of arbitrary events by means of user-defined trace-points and intervals. Each trace-point or interval is associated with a unique identifier. With the help of the identifiers and the logging API provided by RTA-TRACE, instances of the events can then be logged with a time-stamp during the runtime of the ECU system. This is achieved by placing a call to a logging function with the identifier as parameter at the place in the application source code where the event occurs. The identifier of a trace-point, task-trace-point or interval must be unique and defined at pre-compile time. Furthermore, a trace category must be defined. This allows the grouping of multiple user-defined trace-points, task-trace-points or intervals to a specific category.

Table 6.3 provides an overview of the event logging functions provided by RTA-TRACE:

Signature of Logging Function	Description
<code>LogTracepoint(TracepointID, TraceCategory)</code>	Log the instance of a user-defined tracepoint (event) with ID <code>TracepointID</code>
<code>LogTaskTracepoint(TaskTracepointID, TraceCategory)</code>	Log the instance of a user-defined tracepoint (event) with ID <code>TaskTracepointID</code> in the context of an OS-task
<code>LogIntervalStart(IntervalID, TraceCategory)</code>	Log the start of an interval with ID <code>IntervalID</code>
<code>LogIntervalEnd(IntervalID, TraceCategory)</code>	Log the end of an interval with ID <code>IntervalID</code>

Table 6.3: Overview of event logging functions provided by RTA-TRACE

RTA-TRACE is currently restricted to single ECU systems. Monitoring experiments can thus only be performed for single ECU AUTOSAR systems. Further descriptions on the logging API can be found in [27].

For the implementation of the Virtual Function Bus Trace hook functions, the `LogTracepoint` logging function is best be used.

6.4.2 Implementation of Virtual Function Bus Trace Hook Functions

In order to capture the instances of AUTOSAR-specific event and action classes it is necessary to implement the corresponding trace hook functions provided by the Virtual Function Bus Tracing Mechanism with calls to an event logging software.

Table 6.4 provides an overview on how the event logging functions are used for the logging of instances of AUTOSAR-specific event and action classes. The event-name/label/identifier is constructed in such a way that it can be unambiguously identified to what event or action class the event instance belongs and in which context it occurs (i.e., the RunnableEntity that caused the event or action, the software component instance the data that is processed belongs to, etc.). Although this information is not required for the analysis and determination of timing properties later, it provides a human-readable information.

Event Logging Mechanism	Signature of logging function
Generic	<code>LogEventInstance(eventclassname)</code>
RTA-TRACE	<code>LogTracepoint(TracepointID, TraceCategory)</code>
chronSim	<code>EVENTCHAIN(label, int instance, int step)</code>

Table 6.4: Overview on event logging functions for simulation and monitoring

In the following sections, the implementation of the different types of Virtual Function Bus Trace hook functions that correspond to the AUTOSAR-specific event and action classes is described. The event logging functions are the markers for the event classes in the source code. Note that to keep the examples succinct, the event logging function of the generic event logging mechanism is used as a placeholder for the call of a concrete monitoring- or simulation-tool specific logging function.

RunnableEntity Events

The Virtual Function Bus Tracing Mechanism provides two hook functions for each RunnableEntity of an AtomicSoftwareComponentType. The hook functions are invoked just before the start of a RunnableEntity and immediately after the termination of a RunnableEntity. They correspond to the RunnableEntityStartEvent and RunnableEntityEndEvent defined in our Timing Model for AUTOSAR as described in section 5.2.1.

Listing 6.2 depicts an example OS-task body in which a RunnableEntity is executed. The invocation of the RunnableEntity RE1 (line 8) is guarded by calls to the Virtual Function Bus Trace hook functions (lines 5 and 11) that mark the start and termination of the RunnableEntity. The Virtual Function Bus Trace hook functions correspond to the respective RunnableEntityEvents.

To log the instance of a RunnableEntitiyEvent, the Virtual Function Bus Trace hook functions need to be implemented with function calls to a logging software.

```

1 TASK(Task10ms)
{
3 ...
5 //Call RunnableEntity start hook function
6 Rte_Runnable_SWC1_RE1_Start();
7 ...
9 //Invoke RunnableEntity
10 RE1();
11 ...
12 //Call RunnableEntity return hook function
13 Rte_Runnable_SWC1_RE1_Return();
14 ...
}

```

Listing 6.2: Implementation of an OS-task body with invocation of a RunnableEntity

Listing 6.3 shows an example implementation of the Virtual Function Bus Trace hook functions with log function calls for RTA-TRACE.

```

1 //RunnableEntity start hook function
2 void Rte_Runnable_SWC1_RE1_Start(){
3     LogEventInstance(Re_SWC1_inst_RE1_STP);
4 }
5 ...
6 //RunnableEntity return hook function
7 void Rte_Runnable_SWC1_RE1_Return(){
8     LogEventInstance(Re_SWC1_inst_RE1_RTP);
9 }

```

Listing 6.3: Logging of RunnableEntityEvents

For each RunnableEntityEvent, a unique event class identifier is defined. These are the identifiers `Re_SWC1_inst_RE1_STP`⁶ for the RunnableEntityStartEvent and `Re_SWC1_inst_RE1_RTP`⁷ for the RunnableEntityEndEvent.

Runtime Environment (RTE) API Actions

The Virtual Function Bus Tracing Mechanism provides two hook functions for each RTE API function (see section 4.3.7). The start hook function is called before a RunnableEntity makes a call to a specific RTE API function, and the return hook function is called after the RTE API call returns control to the RunnableEntity. The hook functions correspond to the RTEAPIActions of our Timing Model for AUTOSAR as described in section 5.2.2.

Listing 6.4 shows an example implementation of an RTE API function. The implementation was generated by an RTE-generator [25] in the RTE generation phase.

The start hook function is invoked just before the source code section where the read action is performed, and the return hook function is invoked just after that.

⁶STP = Start Trace Point

⁷RTP = Return Trace Point

6 Instrumentation of AUTOSAR Systems for Simulation and Monitoring

```
1 FUNC(Std_ReturnType, RTE_CODE) Rte_Read_SWC10_RPort1_a(CONSTP2VAR(SInt16, AUTOMATIC,
      RTE_APPL_DATA) data)
{
3   Std_ReturnType rtn;
5   //Call VFB Trace start hook function
7   Rte_ReadHook_SWC1_RPort1_a_Start(data);
9   //Perform RTE API call action
11  (*data) = Rte_RxBuf_0;
    rtn = ((Std_ReturnType)RTE_E_OK);
    //Call VFB Trace return hook function
    Rte_ReadHook_SWC1_RPort1_a_Return(data);
11  return rtn;
}
```

Listing 6.4: Implementation of an RTE API function

To log the instance of an RTEAPIAction, it is necessary implement both hook functions such that the start and end event instances are logged. Listing 6.5 shows an example implementation of the Virtual Function Bus Trace hook functions with log function calls.

```
//VFB Trace start hook function
2 void Rte_ReadHook_SWC1_RPort1_a_Start(CONSTP2VAR(SInt16, AUTOMATIC, RTE_APPL_DATA) data){
3   LogEventInstance(ReadHook_swci_rport_a_Start);
4 }
//VFB Trace return hook function
6 void Rte_ReadHook_SWC1_RPort1_a_Return(CONSTP2VAR(SInt16, AUTOMATIC, RTE_APPL_DATA) data){
7   LogEventInstance(ReadHook_swci_rport_a_Return);
8 }
```

Listing 6.5: Logging of RTEAPIActions

Note that in the case of a software component that can be instantiated multiple times, the component-specific instance handle that is used to distinguish the instances of the software component in the source code is given as an additional parameter to the Virtual Function Bus Trace hook function. In the implementation of the Virtual Function Bus Trace hook function, a distinction of cases must then be made based on the instance handle to log the RTEAPIAction of the correct software component instance.

Operating System (OS) Events

The Virtual Function Bus Tracing Mechanism defines several hook functions related to the life cycle of OS-tasks (see section 4.3.7).

In the case of implicit Sender/Receiver or Interrunnable communication, it is necessary to monitor the start and termination of OS-tasks⁸. They must be monitored

⁸The instants when the OS-task is activated, preempted or restarted are irrelevant as the copying of data between the RTE and the task-local buffers only occurs at the start and termination of an OS-task.

6.4 Generation of Software Instrumentations for Simulation and Monitoring

such that the copy processes between the task-buffers and the RTE taking place at right after the start and just before the termination of an OS-task can be adequately analyzed.

To monitor the start and termination of OS-tasks, the OS-task dispatch hook `Rte_Task_Dispatch(...)` provided by the Virtual Function Bus Tracing Mechanism is used. It corresponds to the `TaskDispatchEvent` defined in our Timing Model for AUTOSAR. The Virtual Function Bus Tracing Mechanism does not provide a hook function that is invoked just before the termination of an OS-task. As we require to monitor the termination of an OS-task, we introduce the OS-task terminate hook function `Rte_Task_Terminate(...)` which is analogous to the `Rte_Task_Dispatch(...)` hook function. It is called just before the termination of an OS-task and corresponds to the `TaskTerminateEvent` defined in our Timing Model for AUTOSAR.

Listing 6.6 depicts an example implementation of an OS-task body as generated by an RTE-generator [25] for an example AUTOSAR ECU system. The invocation of the OS-task terminate hook function `Rte_Task_Terminate(Task10ms);` has been inserted manually as it is not part of the AUTOSAR standard yet and is thus not being generated automatically by current AUTOSAR-compliant RTE generators.

```
2 TASK(Task10ms)
3 {
4     //Call Task Dispatch hook function
5     Rte_Task_Dispatch(Task10ms);
6     ...
7     //Execute RunnableEntities
8     RE1();
9     ...
10    //Call Task Terminate hook function
11    Rte_Task_Terminate(Task10ms);
12 }
```

Listing 6.6: Implementation of an OS-task body

To log the instances of a `TaskDispatchEvent` and a `TaskTerminateEvent`, the Virtual Function Bus Trace hook functions need to be implemented with function calls to a logging software. Listing 6.7 shows the implementation of the VFB Trace hook functions with such log function calls.

Communication (COM) Events

As described in section 4.3.7, the Virtual Function Bus Tracing Mechanism provides hook functions that allow the monitoring of the transmission and reception of system signals and system signal groups via the services provided by the COM module. These hook functions directly correspond to the `COMSignalEvents` and `COMSignalGroupEvents` of our Timing Model for AUTOSAR as described in section 5.2.3. In the following, the implementation of the Virtual Function Bus Trace hook functions for `COMSignalEvents` is explained. Although the transmission and

6 Instrumentation of AUTOSAR Systems for Simulation and Monitoring

```
1 //Task Dispatch hook function
2 void Rte_Task_Dispatch(TaskType t){
3     if (t == Task10ms) {
4         LogEventInstance(Task10ms_Start_TP);
5     }
6 }
7 //Task Terminate hook function
8 void Rte_Task_Terminate(TaskType t){
9     if (t == Task10ms) {
10        LogEventInstance(Task10ms_Terminate_TP);
11    }
12 }
```

Listing 6.7: Logging of TaskEvents

reception of signal groups are implemented differently in the source code, the implementation of the corresponding Virtual Function Bus Trace hook functions for COMSignalGroupEvents is in principle analogous.

Signal Transmission

Listing 6.8 shows an example implementation of an RTE API function.

```
1 FUNC(Std_ReturnType, RTE_CODE)
2 Rte_Write_SWC1_PPort_a(VAR(SInt16, AUTOMATIC) data)
{
3     Std_ReturnType rtn = RTE_E_OK;
4     Rte_WriteHook_SWC1_PPort_a_Start(data);
5     //Call COM Transmit Signal hook function
6     Rte_ComHook_ComSignal_a_SigTx(&data);
7     if ( ((StatusType)E_OK) != Com_SendSignal(((Com_SignalIdType)1), &data) )
8     {
9         rtn = ((Std_ReturnType)RTE_E_COMM_ERROR);
10    }
11    Rte_WriteHook_SWC1_PPort_a_Return(data);
12    return rtn;
13 }
```

Listing 6.8: Implementation of an RTE API function including sending a system signal via the communication service

The RTE API function realizes an explicit write action to the DataElementPrototype **a** in the providing port **PPort** of a software component **SWC1** on the sender ECU side. The implementation of the RTE API function has been generated by an RTE generator [25]. It includes the invocation of the **Com_SendSignal(...)** function provided by the COM module. Through this, a COM signal transmission is initiated. The COM hook function for signal transmission **Rte_ComHook_ComSignal_a_SigTx** provided by the Virtual Function Bus Tracing Mechanism is invoked directly before the invocation of **Com_SendSignal(...)**.

To log the instance of a COMSignalEvent, the corresponding Virtual Function Bus Trace hook function needs to be implemented with a function call to a logging

6.4 Generation of Software Instrumentations for Simulation and Monitoring

software. Listing 6.9 depicts the implementation of the Virtual Function Bus Trace hook for the COMSendSignalEvent.

```
1 //COM Transmit Signal hook function
2 void Rte_ComHook_ComSignal_a_SigTx(SInt16* data){
3     LogEventInstance(ComSignal_a_SigTx_TP);
4 }
```

Listing 6.9: Logging of a COMSendSignalEvent

Signal Reception

Listing 6.10 shows an example implementation of a COM callback function that realizes the reception of a system signal **a** on the receiver ECU side.

```
1 COMCallback(Rte_COMCbk_ComSignal_a)
2 {
3     StatusType comstatus;
4     UInt16 data;
5     boolean read_ok = TRUE;
6     comstatus = Com_ReceiveSignal(((Com_SignalIdType)1), &data);
7     if ( ((StatusType)E_OK) != comstatus ) {
8         read_ok = ((boolean)FALSE);
9     } else {
10        //Call COM Receive Signal hook function
11        Rte_ComHook_ComSignal_a_SigRx(&data);
12    }
13    if ( ((boolean)TRUE) == read_ok ) {
14        RTE_ATOMIC16(Rte_RxBuf_1 = data);
15    }
16 }
```

Listing 6.10: Implementation of a COM callback function

The COM callback function includes the invocation of the **Com_ReceiveSignal(...)** function provided by the COM module, i.e., a signal reception is initiated. The COM hook function for signal reception **RTE_ComHook_ComSignal_a_SigRx** is provided by the Virtual Function Bus Tracing Mechanism and is invoked directly after the invocation of **Com_ReceiveSignal()** upon a successful signal reception.

To log the instance of a COMReceiveSignalEvent, the corresponding Virtual Function Bus Trace hook function needs to be implemented with a function call to a logging software. Listing 6.11 depicts the implementation of the Virtual Function Bus Trace hook for the COMReceiveSignalEvent.

```
1 //Com Receive Signal hook function
2 void Rte_ComHook_ComSignal_a_SigRx(SInt16* data){
3     LogEventInstance(ComSignal_a_SigRx_TP);
4 }
```

Listing 6.11: Logging of a COMReceiveSignalEvent

6.4.3 Integration into AUTOSAR Development Methodology and ECU Software Build Process

As the instrumentation for simulation and monitoring-based performance analysis approaches employs the Trace hook functions provided by the Virtual Function Bus Tracing Mechanism, the necessary source code files that make up the instrumentation of the RTE can be seamlessly included into the ECU software build process of an AUTOSAR ECU. Figure 6.16 shows the overview of the second part of the AUTOSAR methodology (compare to section 4.3.3) where the logical PathSpecifications that complement an AUTOSAR system description are used to generate the required instrumentation for monitoring or simulation-based approaches.

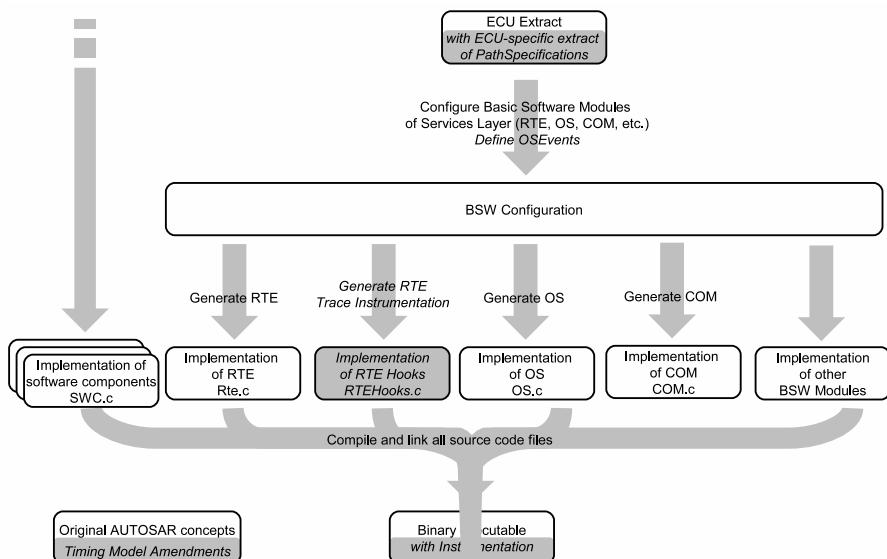


Figure 6.16: Overview of AUTOSAR methodology with usage of timing model concepts for automatic generation of an RTE tracing instrumentation

The implementation of the Virtual Function Bus Trace hook functions with the invocation of the log functions can be generated as described in this chapter. This results in additional source code files that complement the RTE of an AUTOSAR-compliant ECU in such a way that no other source code files, neither of the application software nor of the RTE or any other basic software module, do need to be modified. The additional source code files can then be integrated into the ECU software build process to obtain the final instrumented binary that can be used for simulation- or monitoring-based performance evaluation (see figure 6.16).

6.4 Generation of Software Instrumentations for Simulation and Monitoring

In the future, an extended RTE generator tool could automatically provide the necessary instrumentation in the form of implementations of the Virtual Function Bus Trace hook functions based on the logical PathSpecifications defined for an AUTOSAR system. This would allow for a direct integration of performance analysis by means of simulation- and monitoring-based approaches into the development of AUTOSAR-compliant ECU and E/E systems.

7 Analysis of Event Traces from Monitoring and Simulation and Determination of Application-Specific Timing Properties

7.1 Introduction

For the precise description of application-specific timing requirements, a concept has been introduced in chapter 5 that allows an exact specification of signal paths in an AUTOSAR system. Signal paths are described on the basis of events and actions that can be observed during the runtime of the system and that lie on the signal path. Due to the fact that the signal path specifications are based on observable events and actions, such signal path specifications can be used to derive and generate an instrumentation such that instances of these events and actions can be recorded in simulation and monitoring-based approaches during the runtime of the system. During the procedure of simulation- or monitoring experiments, instances of events and actions are then automatically recorded by the instrumentation into an event trace.

In order to make statements about the degree of fulfillment of the application-specific timing requirements, their corresponding timing properties need to be determined. As the timing requirements refer to specifications of signal paths, it is required to first determine instances of signal paths based on which meaningful timing properties can then be determined. The determination of instances of signal paths is, however, not trivial.

For the determination of path instances it is required to identify those instances of events and actions that stand in a temporal and plausible effective relation, and for which the causal relations among their event and action classes is described by the respective path specification. In other words: What are the instances of events and actions between which there is a true causal and temporal relation? These instances of events and actions then form a path instance. Through the transitivity property of the causal and temporal relations on events (see foundations of timing models, section 4.2) it follows what event instance at the start of a signal path instance is causally related to what event instance at the end of a signal path.

There are different approaches for the determination of path instances in simulation- and monitoring-based performance analysis approaches. These principally differ in the type of information for instances of events and actions that is available for the determination of path instances. In simulation-based approaches, and in contrast to monitoring-based approaches, during the logging of an event or action instance,

additional information can be recorded at no extra cost¹ that could also influence the dynamic behavior of the system and consequently its timing characteristics. Such additional information can help to analyze event traces for the determination of effective path instances. For example, the real-time simulation tool chronSim [44] from INCHRON GmbH employs a so-called instance number² to directly distinguish the instances of events belonging to different event chain instances during the logging of the respective events. A problem with this approach, however, is that when the event chain instance number is increased at an incorrect place the analysis software determined invalid path instances. While correct programming is well possible, it poses a challenge especially in multi-rate control applications. For the determination of timing properties based on event traces that are recorded by means of monitoring-based approaches it is often not possible to obtain such information due to the overheads that it entails during monitoring. Consequently, a different approach is required that allows the determination of path instances without such additional information. As no such approach that could be directly adapted is available, a new approach is introduced in this thesis. The tradeoff of the new approach compared to the existing approach with additional information is, however, an increased computational complexity and consequently an increased computational effort. This, however, is not addressed in this thesis.

Based on the effective path instances, meaningful timing properties such as end-to-end path delays, interval delays and synchronization intervals can then be determined.

For the visualization of results and the comparison of timing properties with the timing requirements, two types of diagrams are principally distinguished: statistical diagrams and run-time oriented diagrams (see Klar et al. [48]). Statistical diagrams present the results of a summary analysis. Examples for statistical diagrams are the well-known histograms. They provide an overview on the statistical distribution of timing properties and to what degree the timing requirements are satisfied and if there are any anomalies (e.g., outliers, shift to an extremum, accumulations). Runtime-oriented diagrams show the results from a simulation- or monitoring-based event trace on a time axis³. Examples for runtime-oriented diagrams are Gantt diagrams. Runtime-oriented diagrams provide a view on the timing behavior of a system and its timing properties for each point in time and or interval for which data is available in the recorded event trace.

In this thesis, we employ a special kind of Gantt diagrams that is tailored to our needs to visualize the instances of events and actions on a time axis and to visualize relations between these through directed edges. Gantt diagrams serve for the purpose of visualizing path instances and to show how timing properties are determined through measuring temporal distances between specific event instances that are of interest. Furthermore, a new form of runtime-oriented diagram is introduced, the

¹e.g., delays through processing additional instructions, overhead through additional memory consumption

²More precisely, this should be termed event chain instance number.

³Runtime-oriented diagrams are also referred to as time-axis diagrams.

7.2 Measurement Of Temporal Distances Between Events And Actions

so-called Timing Oscilloscope Diagram, to adequately visualize the development of timing properties over time such that it is possible to evaluate at each point in time if and to what degree the imposed timing requirements are satisfied.

Figure 7.1 depicts an overview on the approach for the determination of path instances and timing properties introduced in this chapter.

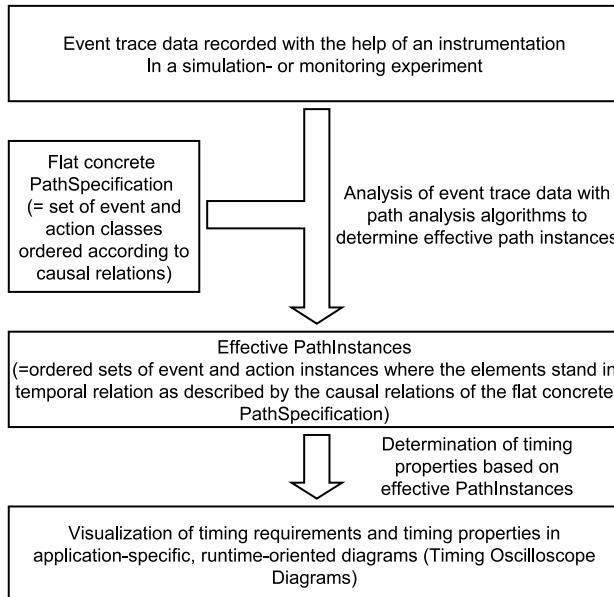


Figure 7.1: Overview on determination of path instances for determination of timing properties and their visualization

7.2 Measurement Of Temporal Distances Between Events And Actions

In order to measure the time between any two instances of event or action classes we introduce the notion of an *inner distance* and an *outer distance* between them. This distinction is mainly required due to the fact that we distinguish between events and actions.

Figures 7.2(a) and 7.2(b) depict an event class A and an action class B. Several instances of both A and B are plotted in the event trace scenario. In each figure, four cases are distinguished:

1. The distance between two instances of event class A.

7 Analysis of Event Traces from Monitoring and Simulation

2. The distance between two instances of action class of B.
3. The distance between an instance of event class A and an instance of action class B.
4. The distance between an instance of action class B and an instance of event class A.

Figure 7.2(a) depicts how the outer distance between two considered instances is measured. Analogously, figure 7.2(b) depicts how the inner distance is measured.

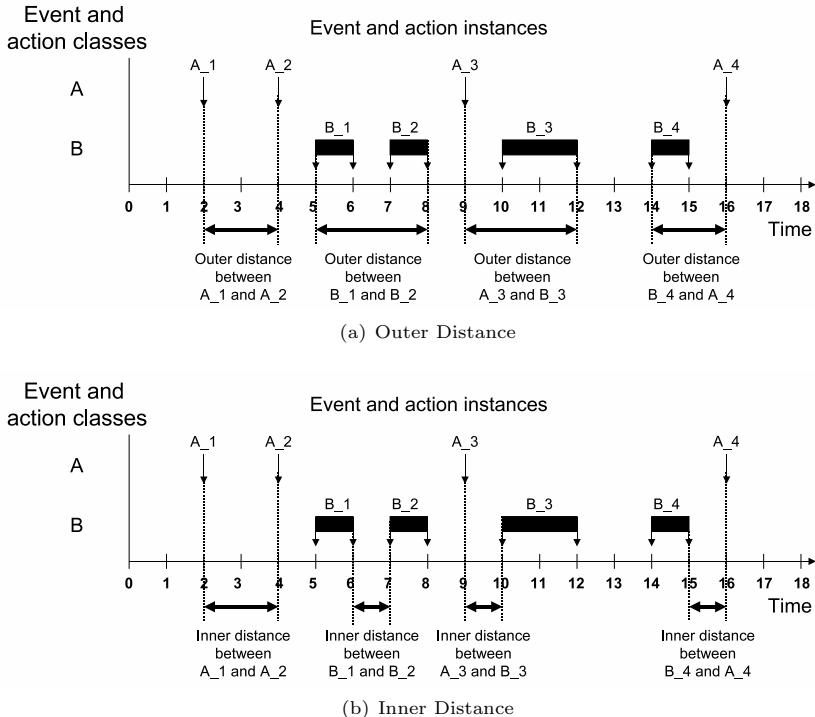


Figure 7.2: Inner and outer distance between instances of event and action classes

As the instances of event classes are only associated with a single time value (instant) they only have a single handle to start or end temporal measurement.

Instances of action classes are associated with a start time value (`startinstant`) and an end time value (`endinstant`). Depending on whether such an action instance denotes the start of measurement or the end of measurement, and depending on

7.2 Measurement Of Temporal Distances Between Events And Actions

whether the inner distance or the outer distance is to be measured, either the start time value or the end time value serves as handle for measurement.

If the two instances for which the latency is to be measured both belong to an event class then the inner distance equals to the outer distance.

Listing 7.1 shows an algorithm to compute the inner distance between two instances of event or action classes. In the algorithm, the above mentioned four cases do need to be distinguished explicitly.

```
1 def innerDistance(a, b)
2     innerDistance := null
3     if (a.type ≡ EventClass ∧ b.type ≡ EventClass)
4         innerDistance := b.instant – a.instant
5     elseif (a.type ≡ ActionClass ∧ b.type ≡ ActionClass)
6         innerDistance := b.startinstant – a.endinstant
7     elseif (a.type ≡ EventClass ∧ b.type ≡ ActionClass)
8         innerDistance := b.instant – a.endinstant
9     elseif (a.type ≡ ActionClass ∧ b.type ≡ EventClass)
10        innerDistance := b.startinstant – a.instant
11    end
12    return innerDistance
13 end
```

Listing 7.1: Algorithm to compute the inner distance between any two instances of event or action classes

Listing 7.2 shows an algorithm to compute the outer distance between two event or action classes.

```
1 def outerDistance(a, b)
2     outerDistance := null
3     if (a.type ≡ EventClass ∧ b.type ≡ EventClass)
4         outerDistance := b.instant – a.instant
5     elseif (a.type ≡ ActionClass ∧ b.type ≡ ActionClass)
6         outerDistance := b.endinstant – a.startinstant
7     elseif (a.type ≡ EventClass ∧ b.type ≡ ActionClass)
8         outerDistance := b.instant – a.startinstant
9     elseif (a.type ≡ ActionClass ∧ b.type ≡ EventClass)
10        outerDistance := b.endinstant – a.instant
11    end
12    return outerDistance
13 end
```

Listing 7.2: Algorithm to compute the outer distance between any two instances of event or action classes

By means of the inner distance, temporal precedence can be evaluated between two instances of event or action classes whereas by means of the outer distance, the latency between two instances of event or action classes can be determined.

Example: Figure 7.3 depicts an example with two action classes `WriteAction` and `ReadAction` along with two of their instances.

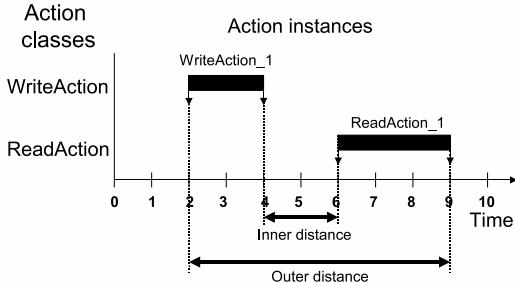


Figure 7.3: Example for two action classes and the inner and outer distance between them

The inner distance between `WriteAction_1` and `ReadAction_1` allows the determination of the temporal order in which both instances occurred. In the scenario, `WriteAction_1` occurred before `ReadAction_1` because the inner distance is 2 time units and thus greater than zero. The outer distance between `WriteAction_1` and `ReadAction_1` allows the measurement of the latency between the start of `WriteAction_1` and the end of `ReadAction_1`. In the example, the latency is 7 time units. \square

Instances of event or action classes can in principle also coincide or overlap. These special cases need to be considered carefully as these are important for correct causal and temporal reasoning and consequently for the determination of valid timing properties. Due to the fact that a causal order on two instances of events implies the same temporal order on the same instances of events (see foundations of timing models, section 4.2.1), it is not possible that two causally related instances of events happen at the same time. Event or action instances that are causally related cannot coincide or overlap. This is a contradiction to Newtonian physics. If two instances of events are indeed causally related, and the measured temporal distance is zero, then the time base has a too coarse grain granularity. This is a problem of the simulation or monitoring system and the employed clocks. The granularity of the clocks from which the time values of the event and action instances originate from must be at least that large such that the inner distance between two causally related instances of events is greater than zero.

7.3 Determination of Path Instances and Timing Properties

Hierarchical event chains have been introduced as a means to specify an order on a set event and action classes that mark relevant places and actions in a system. While hierarchical event chains are required for the specification of a logical signal path in component-based development approaches, their concrete flattened representation is required for the determination of instances of signal paths. In the following, we assume that a `PathSpecification` is given in terms of a flat ordered set of event and

action classes. A flat ordered set of event and action classes can be obtained from a hierarchical event chain as described in chapter 6.

While it is possible to specify in which order event and action classes must occur through a PathSpecification such that a specific cause-and-effect relation can be observed, the determination of event and action instances that satisfy this condition is not trivial. Such sets of event and action instances are termed *PathInstances*. To define this term, we first define the term *EventChainInstance* and based on that, the term *PathInstance*.

Definition 27 An *EventChainInstance* is an ordered set of event and action instances where the single event and action instances stand in a temporal precedence order relation as specified by the *EventChainType* to which the *EventChainInstance* refers.

Definition 28 A *PathInstance* is a *EventChainInstance* that refers to a *PathSpecification* as its *EventChainType*.

Knowing the PathInstances and thus the event and action instances that stand in relation as specified by a PathSpecification is necessary in order to determine timing properties of chains of cause-and-effect and signal paths described by the PathSpecification. The following example motivates the need for dedicated algorithms to determine the PathInstances.

Example: Figure 7.4 depicts an event trace scenario in which three event classes A, B and C are considered together with their instances.

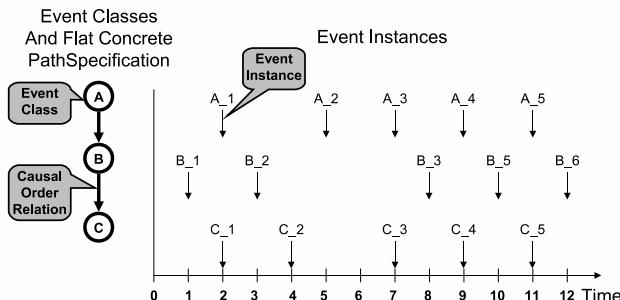


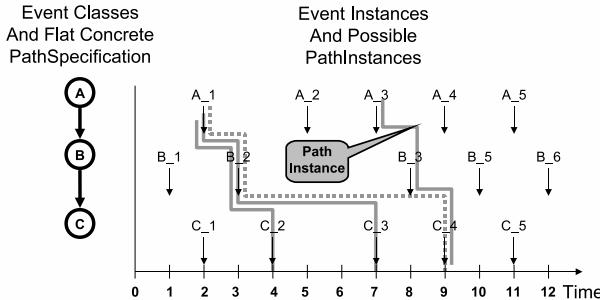
Figure 7.4: Example: Event trace scenario with three event classes and their instances and a given PathSpecification

It is assumed that a PathSpecification $A \rightarrow B \rightarrow C$ is given. The question is: Which event instances stand in a temporal relation as specified by the flat concrete PathSpecification and which relations are causally effective?

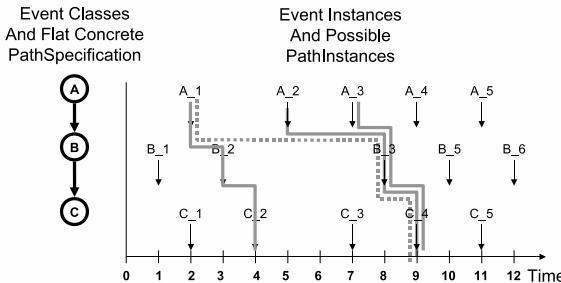
It is obvious that this question cannot be answered easily. In a pen and paper approach, one would draw lines from event instances A_i over B_j to C_k . This,

7 Analysis of Event Traces from Monitoring and Simulation

however, quickly gets confusing and calls for a divide and conquer approach. In a divide and conquer approach, one would start to identify pairs of event instances where the event classes stand in a direct relation to each other (e.g., A and B, or B and C), where the first event instance temporally comes before the second event instance (e.g., A₁ → B₂ or B₃ → C₄), and one would draw a line between these. The next step would then be to connect lines such that a continuous line is established from an instance of event class A over one of B to one of C. When this is done for all possible combinations, the result is a set of *possible* PathInstances. Figures 7.5(a) and 7.5(b) depict the same scenario where several possible PathInstances are shown.



(a) PathInstances from event instance A₁ to event instance C



(b) PathInstances from event instance of A to event instance C₄

Figure 7.5: Example: Event trace scenario with three event classes and their instances, a given PathSpecification and several feasible PathInstances

From the figures it gets obvious that some of the possible PathInstances are not *feasible*, meaning that from a causal perspective there are conflicts with other PathInstances. This makes it difficult to determine meaningful timing properties.

- In the upper figure, the solid lines correspond to PathInstances that are possible and feasible. The dotted line corresponds to a PathInstance that is

possible, but not feasible as there is a causal conflict. The PathInstance $A_3 \rightarrow B_3 \rightarrow C_4$ ends in C_4 and where A_3 comes after A_1 . The cause for C_4 is thus A_3 and not A_1 .

- In the lower figure, the solid lines also correspond to PathInstances that are possible and feasible. The dotted line corresponds to a PathInstance that is possible, but not feasible.

As there are potentially multiple feasible PathInstances with the same first event instance (see figure 7.5(a)) or the same last event instance (see figure 7.5(b)), these PathInstances need to be interpreted adequately. Figure 7.6 depicts the same scenario where the identified feasible PathInstances are shown.

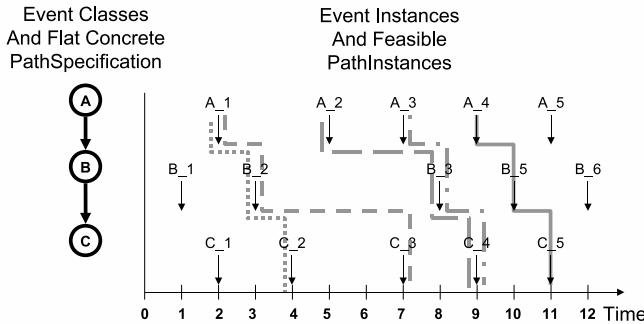


Figure 7.6: Example: Same event trace scenario of three event classes, their instances and identified relations between them

- The dotted line corresponds to the PathInstance $A_1 \rightarrow B_2 \rightarrow C_2$ whereas the dashed line corresponds to the PathInstance $A_1 \rightarrow B_2 \rightarrow C_3$. Both PathInstances are feasible. From the perspective of A_1 , i.e., looking *forwards* on the time axis, there are two effects C_2 and C_3 that are caused by A_1 . C_3 comes after C_2 whereby both have a relation to B_2 . C_3 , however, does not bring a new effect compared to C_2 with respect to A_1 . The PathInstance $A_1 \rightarrow B_2 \rightarrow C_3$ is not *effective*. Due to that, we are only interested in the *shortest feasible forward* PathInstance as this is effective.
- The long-dashed line corresponds to the PathInstance $A_2 \rightarrow B_3 \rightarrow C_4$ whereas the dashed-dotted line corresponds to the PathInstance $A_3 \rightarrow B_3 \rightarrow C_4$. Both are feasible PathInstances. From the perspective of C_4 , i.e., looking *backwards* on the time axis, there are two causes, A_2 and A_3 . As A_3 comes after A_2 , A_3 invalidates A_2 . A_2 thus does not have an effect on C_4 , and $A_2 \rightarrow B_3 \rightarrow C_4$ is not effective. We are thus only interested in the *shortest feasible backwards* PathInstance as this is effective.

- The grey-solid line corresponds to the PathInstance $A_4 \rightarrow B_5 \rightarrow C_5$. The PathInstance is feasible, and there are no conflicts of this PathInstance with other PathInstances. A_4 can clearly be identified as being the cause for C_5 whereas for C_5 it can be stated that it is clearly affected by A_4 . In the considered event trace scenario, this PathInstance is thus also an effective PathInstance.

□

For a given PathSpecification, our goal is to identify PathInstances that are feasible and that do not overlap with other PathInstances that have the same first or last event instance. For these PathInstances we then can determine meaningful timing properties as there is only one event instance denoting the cause and one event instance denoting the effect. Furthermore, these PathInstances can be analyzed together with PathInstances determined for other PathSpecifications to identify the synchrony between event instances.

In the following, we introduce a number of algorithms which operate on ordered sets of event classes (PathSpecifications) and their instances. The algorithms allow to determine the PathInstances that correspond to the PathSpecification and that we are interested in.

At first, we introduce an algorithm which allows the computation of all *possible PathInstances* between the instances of two event classes (section 7.3.1). The result is a set of PathInstances which are in principle valid from the perspective of temporal precedence between the contained event instances, however, some of the PathInstances might not be feasible due to occurrences of later or earlier instances of event classes which invalidate certain PathInstances due to causal conflicts.

We then present two algorithms which extract the *feasible PathInstances* from a previously computed set of all possible PathInstances (section 7.3.2). We distinguish between feasible forward PathInstances and feasible backward PathInstances, thus two algorithms are introduced here.

In section 7.3.3, the transitivity property of the causal and temporal order relation on sets of events is exploited. An algorithm is introduced which allows to *join* two sets of PathInstances into a single set of PathInstances. Together with the previously introduced algorithms to extract all feasible PathInstances from a set of all PathInstances, the algorithm for joining PathInstances is combined to a single algorithm to directly compute all *feasible forward or backward PathInstances* from an ordered set of event or action classes (a PathSpecification) with two or more event classes (section 7.3.4).

We then introduce two algorithms to extract the *shortest* PathInstances from previously computed sets of feasible forward or backward PathInstances (section 7.3.5). This results in sets of PathInstances that we are actually interested in. For each of the PathInstances, the latency between the first event instance (i.e., the cause) and last event instance (i.e., the effect) can be determined as a meaningful timing property. Furthermore, we define the notion of *effective PathInstances*.

In section 7.3.6, we show how flat concrete PathSpecifications can be intersected to determine so-called *intersection points* between two distinct, but overlapping flat concrete PathSpecifications. Such intersection points can be either *join points* or *fork points*. Similarly, PathInstances of distinct PathSpecifications can also be intersected in order to determine *intersection point instances*. These are then either *fork point instances* or *join point instances*. Algorithms to compute the latter are introduced in section 7.3.7.

We furthermore introduce algorithms to determine subsets of intersected PathInstances, making up *join path segments* or *fork path segments* (section 7.3.8). Based on these, we can then determine the size of the synchronization between two or more event instances with respect to a common event instance (join or fork point instance) that is on the same PathInstance.

The algorithms presented in the following have been implemented in a script-based tool framework. They are shown in a pseudo-code notation that is directly derived from the implementation of the algorithms. The example figures shown in the following are automatically generated with the help of the proof-of-concept implementation.

Note that in most of the presented examples, events are shown rather than actions. The algorithms equally apply to actions, however, the generated figures are then more difficult to read as they require more space.

7.3.1 All Possible Path Instances between the Instances of two Event or Action Classes

As a starting point we introduce an algorithm which takes two event classes and computes the set of all possible PathInstances between any of the event instances. Listing 7.3 shows the algorithm in a pseudo-code notation.

```

1 def computeAllPossiblePathInstances(a, b)
2   allPossiblePathInstances := [];
3   a.instances.each {
4     |a_i|
5     b.instances.each {
6       |b_j|
7       if innerDistance(a_i, b_j) > 0
8         allPossiblePathInstances ≪ [a_i, b_j];
9       end
10      }
11    }
12  return allPossiblePathInstances
13 end

```

Listing 7.3: Algorithm to compute all possible PathInstances between the instances of two event classes

A PathInstance between two event instances $a.i$ and $b.j$ exists if the end instant of $a.i$ lies before the start instant of $b.j$, or if the inner distance between both event instances is greater than zero (temporal precedence). The result is a set of tuples of event instances, each denoting a possible PathInstance between two event

7 Analysis of Event Traces from Monitoring and Simulation

instances. Note that event classes are treated in the same way as action classes. This is encapsulated in the algorithm that computes the inner distance between any combination of two event or action classes.

Example: Figures 7.7 and 7.8 depict an example where the algorithm is applied to two event classes A and B. In figure 7.7, only the instances of A and B are shown.

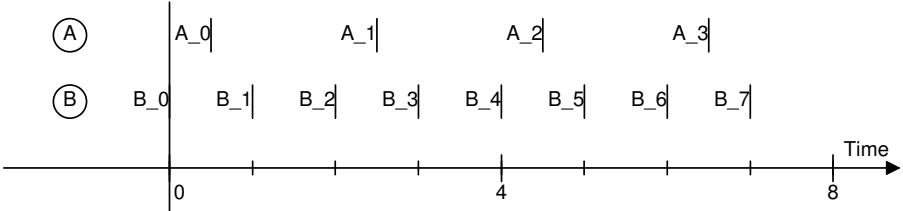


Figure 7.7: Event trace scenario with event classes A and B and their instances

In figure 7.8, all possible PathInstances which have been computed by the algorithm shown in Listing 7.3 are additionally shown. Not all PathInstances can be seen in the upper part of the figure as multiple PathInstances overlap. The individual PathInstances are additionally shown in the lower part of the figure as intervals along with their outer distance (i.e., the latency between the first and the last event instance of the PathInstance).

Clearly, not all PathInstances are feasible, meaning that their first and last event instances have a meaningful cause and effect relationship. For instance, the PathInstance $A_0 \rightarrow B_3$ is not feasible as it collides with PathInstance $A_1 \rightarrow B_3$. More concrete, both PathInstances end in B_3 , but there can only be one instance from event class A responsible for this, i.e. being interpreted as the direct cause for B_3 or having an effect on B_3 . From a temporal perspective, A_1 happens after A_0 and thus is the last instance of event class A before B_3 . Thus, A_1 , can be considered as being the direct cause for B_3 . This means that that $A_1 \rightarrow B_3$ is a feasible PathInstance. \square

In the following, we introduce two algorithms to extract feasible PathInstances from the set of all possible PathInstances.

7.3.2 Feasible Path Instances

The determination of feasible PathInstances depends on the point of view or direction from which the PathInstances are considered. In principle, there are two points of views: either from the first event instance to the last event instance (*forwards*), or from the last event instance to the first event instance (*backwards*). In the following, we introduce an algorithm to determine the set of feasible forward PathInstances from a set of all possible PathInstances. The algorithm to determine the set of feasible backward PathInstances is analogous and is shown in Appendix E.

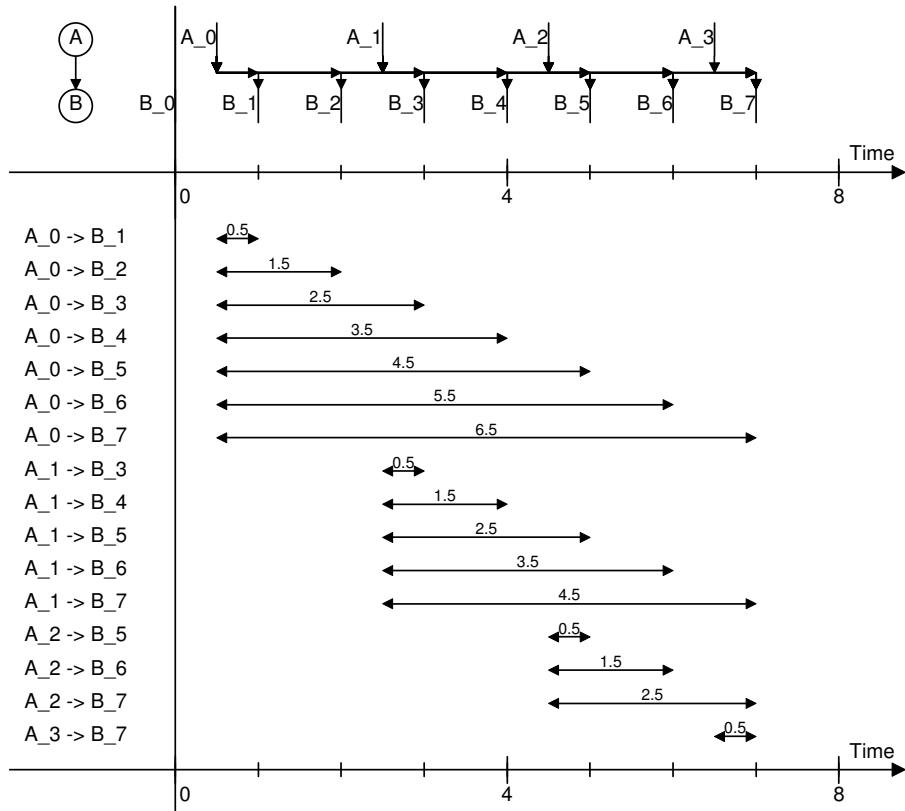


Figure 7.8: Event trace scenario with computed set of all possible PathInstances between instances of event classes A and B

Listing 7.4 shows an algorithm to extract all feasible forward PathInstances from a given set of all possible PathInstances.

```

1 def extractFeasibleForwardPathInstances(possiblePathInstances)
2   feasibleForwardPathInstances := []
3   mapOfPossiblePathInstances := Hash.new
4   possiblePathInstances.collect{
5     |possiblePathInstance|
6     mapOfPossiblePathInstances[possiblePathInstance.last] := []
7   }
8   possiblePathInstances.collect{
9     |possiblePathInstance|
10    mapOfPossiblePathInstances[possiblePathInstance.last] << possiblePathInstance
11  }
12  mapOfPossiblePathInstances.keys.each {
13    |ei2|

```

7 Analysis of Event Traces from Monitoring and Simulation

```

possiblePathInstanceMin := mapOfPossiblePathInstances[ei2][0]
15
mapOfPossiblePathInstances[ei2].each {
17    |possiblePathInstance|
    if innerDistance(possiblePathInstance.first, ei2) < innerDistance(possiblePathInstanceMin.first, ei2)
19        possiblePathInstanceMin := possiblePathInstance
        end
21    }
23 } feasibleForwardPathInstances << possiblePathInstanceMin
25 return feasibleForwardPathInstances.sort
end

```

Listing 7.4: Algorithm to extract all feasible forward PathInstances from a set of possible PathInstances

In a first step, a hash map is constructed where the keys correspond to the last event instances of the given possible PathInstances and the values correspond to the PathInstances that have the key event instance as their last event instance. In a second step, the hash map is traversed in two phases where for each key (i.e., an event instance which is the end of a possible PathInstance) the PathInstance with the minimal inner distance is determined. The PathInstance with the minimal inner distance is then the feasible forward PathInstance as it is based on the last possible, previous event instance.

Example: Figure 7.9 shows the feasible forward PathInstances which have been extracted from the set of all possible PathInstances for the previous introduced example.

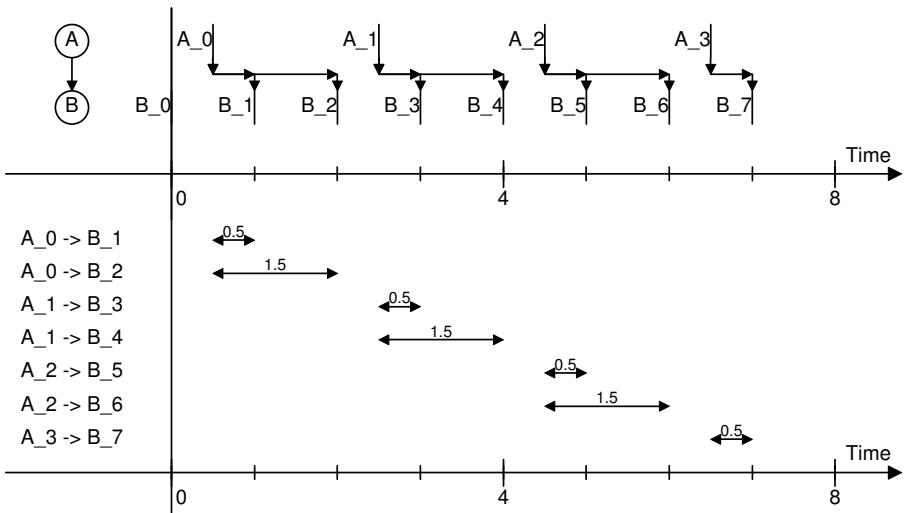


Figure 7.9: Event trace scenario with computed set of all feasible forward PathInstances between instances of event classes A and B

For each event instance of B , an event instance of A has been determined as its feasible cause. \square

The presented algorithm assumes that the PathInstance with the minimum inner distance is the feasible PathInstance. This assumption is valid as in general, a previous event instance is overwritten by its directly succeeding event instance. The algorithm in Listing 7.4 can easily be modified such that the feasible PathInstances with the maximum outer distance is extracted. For these PathInstances, however, we do not have a meaningful interpretation and thus are not interested in them.

So far, we have introduced algorithms to determine feasible forward PathInstances and feasible backward PathInstances between instances of two event classes (i.e., assuming PathInstances and PathSpecifications of length two). The algorithms in this section, however, are independent of the size of the paths. In the following, we will consider PathSpecifications and PathInstances of arbitrary length where the algorithms can also be applied.

7.3.3 Joining Path Instances

In principle, a PathSpecification of length n can be split into $n-1$ PathSpecifications of length 2. For these, it is possible to first compute all possible PathInstances between the instances of their two event classes and then to extract the feasible forward or backward PathInstances. In order to determine PathInstances of length n that correspond to the PathSpecification of length n , the determined feasible PathInstances of length 2 need to be joined. Note that this needs to be performed in the correct order.

Listing 7.5 shows the algorithm to join two sets of PathInstances.

```

1 def joinPathInstances(pathInstances1, pathInstances2)
2     joinedPathInstances := []
3     pathInstances1.each{
4         |pathInstance1|
5         pathInstances2.each{
6             |pathInstance2|
7             if pathInstance1.last ≡ pathInstance2.first
8                 joinedPathInstance := pathInstance1[0..pathInstance1.size-2].concat(pathInstance2)
9
10            if joinedPathInstance ≠ []
11                joinedPathInstances ≪ joinedPathInstance
12            end
13        end
14    }
15 }
16 return joinedPathInstances.uniq
17 end

```

Listing 7.5: Algorithm to join two sets of PathInstances

The algorithm takes two sets of PathInstances and computes a set of joined PathInstances. For this, the two sets of PathInstances are traversed, and for each pair of PathInstances it is checked if they can be joined. Two PathInstances $pathInstance1$ and $pathInstance2$ can be joined if the last event instance of $pathInstance1$ is the same

7 Analysis of Event Traces from Monitoring and Simulation

as the first event instance of `pathInstance2`. This is the exploitation of the transitivity property of the temporal and causal order relations. The joined PathInstance is then the concatenation of both `pathInstance1` and `pathInstance2` where one of the common event instance (in our case the last event instance of `pathInstance1`) is removed in order to avoid duplicates in the joined PathInstance.

Example: Figure 7.10 shows an example where the feasible forward PathInstances between the instances of event classes A and B, and B and C, respectively, have been computed.

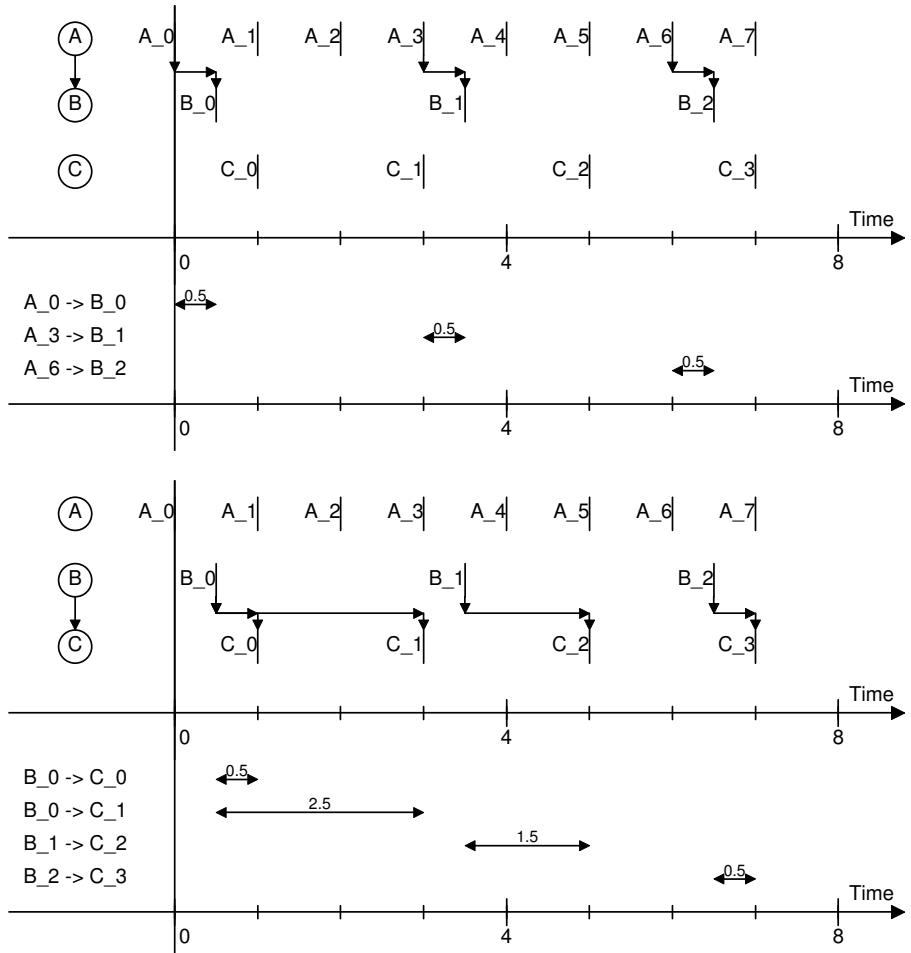


Figure 7.10: PathInstances between instances of event classes A and B, and B and C, respectively

The goal is to determine all feasible forward PathInstances going from instances of the event classes A over B to C. In Figure 7.11, both sets of feasible forward PathInstances have been joined by applying the algorithm shown in Listing 7.5. \square

Note that the algorithm can be applied to paths of arbitrary lengths and is not restricted to joining PathInstances of length two.

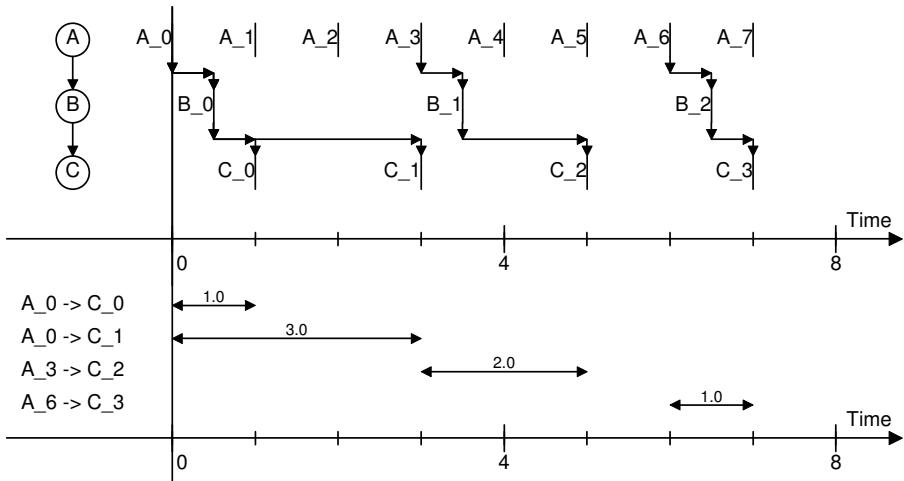


Figure 7.11: Joined PathInstances between instances of event classes A and B and C

7.3.4 Feasible Path Instances for Path Specifications of Arbitrary Length

The introduced algorithms for determining feasible forward (and backward) PathInstances and the algorithm to join PathInstances can be combined into a single algorithm to compute the set of all feasible forward (or backward) PathInstances for a PathSpecification in a single run.

Listing 7.6 shows an algorithm which computes all feasible forward PathInstances for a given PathSpecification (i.e., an ordered set of event or action classes).

```

1 def computeAllFeasibleForwardPathInstances(pathSpecification)
2     feasibleForwardPathInstances := []
3     for i in 1..pathSpecification.size-1 do
4         allPossiblePathInstances :=
5             computeAllPossiblePathInstances(pathSpecification[i-1],pathSpecification[i])
6         feasibleForwardPathInstances << extractFeasibleForwardPathInstances(allPossiblePathInstances);
7     end
8     joinedPathInstances := []
9     if feasibleForwardPathInstances ≠ []
10        if feasibleForwardPathInstances[0].size > 0
11            joinedPathInstances := feasibleForwardPathInstances[0]
12        end
13    end
14 end

```

7 Analysis of Event Traces from Monitoring and Simulation

```

11     for i in 1..feasibleForwardPathInstances.size - 1 do
12         joinedPathInstances := joinPathInstances(joinedPathInstances, feasibleForwardPathInstances[i])
13     end
14 end
15 return joinedPathInstances
16
17 end

```

Listing 7.6: Algorithm to compute all feasible forward PathInstances for a PathSpecification

In a first step, the ordered set of event classes `pathSpecification` denoting the PathSpecification is traversed, and the feasible forward PathInstances are computed for each pair of successive event classes. This results in a set of feasible forward PathInstances between the event instances of each two successive event classes. In a second step, the set of feasible forward PathInstances is traversed, and the currently considered feasible forward PathInstance is joined as successor PathInstance to the already determined joined PathInstance. The result from the second step is then returned.

An algorithm that computes all feasible backward PathInstances for an ordered set of event classes is shown in section E.8.2 in the Appendix.

The difference between both algorithms is in the computation of the feasible PathInstances (forward vs. backward PathInstances) and in the order in which the feasible forward (or backward) PathInstances are joined.

Example: Figure 7.12 shows the result of the algorithm from Listing 7.6 applied to the example from section 7.3.3.

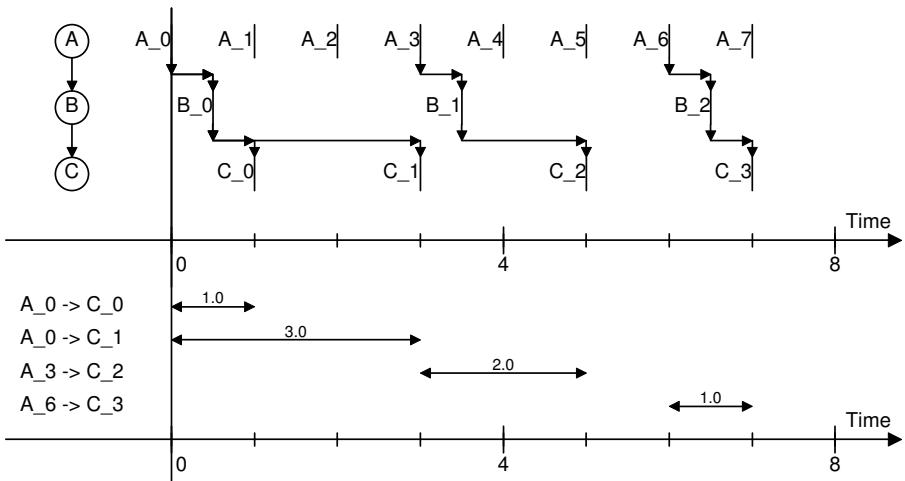


Figure 7.12: Event trace scenario with all feasible forward PathInstances for the PathSpecification $A \rightarrow B \rightarrow C$

The algorithm produces the same result as the sequentially applied algorithms from which the example in figure 7.11 has been constructed. \square

7.3.5 Shortest (feasible) Path Instances

In order to determine those feasible PathInstances we are interested in, we introduce algorithms to determine the shortest PathInstances from a set of feasible PathInstances.

Determining the Shortest Feasible Forward Path Instances

Listing 7.7 shows an algorithm to extract the shortest PathInstances from a set of forward PathInstances.

```

1 def extractShortestForwardPathInstances(pathInstances)
2   shortestPathInstances := Hash.new
3   pathInstances.each {
4     |pathInstance|
5     if shortestPathInstances[pathInstance.first] == nil
6       shortestPathInstances[pathInstance.first] := pathInstance
7     else
8       currentShortestPathInstance := shortestPathInstances[pathInstance.first]
9       if outerDistance(pathInstance.first, pathInstance.last) <
10         outerDistance(currentShortestPathInstance.first, currentShortestPathInstance.last)
11         shortestPathInstances[pathInstance.first] := pathInstance
12       end
13     end
14   }
15   return shortestPathInstances.values.uniq.sort

```

Listing 7.7: Algorithm to extract the shortest PathInstances from a set of forward PathInstances

For each event instance that is the first event instance of a forward PathInstance, there can be only one shortest feasible forward PathInstance. The algorithm employs a hash map to store the first event instances of the given PathInstances as its keys and the corresponding shortest feasible forward PathInstance as its value. The algorithm traverses the given set of feasible forward PathInstances `pathInstances`, and for each PathInstance evaluates if it is the shortest PathInstance from its given first event instance. The result is the set of shortest feasible forward PathInstances.

Example: Figure 7.13 depicts the scenario as figure 7.12 where the shortest PathInstances have been extracted.

There are three PathInstances which each are the shortest feasible forward PathInstances:

- A.0 → B.0 → C.0
- A.3 → B.1 → C.2
- A.6 → B.2 → C.3

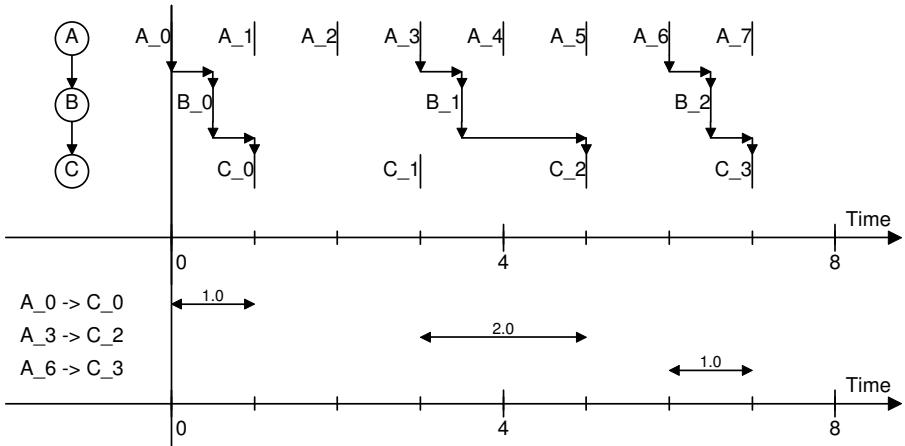


Figure 7.13: Shortest feasible forward PathInstances for the PathSpecification $A \rightarrow B \rightarrow C$

The outer distance between the first and the last event instance denotes the latency of the PathInstance. \square

The algorithm for determining the shortest feasible backward PathInstances is analogous and is shown in Appendix E.8.3.

It can be observed that the shortest backward PathInstances that can be determined from a set of feasible backward PathInstances are the same as the shortest forward PathInstances that can be determined from a set of feasible forward PathInstances. This means that in principle, the shortest PathInstances can be determined in two ways, either from feasible forward PathInstances or feasible backward PathInstances. This leads us to the definition of what we call an *effective PathInstance*.

Definition 29 An *Effective PathInstance* is a PathInstance where the first event instance, denoting a cause, and the last event instance, denoting an effect, stand in a relation where

- there must not exist another event instance that comes after the event instance denoting the cause which has an effect on the event instance denoting the effect, and
- there must not exist another event instance that comes prior to the event instance denoting the effect which is affected by the event instance denoting the cause.

As the outer distance between the first and the last event instance of an effective PathInstance denotes a meaningful timing property for our purposes, we refer to this as the PathDelay of an effective PathInstance, or the effective Pathdelay, for short.

Definition 30 *The effective PathDelay is the outer distance between the first and the last event instance of an effective PathInstance.*

Note that the longest feasible forward and backward PathInstances can be also determined by similar algorithms, however, there is no meaningful interpretation of these PathInstances and thus we are not interested in them. If these PathInstances were of interest, one would need to clarify their interpretation. In general, the set of longest feasible forward PathInstances is not equal to the set of longest feasible backward PathInstances.

So far, we have shown how effective PathInstances can be determined, and that the effective PathDelay is the timing property we are interested in. This, however, was with respect to a single PathSpecification. When multiple PathSpecifications are considered, e.g. PathSpecifications that describe signal paths between multiple different system inputs and outputs, further timing properties become of interest. These are the size of the synchronization between multiple correlated system input signals or multiple correlated output signals. In order to compute the size of the synchronization, the PathInstances of the PathSpecifications need to be further analyzed.

In the following, we first consider multiple PathSpecifications and then multiple effective PathInstances in relation to each other. In section 7.3.6, we introduce algorithms to compute the event classes at which two PathSpecifications intersect each other. Similarly to that, we introduce algorithms to compute the event instances at which two PathInstances intersect each other (section 7.3.7). Based on the latter, it is then possible to compute segments of PathInstances up to or starting from the event instance where the PathInstances intersect each other. Doing that for all PathInstances that intersect each other then allows us to compute the synchronization of certain event instances. The algorithms that we developed for the latter are presented in section 7.3.8.

7.3.6 Intersection Points of Path Specifications

In applications with two or more inputs and outputs, a signal path can exist between each input signal and output signal. If two or more input signals are used for the computation of a single output signal then the signal paths overlap. The same holds for the case of one input signal has an effect on two or more output signals. When two PathSpecifications include the same event classes they intersect with each other. The common event classes are referred to as *Intersection Points*.

Definition 31 *An IntersectionPoint of two (or more) PathSpecifications is an event class that is included in both (or all) PathSpecifications.*

Two special kinds of IntersectionPoints can be distinguished: *JoinPoints* and *ForkPoints*.

7 Analysis of Event Traces from Monitoring and Simulation

Definition 32 A **JoinPoint** of two (or more) PathSpecifications is an event class where for each PathSpecification, the preceding event classes to the event class denoting the JoinPoint in the PathSpecifications are distinct.

Definition 33 A **ForkPoint** of two (or more) PathSpecifications is an event class where for each PathSpecification, the succeeding event classes to the event class denoting the ForkPoint in the PathSpecifications are distinct.

Figure 7.14 depicts examples for overlapping PathSpecifications with IntersectionPoints.



(a) Overlapping PathSpecifications with a JoinPoint C as IntersectionPoint (b) Overlapping PathSpecifications with a ForkPoint D as IntersectionPoint

Figure 7.14: Examples for overlapping PathSpecifications with IntersectionPoints

- PathSpecification $A \rightarrow C \rightarrow D$ overlaps with PathSpecification $B \rightarrow C \rightarrow D$. The event classes C and D are IntersectionPoints. Event class C is a JoinPoint.
- PathSpecification $C \rightarrow D \rightarrow E$ overlaps with PathSpecification $C \rightarrow E \rightarrow F$. The event classes C and D are IntersectionPoints. Event class D is a ForkPoint.

Listing 7.8 shows an algorithm to compute the IntersectionPoints for two PathSpecifications.

```

1 def computeIntersectionPoints(pathSpecification1, pathSpecification2)
  return pathSpecification1 ∩ pathSpecification2
3 end

```

Listing 7.8: Algorithm to compute the IntersectionPoints for two PathSpecifications

The algorithm performs a set intersection of the two sets of event classes denoting the PathSpecifications.

Figure 7.15 depicts two PathSpecifications that intersect with each other.

For the PathSpecifications, the event classes C, D and E are determined as IntersectionPoints. However, according to our definitions, D is neither a JoinPoint nor a ForkPoint as the preceding and succeeding event classes in both PathSpecifications are the same. There are thus also IntersectionPoints which are neither JoinPoints nor ForkPoints.

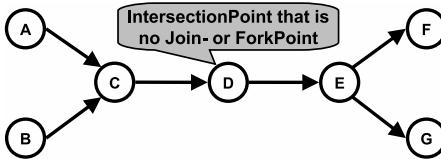


Figure 7.15: Example for overlapping PathSpecifications with IntersectionPoints that are not JoinPoints or ForkPoints

In order to evaluate if an `IntersectionPoint` is a `JoinPoint` or a `ForkPoint` the following two algorithms are introduced. Listing 7.9 shows an algorithm to determine if an `IntersectionPoint` (more generally, an event class that is considered as being a candidate for a `JoinPoint`) is a `JoinPoint` of two given `PathSpecifications`.

```

1 def isJoinPoint(joinPointCandidate, pathSpecification1, pathSpecification2)
2   i := pathSpecification1.index(joinPointCandidate)
3   j := pathSpecification2.index(joinPointCandidate)
4   result := false
5   if (i > 0) and (j > 0)
6     if pathSpecification1[i-1] ≠ pathSpecification2[j-1]
7       result := true
8     end
9   end
10  return result
11 end

```

Listing 7.9: Algorithm to determine if an `IntersectionPoint` is a `JoinPoint`

The algorithm evaluates the condition specified in our definition for a JoinPoint, i.e., whether there are distinct event classes in both PathSpecifications that directly precede the event class denoting the JoinPoint.

Similarly, listing 7.10 shows an algorithm to determine if an `IntersectionPoint` (more generally, an instance of an event or action class that is considered as being a candidate for a `ForkPoint`) is a `ForkPoint` of two given `PathSpecifications`.

```

1 def isForkPoint(forkPointCandidate, pathSpecification1, pathSpecification2)
2     i := pathSpecification1.index(forkPointCandidate)
3     j := pathSpecification2.index(forkPointCandidate)
4     result := false
5     if (i < pathSpecification1.size-1) and (j < pathSpecification2.size-1)
6         if pathSpecification1[i+1] ≠ pathSpecification2[j+1]
7             result := true
8         end
9     end
10    return result
11 end

```

Listing 7.10: Algorithm to determine if an IntersectionPoint is a ForkPoint

With the help of the latter two algorithms it is possible to determine only those IntersectionPoints of two PathSpecifications that are either JoinPoints or ForkPoints. The algorithm in listing 7.11 does that.

```

1 def computeJoinAndForkPoints(pathSpecification1, pathSpecification2)
2     intersectionPoints := []
3     intersectionPointCandidates := (pathSpecification1 ∩ pathSpecification2)
4     intersectionPointCandidates.each {
5         |intersectionPointCandidate|
6             if isJoinPoint(intersectionPointCandidate, pathSpecification1, pathSpecification2) ||
7                 isForkPoint(intersectionPointCandidate, pathSpecification1, pathSpecification2)
8                 intersectionPoints ≪ intersectionPointCandidate
9             end
10        }
11    return intersectionPoints
end

```

Listing 7.11: Algorithm to compute the JoinPoints and ForkPoints for two PathSpecifications

7.3.7 Intersection Point Instances of Path Instances

As described in the previous section, when two or more PathSpecifications include the same event or action classes they intersect with each other. The PathInstances that can be determined for such PathSpecifications can naturally intersect each other, too, meaning that there are event instances that are common to the PathInstances of the different PathSpecifications. The common event instances are referred to as *IntersectionPointInstances*.

Definition 34 *An IntersectionPointInstance of two or more PathInstances is an event instance that is included in all of the PathInstances.*

Analogously to IntersectionPoints and PathSpecifications, two special kinds of IntersectionPointInstances can be distinguished: *JoinPointInstances* and *ForkPointInstances*.

Definition 35 *A JoinPointInstance is an event instance common to two intersecting PathInstances. The event class of the JoinPointInstance is a JoinPoint with respect to the PathSpecifications of the two intersecting PathInstances.*

Definition 36 *A ForkPointInstance is an event instance common to two intersecting PathInstances. The event class of the ForkPointInstance is a ForkPoint with respect to the PathSpecifications of the two intersecting PathInstances.*

In the following we introduce algorithms to compute IntersectionPointInstances, JoinPointInstances and ForkPointInstances.

Listing 7.12 shows an algorithm which determines the IntersectionPointInstances of two sets of PathInstances.

```

def computeIntersectionPointInstances(pathInstances1, pathInstances2)
2     intersectionPointInstances := []
3     pathInstances1.each{
4         |pathInstance1|
5             pathInstances2.each{
6                 |pathInstance2|
7                     if pathInstance1.intersects(pathInstance2)
8                         intersectionPointInstances ≪ pathInstance1
9                     end
10                }
11            }
12        }
13    return intersectionPointInstances
end

```

```

6     |pathInstance2|
7     intersectionPointInstances <= (pathInstance1 ∩ pathInstance2)
8   }
9
10    return intersectionPointInstances.flatten.uniq.sort
end

```

Listing 7.12: Algorithm to compute the IntersectionPointInstances of two sets of PathInstances

The PathInstances of the two sets are intersected with each other, resulting in the IntersectionPointInstances.

The algorithm in listing 7.13 is then used to extract the JoinPointInstances from a set of IntersectionPointInstances for a given JoinPoint.

```

1 def extractJoinPointInstancesFromIntersectionPointInstances(intersectionPointInstances, joinPoint)
2   joinPointInstances := []
3   intersectionPointInstances.each {
4     |intersectionPointInstance|
5       if intersectionPointInstance.event ≡ joinPoint
6         joinPointInstances <= intersectionPointInstance
7     end
8   }
9   return joinPointInstances
end

```

Listing 7.13: Algorithm to extract the JoinPointInstances from a set of IntersectionPointInstances for a given JoinPoint

Similar, the algorithm in listing 7.14 is the used to extract the ForkPointInstances from a set of IntersectionPointInstances for a given ForkPoint.

```

1 def extractForkPointInstancesFromIntersectionPointInstances(intersectionPointInstances, forkPoint)
2   forkPointInstances := []
3   intersectionPointInstances.each {
4     |intersectionPointInstance|
5       if intersectionPointInstance.event ≡ forkPoint
6         forkPointInstances <= intersectionPointInstance
7     end
8   }
9   return forkPointInstances
10 end

```

Listing 7.14: Algorithm to extract the ForkPointInstances from a set of IntersectionPointInstances for a given ForkPoint

With the algorithms introduced in this section it is possible to determine at which event instances the PathInstances of two or more PathSpecifications intersect with each other. In the following, we introduce algorithms which in the end allow us to determine the size of the synchronization between certain event instances.

7.3.8 Join and Fork Path Segments from Intersecting Path Instances

With the help of a JoinPointInstance, we can determine segments of PathInstances that lead from their first event instance to a given JoinPointInstance. Such segments are called *JoinPathSegments*.

Definition 37 A *JoinPathSegment* is a subset of the event instances of a Path-Instance (i.e., a segment) where the first event instance of the JoinPathSegment corresponds to the first event instance in the PathInstance, and the last event instance of the JoinPathSegment corresponds to the JoinPointInstance.

Listing 7.15 shows an algorithm to extract the JoinPathSegments to a given JoinPointInstance from a set of PathInstances.

```

1 def extractJoinPathSegmentsToJoinPointInstanceFromPathInstances(pathInstances, joinPointInstance)
2   joinPathSegments := Hash.new
3   pathInstances.each {
4     |pathInstance|
5     joinPathSegment := []
6     pathInstance.each {
7       |eventInstance|
8       joinPathSegment << eventInstance;
9       if eventInstance == joinPointInstance
10         if joinPathSegments[eventInstance] == nil
11           joinPathSegments[eventInstance] := []
12         end
13         joinPathSegments[eventInstance] << joinPathSegment;
14         joinPathSegments[eventInstance] := joinPathSegments[eventInstance].uniq
15         break;
16       end
17     }
18   }
19   return joinPathSegments
20 end

```

Listing 7.15: Algorithm

to extract the JoinPathSegments to a given JoinPointInstance from a set of PathInstances

In order to obtain the JoinPathSegments for more than one JoinPointInstance, the following algorithm is applied.

```

1 def extractJoinPathSegmentsToJoinPointInstancesFromPathInstances(pathInstances, joinPointInstances)
2   joinPathSegments := Hash.new
3   joinPointInstances.each {
4     |joinPointInstance|
5     joinPathSegments.merge!(
6       extractJoinPathSegmentsToJoinPointInstanceFromPathInstances(pathInstances, joinPointInstance))
7   }
8   return joinPathSegments
9 end

```

Listing 7.16: Algorithm to extract the JoinPathSegments to a given set of JoinPointInstance from a set of PathInstances

Similarly, we can determine segments of PathInstances that lead from a given ForkPointInstance to their last instance of an event class. Such segments are called *ForkPathSegments*.

Definition 38 A *ForkPathSegment* is a subset of the event instances of a Path-Instance (i.e., a segment) where the first event instance of the ForkPathSegment corresponds to the ForkPointInstance, and the last event instance of the ForkPath-Segment corresponds to the last event instance in the PathInstance.

Listing 7.17 shows an algorithm to extract the ForkPathSegments to a given Fork-PointInstance from a set of PathInstances.

```

1 def extractForkPathSegmentsFromForkPointInstanceFromPathInstances(pathInstances, forkPointInstance)
2   forkPathSegments := Hash.new
3   pathInstances.each {
4     |pathInstance|
5     forkPathSegment := []
6     pathInstance.reverse.each {
7       |eventInstance|
8       forkPathSegment << eventInstance;
9       if eventInstance == forkPointInstance
10         if forkPathSegments[eventInstance] == nil
11           forkPathSegments[eventInstance] := []
12         end
13         forkPathSegments[eventInstance] << forkPathSegment.reverse;
14         forkPathSegments[eventInstance] := forkPathSegments[eventInstance].uniq
15         break;
16       end
17     }
18   }
19   return forkPathSegments
end

```

Listing 7.17: Algorithm

to extract the ForkPathSegments to a given ForkPointInstance from a set of PathInstances

In order to obtain the ForkPathSegments for more than one ForkPointInstance from a set of PathInstances, the following algorithm is applied.

```

def extractForkPathSegmentsFromForkPointInstancesFromPathInstances(pathInstances,
  forkPointInstances)
2   forkPathSegments := Hash.new
3
4   forkPointInstances.each {
5     |forkPointInstance|
6     forkPathSegments.merge!(
7       extractForkPathSegmentsFromForkPointInstanceFromPathInstances(pathInstances,
8         forkPointInstance))
9   }
10  return forkPathSegments
end

```

Listing 7.18: Algorithm to extract the ForkPathSegments to a given set of ForkPointInstance from a set of PathInstances

7 Analysis of Event Traces from Monitoring and Simulation

JoinPathSegments and ForkPathSegments of intersected shortest feasible PathInstances allow to determine the size of the effective synchronization between the first and last event instances of different PathInstances with respect to a common IntersectionPointInstance.

Example: Figure 7.16 depicts an example of two PathSpecifications $A \rightarrow C \rightarrow D \rightarrow E$ and $B \rightarrow C \rightarrow D \rightarrow F$ that intersect with each other.

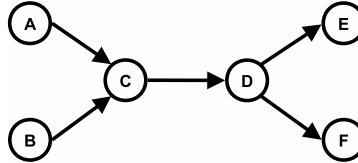


Figure 7.16: Example for two intersecting PathSpecifications

The IntersectionPoints that can be determined are C and D, whereby C is a JoinPoint and D is a ForkPoint. Note that the observable entities are action classes. Figure 7.17 depicts a scenario with action instances for the action classes referred to by the two PathSpecifications. The action instances have constant durations. Furthermore, the shortest feasible forward PathInstances that can be determined are shown. As can be seen in the figure, the PathInstances of the two PathSpecifications intersect each other at the IntersectionPoints. The action instances C_1 and C_3 are JoinPointInstances, the action instances D_0 and D_1 are ForkPointInstances. In order to determine the synchronization between action classes A from the first PathSpecification and B from the second PathSpecification with respect to the common JoinPoint C the JoinPathSegments of the intersected PathInstances are computed. Similarly, in order to determine the synchronization between action classes E from the first PathSpecification and F from the second PathSpecification with respect to the common ForkPoint D the ForkPathSegments of the intersected PathInstances are computed. Figure 7.18 depicts the scenario including the JoinPathSegments and ForkPathSegments that have been determined with the help of the algorithms. In the lower part of the figure, the size of the JoinPathSegments and ForkPathSegments is shown. The following conclusions can be drawn:

- The action instances of A and B are synchronized with respect to instances of the JoinPoint C within 1.9 time units.
- The action instances of E and F are synchronized with respect to instances of the ForkPoint D within 2.3 and 1.3 time units.

□

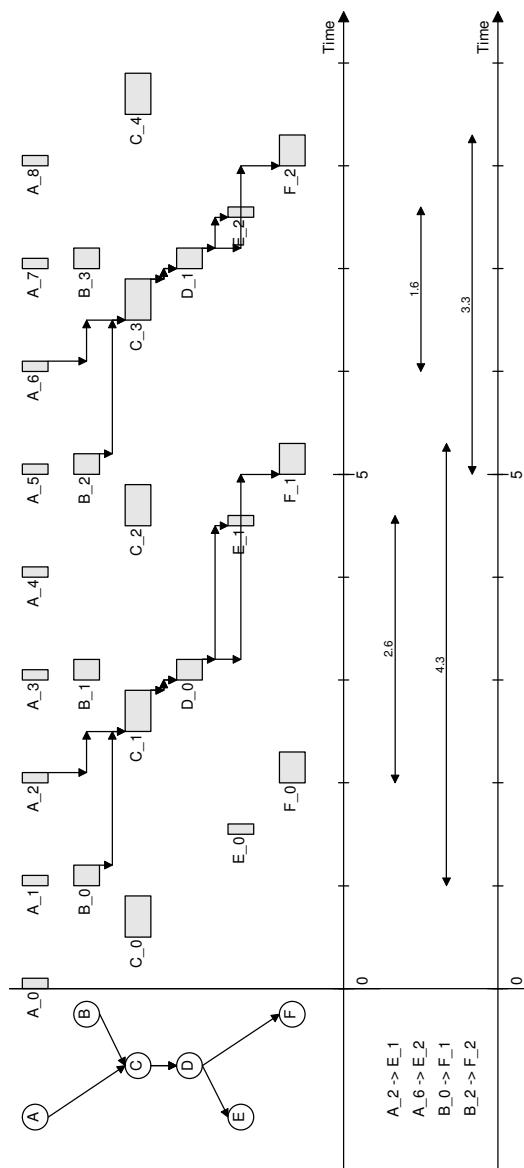
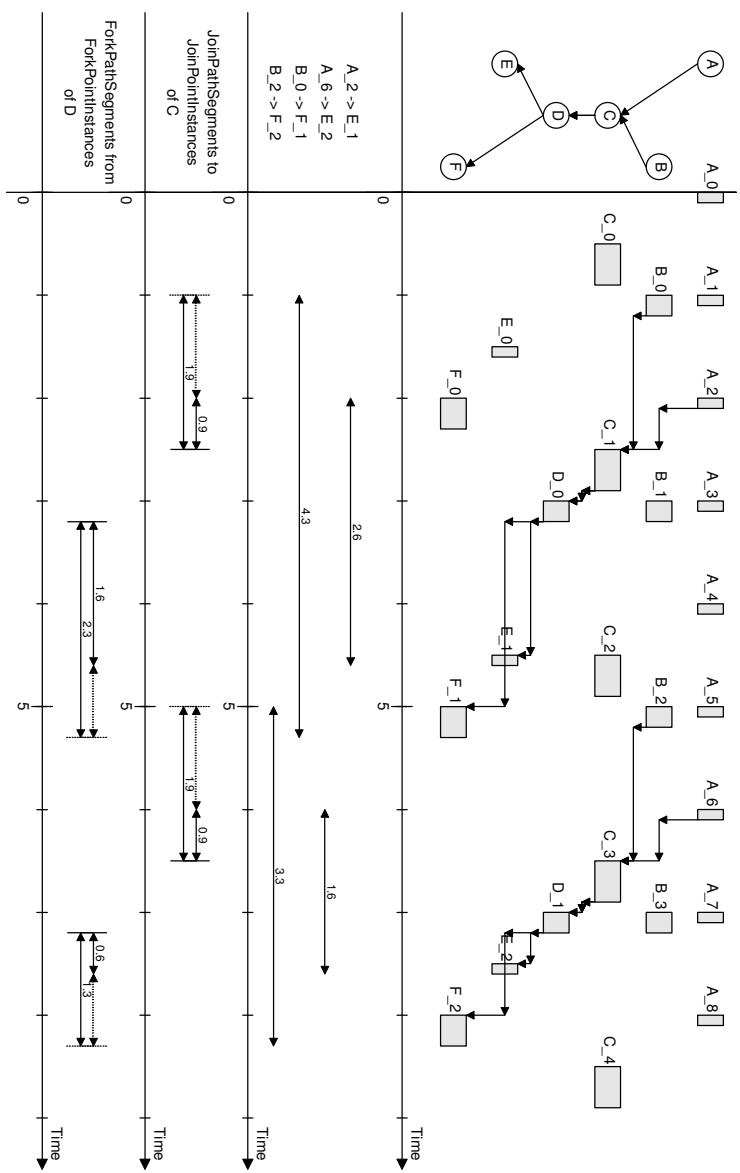


Figure 7.17: Event trace scenario of action instances with shortest forward PathInstances

7 Analysis of Event Traces from Monitoring and Simulation

Figure 7.18: Event trace scenario of action instances with shortest feasible forward PathInstances, JoinPathSegments of C and ForkPathSegments



From the example, several further conclusions can be drawn:

- The length of the PathInstances of the same PathSpecification can vary to large extents.
- The synchronization of action instances with respect to a common JoinPointInstance or ForkPointInstance can also vary to large extents.

For our purposes, Gantt diagrams are not the best suitable means to show the timing properties of PathInstances, JoinPathSegments and ForkPathSegments. Firstly, only small event trace scenarios (or excerpts of larger ones) can be visualized such that the Gantt diagrams are still readable and understandable. Secondly, it is difficult to see the development of the timing properties over time, i.e. to see how they do - or not do - vary over time. In the following section, a new runtime-oriented diagram to visualize the timing properties of PathInstances is introduced.

7.4 Timing Oscilloscope Diagrams

So far, Gantt diagrams have been used to visualize event and action classes and their instances, to show the effective PathInstances that can be determined by the path analysis algorithms and to show the timing properties that can be determined based on the effective PathInstances. The approach of evaluating the degree of fulfillment of timing requirements by means of Gantt diagrams, however, is not user-friendly and efficient for large sets of data, e.g., as recorded during a simulation or monitoring experiment for a larger time period of time. In a Gantt diagram, each effective PathInstance needs to be considered separately, and it must be evaluated if the timing requirements that are imposed on the corresponding PathSpecification are satisfied by the effective PathInstance (e.g., a PathDelayRequirement). This case-by-case evaluation is cumbersome and also time consuming.

In order to ease the interpretation of timing properties with respect to timing requirements, we introduce a new type of diagram which is derived from the Gantt diagrams: the *Timing Oscilloscope Diagram*.

In a Timing Oscilloscope Diagram, the size of the determined timing properties is plotted on a time axis as in an oscilloscope. The parameters that are given for a specific timing requirement can also be shown in a Timing Oscilloscope Diagram such that their degree of fulfillment can directly be evaluated. Timing Oscilloscope Diagrams allow to consider timing properties over a larger time frame compared to a case-per-case evaluation as in Gant Diagrams. Also, the time axis of a Timing Oscilloscope Diagram can better be scaled to show timing properties for larger time frames without the diagrams being cluttered with information compared to Gant diagrams.

Compared to Gant diagrams, Timing Oscilloscope Diagrams offer the advantage that violations of timing requirements can easily be identified. A further advantage is that it can be evaluated for how long violations are effective. This can be used as a basis to decide if a violation of a timing requirement can still be tolerated or not.

7 Analysis of Event Traces from Monitoring and Simulation

In the following, we describe how the Timing Oscilloscope Diagrams are constructed for the different kinds of timing requirements and timing properties. Section 7.4.1 presents how Timing Oscilloscope Diagrams are constructed for the ReactionTimes of reactive real-time applications such that the degree of fulfillment of ReactionTimeRequirements can be evaluated. Section 7.4.2 the presents how Timing Oscilloscope Diagrams are constructed for control application. Here, we individually consider the different kinds timing requirements (PathDelayRequirements, Input- and OutputIntervalDelayRequirements, Input- and OutputSynchronizationRequirements) as the corresponding Timing Oscilloscope Diagrams are constructed differently in each case.

7.4.1 Reactive Real-Time Applications

For reactive real-time applications, we only consider timing requirements on reaction times. Thus, in the following, only the construction of Timing Oscilloscope Diagrams for ReactionTimeRequirements and ReactionTimes is explained.

Reaction Time Requirements and Reaction Times

Figure 7.19 shows an example scenario with event classes and their instances.

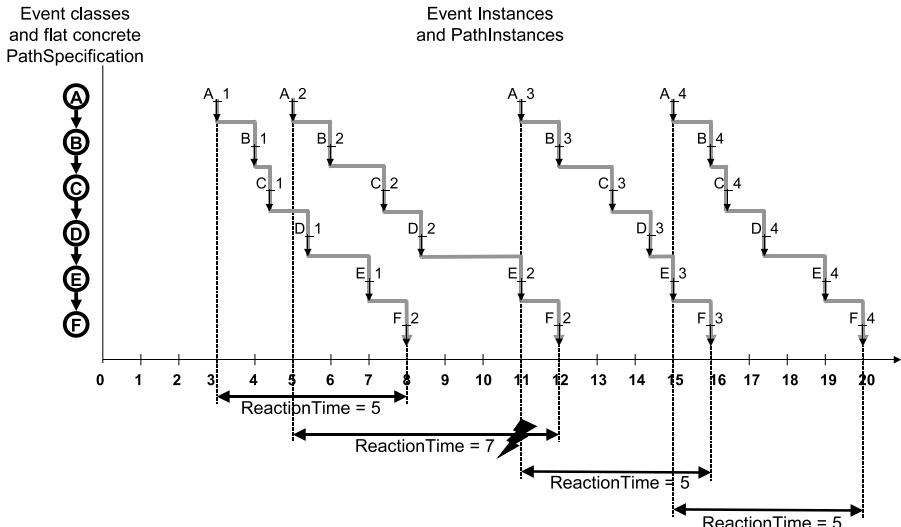


Figure 7.19: Event trace scenario of event instances with identified PathInstances and determined ReactionTimes

The PathSpecification shown on the left side specifies an order on the event classes. It is assumed that a ReactionTimeRequirement is associated with the PathSpecification, demanding a minimum reaction time of 4 time units and a maximum reaction time of 6 time units. The effective PathInstances that can be determined by means of the path analysis algorithms are also shown. Based on the effective PathInstances, the size of the ReactionTimes can be determined.

Figure 7.20 shows the corresponding Timing Oscilloscope Diagram where the ReactionTimeRequirement is shown through its maximum and minimum value (red dashed lines). Allowed ReactionTimes should be within this range. The ReactionTimes that have been determined for the effective PathInstances are plotted in the diagram. A new value of a ReactionTime is plotted from the point in time when the corresponding PathInstance ends, i.e., when the last event instance ends, and thus the ReactionTime becomes effective. As can be seen from the figure, the ReactionTimeRequirement is temporarily violated as it rises to 7 time units for a certain amount of time.

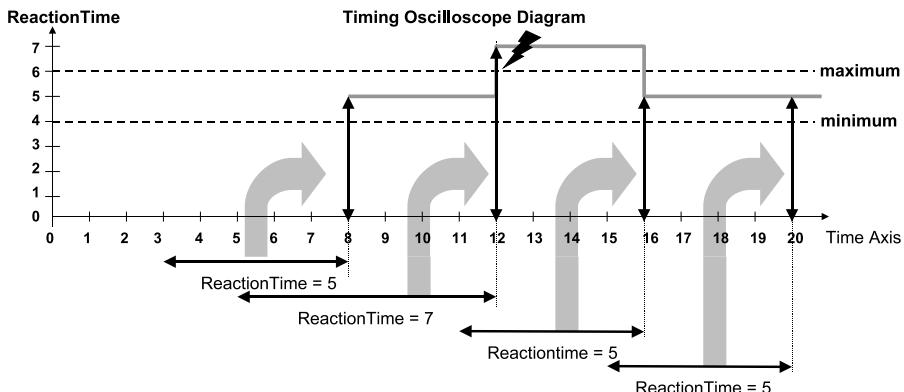


Figure 7.20: Timing Oscilloscope Diagram with ReactionTimeRequirement and ReactionTimes

7.4.2 Control Applications

For the different types of timing requirements that can be formulated for control applications, i.e., PathDelayRequirements, Input- and OutputIntervalDelayRequirements and Input- and OutputSynchronizationRequirements, the corresponding timing properties are determined differently based on effective PathInstances that are determined by means of the path analysis algorithms. In the following, we thus first consider how Timing Oscilloscope Diagrams are constructed for PathDelayRequirements and PathDelays, then Input- and OutputIntervalDelayRequirements and

7 Analysis of Event Traces from Monitoring and Simulation

Input- and OutputIntervalDelays, and finally Input- and OutputSynchronization- Requirements and their corresponding timing properties, i.e. the PathDelays of JoinPathSegments and ForkPathSegments.

Path Delay Requirements and Path Delays

PathDelayRequirements are very similar to ReactionTimeRequirements as they both specify a timing requirement on the outer interval of an effective PathInstance. The distinction lies within the parameters provided by the timing requirements to express the imposed bounds on the timing properties. These parameters are application-specific and different for reactive real-time applications and control applications.

Figure 7.21 depicts an example scenario with event classes and their instances. The PathSpecification is the same as in the example for the ReactionTimeRequirement. It is assumed that a PathDelayRequirement is associated with the PathSpecification. The nominal PathDelay should be 5 time units whereby a deviation of 1 time unit is acceptable. The effective PathInstances that can be determined by means of the path analysis algorithms are also shown. Based on the PathInstances, the size of the PathDelays can be determined.

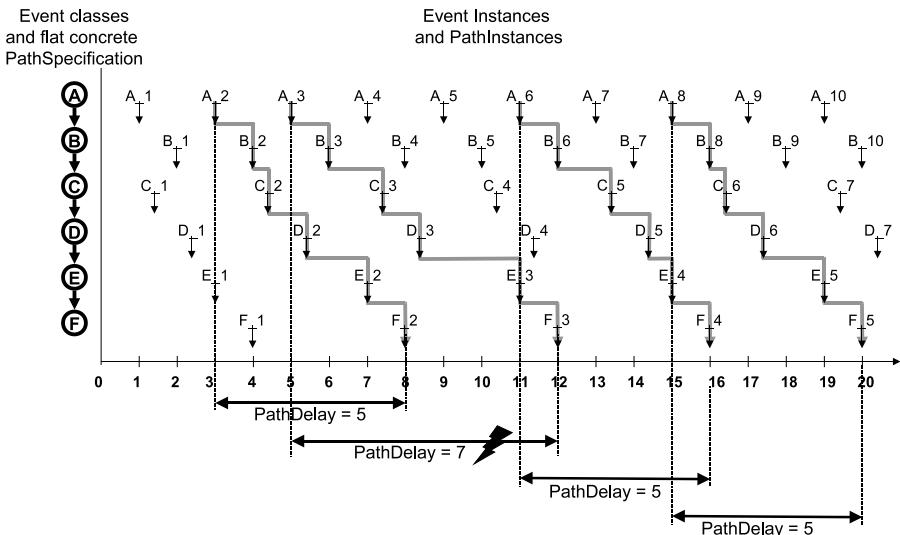


Figure 7.21: Event trace scenario of event instances with identified PathInstances and determined PathDelays

Figure 7.22 depicts the corresponding Timing Oscilloscope Diagram where the PathDelayRequirement is shown though its nominal value (green dashed line) and

the allowed positive and negative deviation from the latter (red dashed lines). Effective PathDelays that satisfy the PathDelayRequirement must be within the bounds of the red dashed lines. The effective PathDelays that have been determined are plotted in the Timing Oscilloscope Diagram. A new value of an effective PathDelay is plotted from the point in time when the corresponding PathInstance ends, i.e., when the last event instance ends.

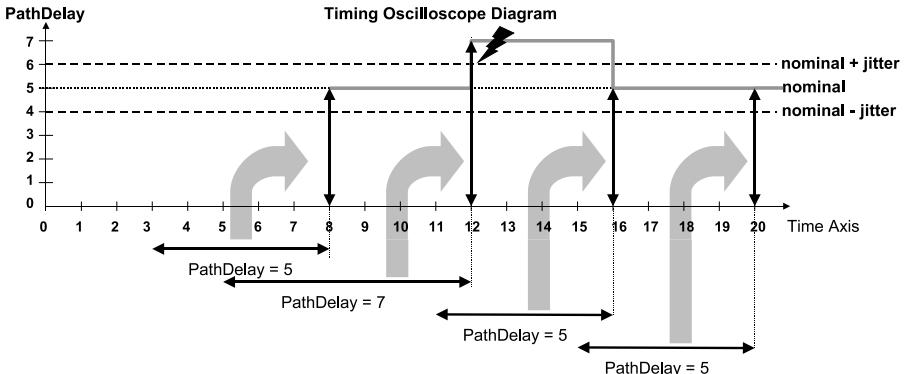


Figure 7.22: Timing Oscilloscope Diagram with PathDelayRequirement and PathDelays

As can be seen from the figure, the PathDelayRequirement is temporarily violated as it rises to 7 time units for a certain amount of time.

Interval Delay Requirements and Interval Delays

Figure 7.23 depicts the same scenario with the event classes and their instances. It is assumed that an InputIntervalDelayRequirement is associated with the Path-Specification on the left side that specifies that the time between instances of event class A that are effective should be 4 time units in the nominal case whereby a deviation of 1 time unit is acceptable.

Figure 7.24 depicts the corresponding Timing Oscilloscope Diagram. The InputIntervalDelayRequirement is shown through its nominal value (green dashed line) and the allowed positive and negative deviation (red dashed lines). The Input-IntervalDelays should be within the range determined by the red dashed lines. The InputIntervalDelays that have been determined based on the effective PathInstances are plotted in the diagram. A specific value of an InputIntervalDelay is plotted for the time that it is effective, i.e., the time between two instances of event class A that belong to two consecutive effective PathInstances. As can be seen from the figure, the InputIntervalDelayRequirement is temporarily violated as it is below 3 time units and the rises to 6 time units for a certain amount of time.

7 Analysis of Event Traces from Monitoring and Simulation

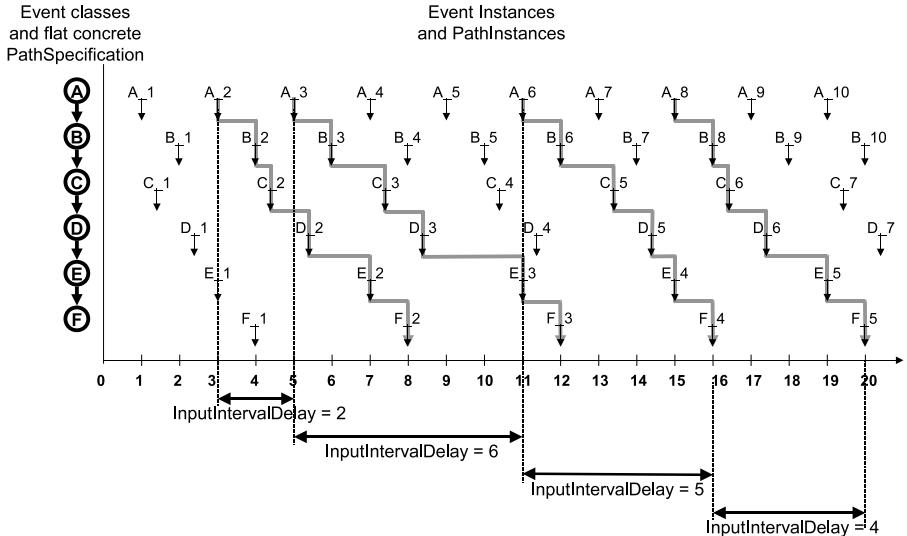


Figure 7.23: Event trace scenario of event instances with identified PathInstances and determined InputIntervalDelays

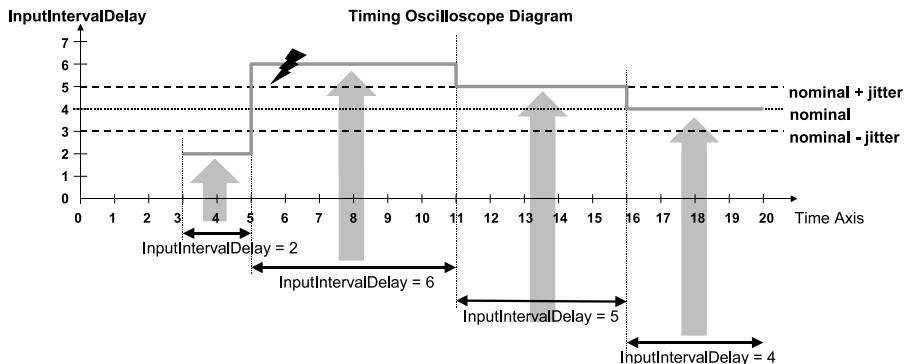


Figure 7.24: Timing Oscilloscope Diagram with InputIntervalDelayRequirement and InputIntervalDelays

The construction of Timing Oscilloscope Diagrams for OutputIntervalDelays is analogous. Here, the OutputIntervalDelays are plotted on a time axis, showing the development of the effective actuation rate over time.

Synchronization Timing Requirements and Synchronization Intervals

Figure 7.25 depicts an example scenario with event classes and their instances. It is assumed that an InputSynchronizationRequirement is associated with two Path-Specifications that intersect each other, and where the event class D is a JoinPoint. The InputSynchronizationRequirement demands that the interval delays of the Join-PathSegments $A_1 \rightarrow B_1 \rightarrow C_1 \rightarrow D$ and $A_2 \rightarrow B_2 \rightarrow C_2 \rightarrow D$ must be less than 3 time units. The PathInstances that have been determined by means of the path analysis algorithms and which are relevant for computing the InputSynchronization-Intervals are also shown.

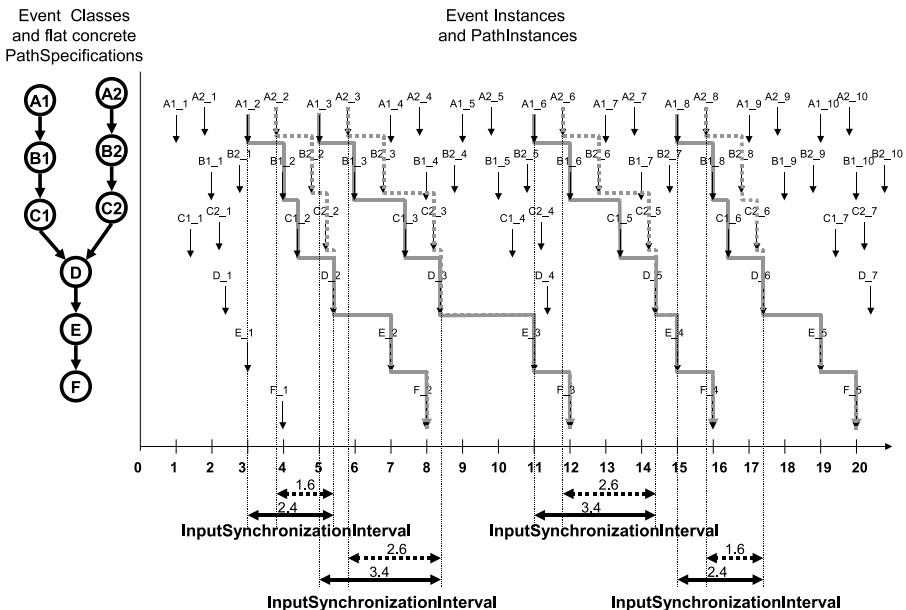


Figure 7.25: Event trace scenario of event instances with identified PathInstances and determined InputSynchronizationIntervals

Figure 7.26 depicts the corresponding Timing Oscilloscope Diagram where the value of the interval specified by the InputSynchronizationRequirement is shown as red dashed line. The InputSynchronizationIntervals should be below the dashed line. The InputSynchronizationIntervals that have been determined by means of the path analysis algorithms are plotted in the diagram.

7 Analysis of Event Traces from Monitoring and Simulation

Note that the solid green line corresponds to the first JoinPathSegment (i.e., A1 → B1 → C1 → D), and the dotted green line corresponds to the second JoinPathSegment (i.e., A2 → B2 → C2 → D) referred to by the InputSynchronizationRequirement. As can be seen from the Timing Oscilloscope Diagram, the InputSynchronizationRequirement is temporarily violated as the InputSynchronizationIntervals of the first JoinPathSegment cross the specified bound for the synchronization interval (red dashed line).

The construction of Timing Oscilloscope Diagrams for OutputSynchronizationRequirements and OutputSynchronizationIntervals is analogous.

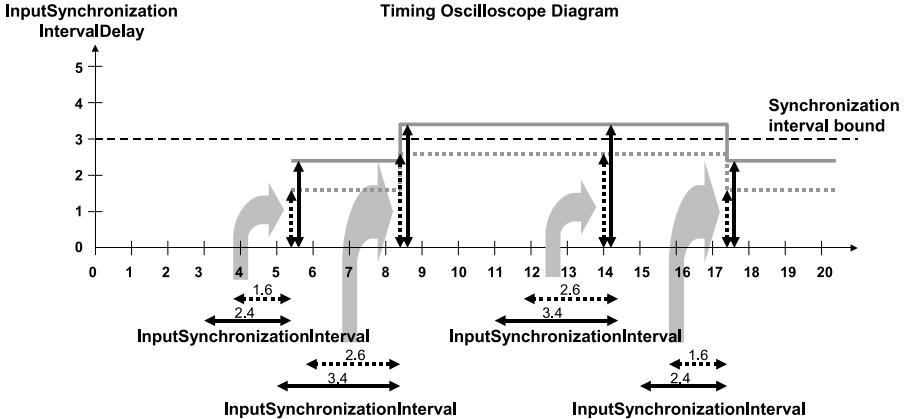


Figure 7.26: Timing Oscilloscope Diagram with InputSynchronizationRequirement and InputSynchronizationIntervals

7.5 Analysis of the Example AUTOSAR System

To demonstrate the applicability and benefits of the concepts introduced in this chapter, several monitoring experiments have been conducted with the example AUTOSAR system of the simple SISO control application. The SISO control application is realized in terms of the application software of an AUTOSAR system with a single ECU.

Due to the large amount of configurations that are possible for the SISO control application we only discuss a selected set of cases in this thesis. However, many more interesting cases have been analyzed that could equally be presented here.

Furthermore, due to the limitation of the employed monitoring tool RTA-TRACE to single-ECU systems, monitoring experiments could only be performed for AUTOSAR systems where the overall application software realizing the SISO control application is mapped to a single ECU. This, however, does not lower the significance of the results and the approach in general.

The focus of the conducted monitoring experiments was on the analysis of the effects of AUTOSAR-specific design and configuration decisions, especially on the choice of implicit or explicit Sender/Receiver communication, and their influence on the application-specific timing properties. Two different configurations for the software architecture, more precisely the employed communication pattern, are considered, combined with two different ECU configurations for inter-task and intra-task communication. For each configuration, a monitoring experiment has been conducted in order to determine the timing properties.

Furthermore, two simulation experiments to analyze the effects for two special cases have been performed. These are the synchronization of sensor data acquisition actions in the case of a multiple-input control application, and the effects of clock drift on the timing properties. For the simulation experiment we have employed a simple discrete-event simulator.

To conduct the monitoring and simulation experiments, an instrumentation has been derived from the logical PathSpecification that describes the feedback path of the SISO control application as described in the previous sections. The instrumentation is used to obtain instances of the AUTOSAR-specific event and action classes such that timing properties can be determined by means of the path analysis algorithms. The development of the timing properties is shown by means of Timing Oscilloscope Diagrams.

In the following, the AUTOSAR system in which the SISO control application is realized and the aspects that are common to all subsequently considered configurations are described. Then, three different configurations for which the timing properties have been determined by means of monitoring experiments are discussed. These are the comparison of implicit and explicit Sender/Receiver communication and the comparison of intra-task and inter-task communication, whereby for the latter a multi-rate configuration has been analyzed.

7.5.1 Description of AUTOSAR System and Base Configuration

Figure 7.27 shows the example AUTOSAR system of the SISO control application, including the instrumentation and the event detection mechanism employed in the monitoring experiments⁴.

The sensor, controller and actuator parts of the SISO control application are modeled as individual AtomicSoftwareComponentTypes. Each AtomicSoftwareComponentType contains a single RunnableEntity within its InternalBehavior.

⁴In the simulation experiments, the time values for the instances of events and action classes are taken directly from the globally visible ideal clock

7 Analysis of Event Traces from Monitoring and Simulation

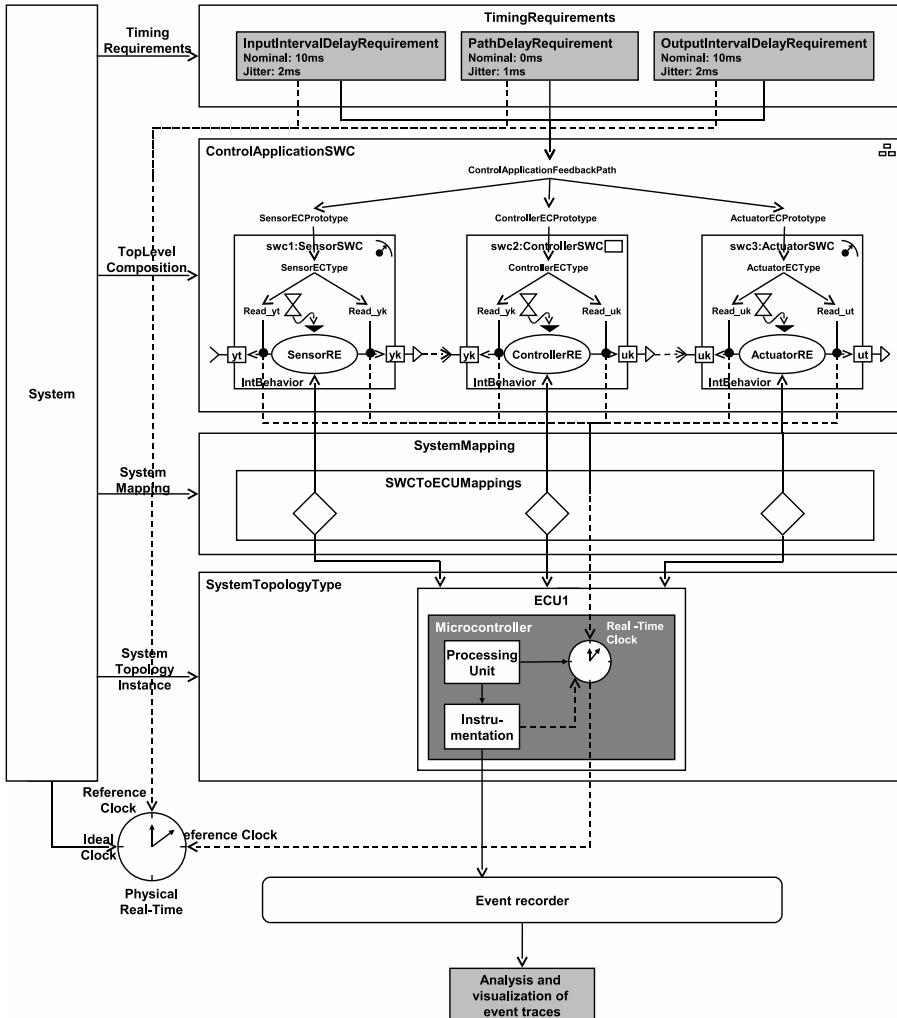


Figure 7.27: Example AUTOSAR system of the SISO control application used for monitoring experiments

The RunnableEntities read the value of the DataElementPrototype in the required PortPrototype, perform a data transformation and write the result on the DataElementPrototype in the provided PortPrototype.

Note that in a microcontroller-based implementation, the data transformation of input data to output data takes a specific amount of time due to the discrete-time clocked processing of code instructions.

Each of the RunnableEntities is triggered by a single TimingEvent. The periods of the TimingEvents are given in the considered configurations.

To describe the timing requirements of the SISO control application, the concepts of the Timing Model for AUTOSAR are applied. The feedback path of the SISO control application is modeled by means of a hierarchical event chain. As the hierarchical event chain correctly captures the causal relations between the reading and writing actions of the RunnableEntities, the hierarchical event chain denotes a logical PathSpecification. The PathSpecification is associated with three TimingRequirements that express the application-specific timing requirements of the SISO control application. These are:

- the PathDelayRequirement on the time delay between the start and the end of related effective sensor sampling and actuator effectuation actions ($\tau_{SCA} = 0 \text{ ms} \pm 1 \text{ ms}$),
- an InputIntervalDelayRequirement on the time between the start of consecutive effective input data acquisition actions ($h_{sampling} = 10 \text{ ms} \pm 2 \text{ ms}$), and
- an OutputIntervalDelayRequirement on the time between the end of consecutive effective output data effectuation actions ($h_{actuation} = 10 \text{ ms} \pm 2 \text{ ms}$).

The SystemTopologyInstance of the AUTOSAR system only contains a single ECUInstance. The complete software architecture of the AUTOSAR system is mapped to this ECUInstance as part of the overall system configuration. We only consider single-ECU systems and not distributed systems as the tool employed in our monitoring experiments, RTA-TRACE, cannot yet obtain synchronized timing data from multiple ECUs. This, however, is a prerequisite for the determination of valid and meaningful timing properties.

Figure 7.28 shows the order of the RTEAPIActions referred to by the logical PathSpecification. This corresponds to the flat logical PathSpecification.

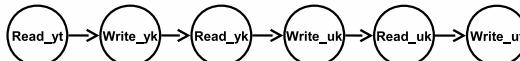


Figure 7.28: Order of RTEAPIActions described by the logical PathSpecification in figure 7.27

The timing properties for three different example configurations are determined and analyzed by means of monitoring experiments. The following aspects are specific for each configuration and have an influence on the timing properties:

- The RunnableEntities communicate with each other according to the Sender/Receiver communication pattern. The access of a RunnableEntity to

7 Analysis of Event Traces from Monitoring and Simulation

a DataElementPrototype can be either implicit or explicit. This decision is part of the software architecture design and is specified as part of a configuration. The choice of the communication pattern has a direct influence on the AUTOSAR-specific event and action classes of a concrete PathSpecification and consequently on the timing properties.

- As part of the ECU configuration, the basic software modules OS and RTE need to be configured such that the RunnableEntities are adequately executed. In the OS configuration, one or more OS-tasks need to be specified. These also need to be configured with a priority according to which they are scheduled by the real-time operating system of the AUTOSAR ECU system. In the RTE configuration, the RunnableEntities must be assigned to the OS-tasks, and the position of the RunnableEntities within the OS-task they are assigned to must be specified. The decisions taken during the ECU configuration also have a direct influence on the timing properties. The ECU configuration details are given as part of a considered configuration.

Event trace data has been captured for a time frame of 1000ms, measured from the start of the runtime of the AUTOSAR system. This is sufficient for determining meaningful timing properties while the obtained figures (Timing Oscilloscope Diagrams) are still readable. Timing properties can also be determined for longer time frames if event trace data was captured for a longer period of time.

7.5.2 Configuration 1: Implicit Sender/Receiver communication, Single-Rate Time-Triggered Activation, Single OS-Task

In the first configuration, the RunnableEntities communicate with each other according to the implicit Sender/Receiver communication pattern. Furthermore, the TimingEvents that activate the RunnableEntities all have the same period and are all assigned to a single OS-task. The positions of the activations of the RunnableEntities in the OS-task are chosen such that their execution order complies with the order of the event and action classes given in the logical PathSpecification. Table 7.1 provides an overview of the configuration details.

RunnableEntity	SensorRE	ControllerRE	ActuatorRE
Triggering	TimingEvent (10ms)	TimingEvent (10ms)	TimingEvent (10ms)
Communication Patterns	implicit read implicit write	implicit read implicit write	implicit read implicit write
OS-Task (Priority) Position in OS-Task	TaskA (1) 1	TaskA (1) 2	TaskA (1) 3

Table 7.1: Configuration 1: Implicit Sender/Receiver communication, single-rate time-triggered activation, single OS-task

7.5 Analysis of the Example AUTOSAR System

Due to the implicit Sender/Receiver communication and the ECU configuration decisions, the logical flat PathSpecification shown in figure 7.28 needs to be augmented in order to adequately account for the copying of data between the RTE and the buffers of the OS-task. Figure 7.29 shows the flat concrete PathSpecification.

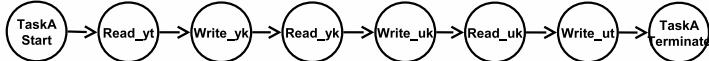
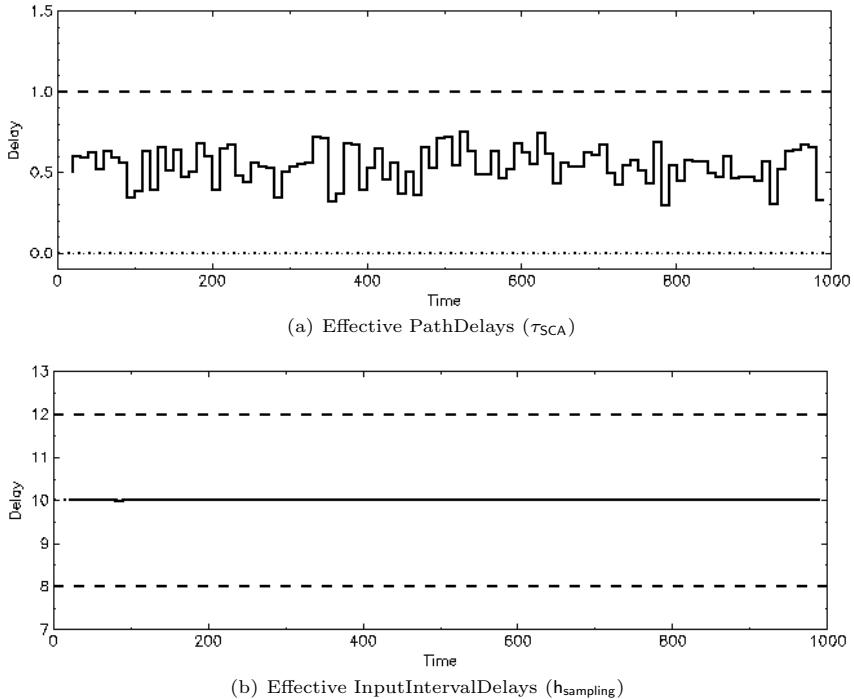


Figure 7.29: Order of AUTOSAR-specific Observable for configuration 1

The start of the OS-task **TaskA** needs to be observed before the implicit read action performed by the **SensorRE**, and the end of the OS-task needs to be observed after the implicit write action performed by the **ActuatorRE**.

Figure 7.30 shows the Timing Oscilloscope Diagrams for the effective PathDelays, InputIntervalDelays and OutputIntervalDelays.



The PathDelays are varying randomly due to the variable execution times of the RunnableEntities, however, the PathDelayRequirement is satisfied as the PathDelays are within the allowed tolerance bound. The InputIntervalDelays are almost

7 Analysis of Event Traces from Monitoring and Simulation

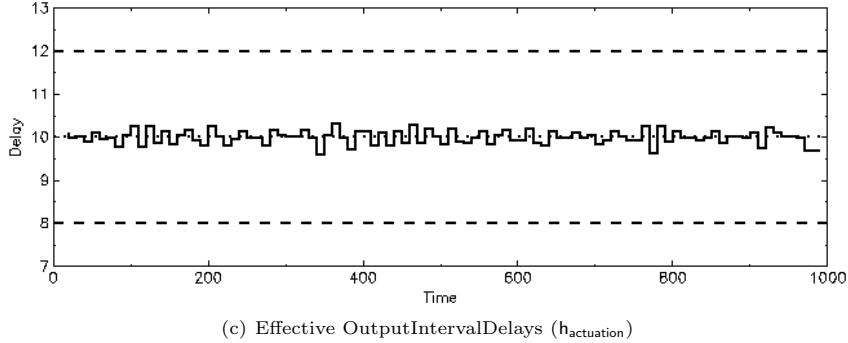


Figure 7.30: Configuration 1: Timing Oscilloscope Diagrams

constantly 10ms, the InputIntervalDelayRequirement is thus also satisfied. The OutputIntervalDelays are also varying randomly, however, also within the allowed tolerance bound. The OutputIntervalDelayRequirement is thus satisfied.

7.5.3 Configuration 2: Implicit Sender/Receiver communication, Multi-Rate Time-Triggered Activation, Multiple OS-Tasks

In the second configuration, the RunnableEntities also communicate with each other according to the implicit Sender/Receiver communication pattern. However, the TimingEvents that activate the RunnableEntities have different periods. Furthermore, the RunnableEntities are assigned to three different OS-tasks. Table 7.2 provides an overview of the configuration details where the differences to the previous configuration are highlighted.

RunnableEntity	SensorRE	ControllerRE	ActuatorRE
Triggering	TimingEvent (2ms)	TimingEvent (10ms)	TimingEvent (5ms)
Communication Patterns	implicit read implicit write	implicit read implicit write	implicit read implicit write
OS-Task (Priority)	TaskA (3)	TaskB (2)	TaskC (1)
Position in OS-Task	1	1	1

Table 7.2: Configuration 2: Implicit Sender/Receiver communication, multi-rate triggering, multiple OS-tasks

Note that the priorities of the OS-tasks are chosen such that the RunnableEntities that are assigned to the OS-tasks are activated in an order that is compliant with the PathSpecification. I.e., TaskA with the SensorRE is prioritized over TaskB with the ControllerRE which in turn is prioritized over TaskC with the ActuatorRE such

7.5 Analysis of the Example AUTOSAR System

that they are executed in this order even when the OS-tasks are all activated at the same time.

Due to the configured implicit Sender/Receiver communication and the assignment of RunnableEntities to three distinct OS-tasks, the logical PathSpecification from figure 7.28 needs to be augmented to a concrete flat PathSpecification as shown in figure 7.31.

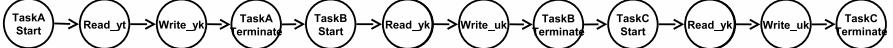
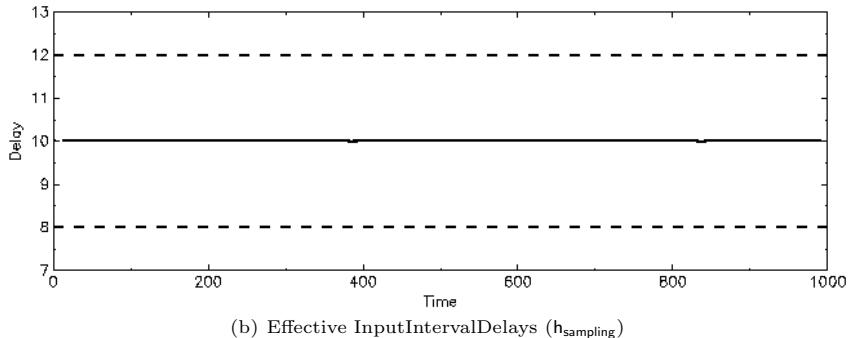
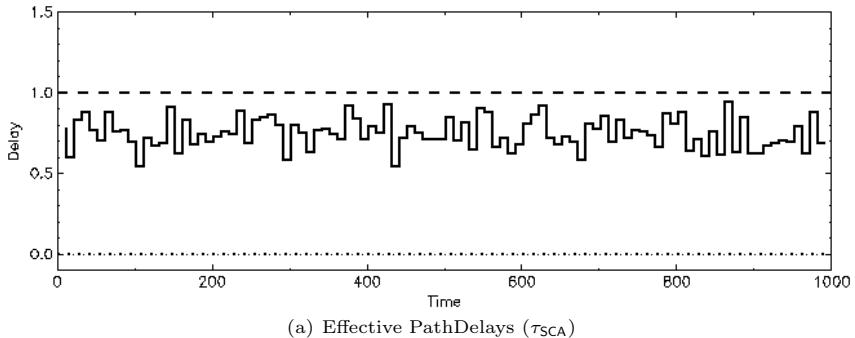


Figure 7.31: Order of AUTOSAR-specific Observable for configuration 2

The start and termination of all three OS-tasks need to be observed before and after the implicit read and write actions performed by the RunnableEntities. Through this, the copying of data between the RTE and the buffers of the OS-tasks is observed and correctly handled during path analysis.

Figure 7.32 shows the Timing Oscilloscope Diagrams for the effective PathDelays, InputIntervalDelays and OutputIntervalDelays.



7 Analysis of Event Traces from Monitoring and Simulation

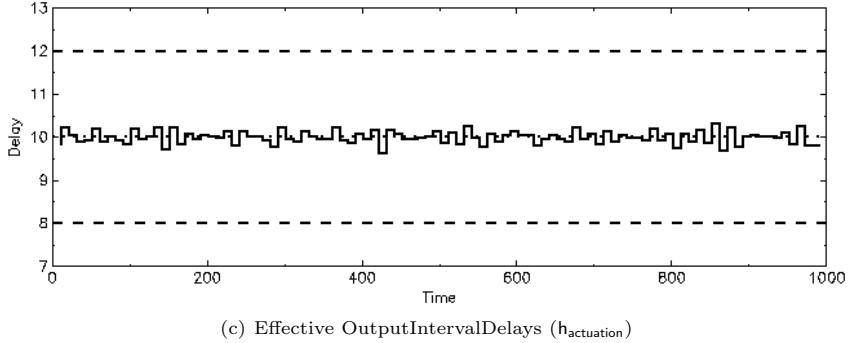


Figure 7.32: Configuration 2: Timing Oscilloscope Diagrams

Again, the PathDelays are varying randomly, however, this time, the PathDelays are larger compared to the first configuration. The rationale for this are that in this configuration, three OS-tasks are managed by the OS compared to one OS-task as in the first configuration. This implies additional overheads by the operating system for the management of OS-tasks. Furthermore, due to the employed implicit Sender/Receiver communication, the necessary copying processes between the RTE and the task buffers consume additional time. Despite of the larger PathDelays, the PathDelayRequirement is still satisfied. The InputIntervalDelayRequirement is almost constantly 10ms such that the InputIntervalDelayRequirement is satisfied. Although the OutputIntervalDelays are varying, the OutputIntervalDelayRequirement is still satisfied as the OutputIntervalDelays are within the specified tolerance bound.

7.5.4 Configuration 3: Explicit Sender/Receiver communication, Multi-Rate Time-Triggered Activation, Multiple OS-Tasks

In the third configuration, the RunnableEntities communicate with each other according to the explicit Sender/Receiver communication pattern. As in the second configuration, the TimingEvents that activate the RunnableEntities have different periods, and the RunnableEntities are assigned to three different OS-tasks (TaskA, TaskB, TaskC). Table 7.3 provides an overview of the configuration details.

RunnableEntity	SensorRE	ControllerRE	ActuatorRE
Triggering	TimingEvent (2ms)	TimingEvent (10ms)	TimingEvent (5ms)
Communication Patterns	explicit read explicit write	explicit read explicit write	explicit read explicit write
OS-Task (Priority)	TaskA (3)	TaskB (2)	TaskC (1)
Position in OS-Task	1	1	1

Table 7.3: Configuration 3: Explicit Sender/Receiver communication, multi-rate triggering, multiple OS-tasks

Due to the configured explicit Sender/Receiver communication, the logical PathSpecification from figure 7.28 does not need to be augmented such that the order of AUTOSAR-specific event and action classes in the concrete PathSpecification is the same as given in the logical PathSpecification.

Figure 7.33 shows the Timing Oscilloscope Diagrams for the effective PathDelays, InputIntervalDelays and OutputIntervalDelays.

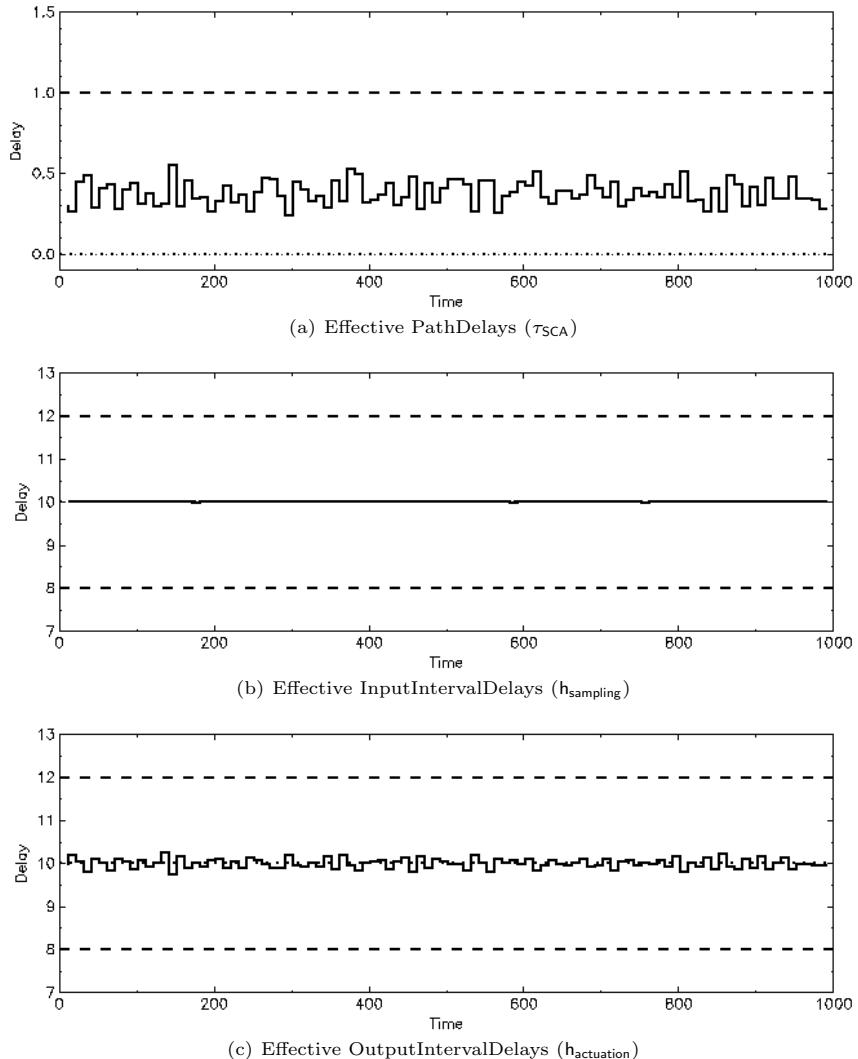


Figure 7.33: Configuration 3: Timing Oscilloscope Diagrams

7 Analysis of Event Traces from Monitoring and Simulation

The PathDelays are varying, but the PathDelayRequirement is always satisfied. Compared to the configurations with implicit Sender/Receiver communication, the effective PathDelays are smaller. This is due to the direct communication via the RTE rather than the indirect communication via the task buffers. The Input- and OutputIntervalDelayRequirements are both satisfied whereby the InputInterval- Delays are almost constant again and the OutputIntervalDelays are varying within the acceptable tolerance bound.

7.5.5 Configuration 4: Synchronization of Sampling Actions of two Sensors

As described in the foundations of control theory (see chapter 4), when multiple input or output devices are employed such as in multiple-input-multiple-output (MIMO) control applications, additional timing requirements are present that specify that the input data acquisition actions of multiple sensors or the output data effectuation actions by multiple actuators must be synchronized within a specific interval bound.

Figure 7.35 shows the relevant excerpt of the example AUTOSAR system of the MIMO control application and the event recording mechanism used for monitoring experiments. The RunnableEntities of the two SensorSWCs independently acquire input data from the same physical plant. This input data is merged into a single input data by the RunnableEntity of the VoterSWC. The InputSynchronizationRequirement demands that the two input data acquisition actions, `Read_yt`, performed by the two SensorREs occur within an interval of 2 ms measured from the start of the read actions to the end of the write action `Write_yk` performed by the VoterRE.

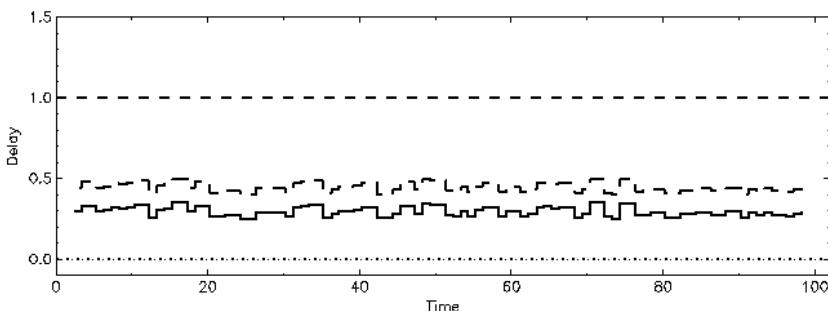


Figure 7.34: Configuration 4: Timing Oscilloscope Diagram

Figure 7.34 shows the Timing Oscilloscope Diagram for the InputSynchronizationIntervals. The upper dashed line denotes the size of the interval within

7.5 Analysis of the Example AUTOSAR System

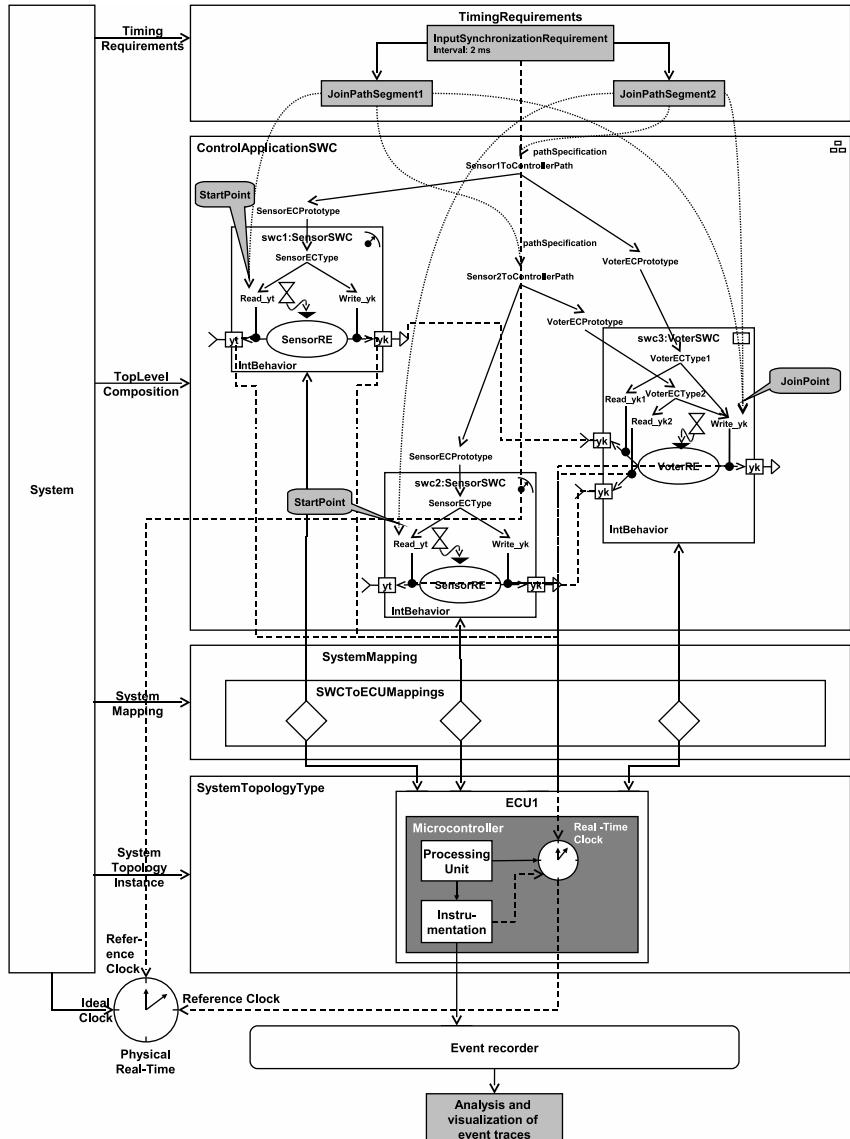


Figure 7.35: Excerpt of the example AUTOSAR system of the MIMO control application used for monitoring experiments

the InputSynchronizationIntervals must lie. As the lines for the two InputSynchronizationIntervals also closely follow each other, and as the size of the InputSynchronizationIntervals is always less than 2 ms, the two read actions `Read_yt` of the two SensorREs are adequately synchronized with write action `Write_yk` by the VoterRE.

7.5.6 Configuration 5: Clock Drift

If multiple real-time clocks are employed in an AUTOSAR system to which the TimingEvents that activate the RunnableEntities are bound, clock drift effects can occur in the case when no clock synchronization mechanism is employed. This consequently also has an effect on the timing properties. The latter can be analyzed by means of monitoring or simulation experiments where the event trace data is analyzed and visualized by means of the runtime-oriented Timing Oscilloscope Diagrams.

Due to the fact that the employed tools for monitoring and simulation, RTA-TRACE and chronSim, were not yet capable of capturing event traces from a distributed system (RTA-TRACE) or simulating a distributed AUTOSAR system (chronSim), it was not possible to research clock drift effects based on a real microcontroller implementation of a distributed AUTOSAR system. Clock drift effects, however, can also be simulated if some adequate assumptions are being made. For the analysis of such effects, it is assumed that the ActuatorRE is triggered by a TimingEvent that has a marginally greater period than the TimingEvents that trigger the SensorRE and ControllerRE. Through this, the ActuatorRE performs the reading and writing actions less frequent as if the real-time clock to which the TimingEvent triggering the ActuatorRE is slower than the real-time clock triggering the SensorRE and ControllerRE.

Table 7.4 provides an overview on the configuration details.

RunnableEntity	SensorRE	ControllerRE	ActuatorRE
Triggering	TimingEvent (10ms)	TimingEvent (10ms)	TimingEvent (10.5ms)
Communication Patterns	explicit read explicit write	explicit read explicit write	explicit read explicit write
OS-Task (Priority) Position in OS-Task	TaskA (3) 1	TaskB (2) 1	TaskC (1) 1

Table 7.4: Configuration 5: Explicit Sender/Receiver communication, multiple OS-tasks, clock drift

Figure 7.36 shows an event trace scenario that has been obtained from a simple discrete-event simulation. As can be seen from the event trace scenario, the explicit read and write actions by the ActuatorRE are performed less frequent than the read and write actions performed by the SensorRE and ControllerRE. Through this, the effective PathDelays get larger and larger.

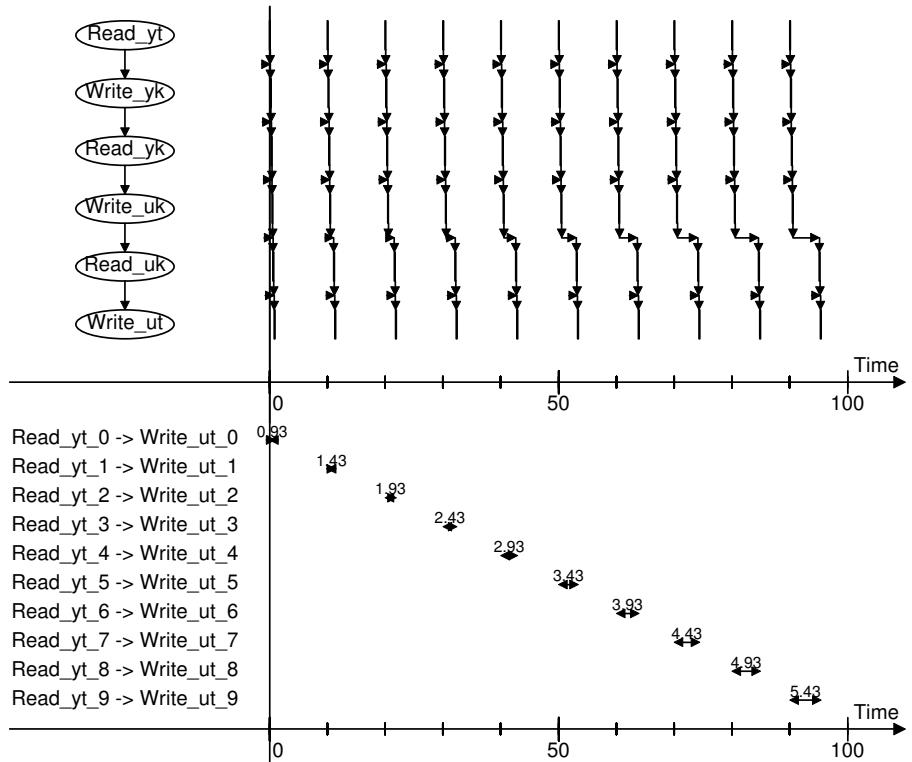


Figure 7.36: Event trace scenario for clock drift effects

Figure 7.37 shows the Timing Oscilloscope Diagrams for the effective PathDelays, InputIntervalDelays and OutputIntervalDelays. Note that a larger period of time (1000 ms) is shown as compared to the event trace scenario (100 ms).

The staircase-pattern that is present in the Timing Oscilloscope Diagram for the effective PathDelays is characteristic for clock drift effects. By means of Timing Oscilloscope Diagrams, clock drift effects can thus be detected such that counter measures such as a synchronization mechanism with adequate precision can be taken. Besides the staircase pattern, another characteristic effect is the regular peak that occurs in the Timing Oscilloscope Diagram for the effective InputIntervalDelays (effective sampling rate $h_{sampling}$). This peak is due to the fact that the clock drift leads to a wrap around in the staircase pattern as the TimingEvents that trigger the SensorRE and ControllerRE overtake the TimingEvent that triggers the ActuatorRE. Together with the staircase pattern in the Timing Oscilloscope Diagram for the

7 Analysis of Event Traces from Monitoring and Simulation

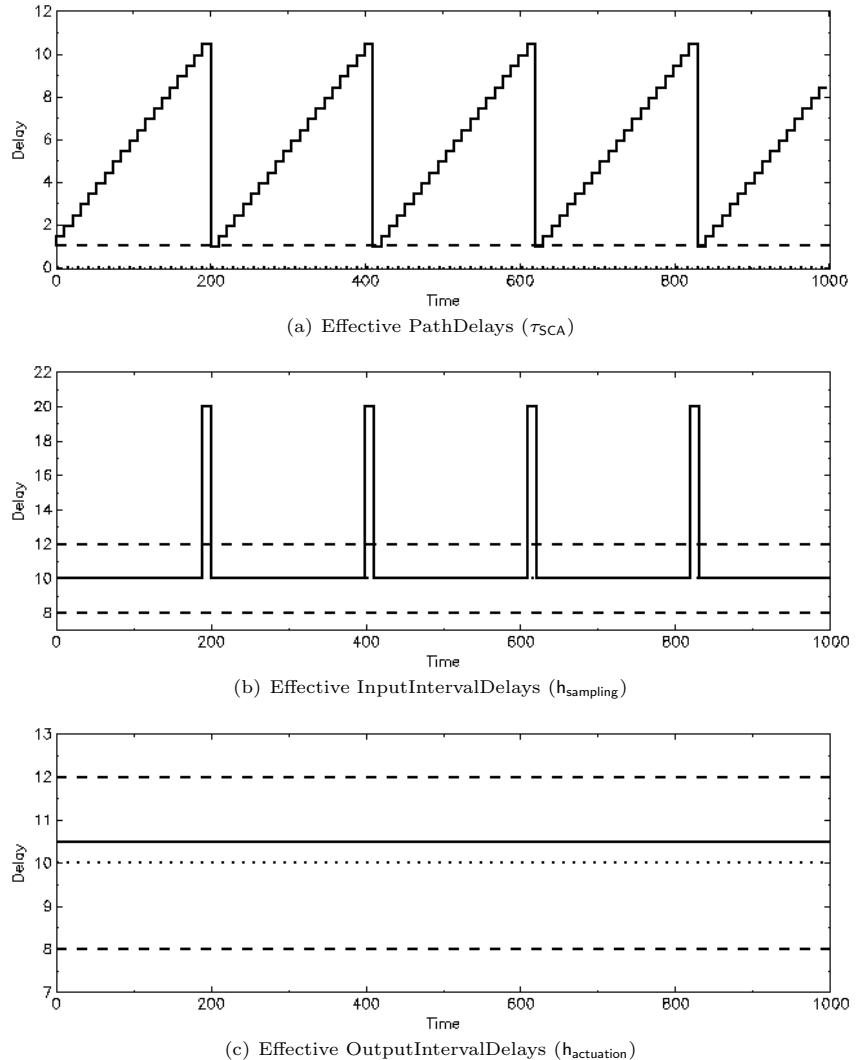


Figure 7.37: Configuration 5: Timing Oscilloscope Diagrams

effective PathDelays, such peaks give an even stronger indication that clock drift effects are present.

8 Case Study: Engine Control Application

8.1 Introduction

In order to demonstrate the applicability of the developed concepts of the Timing Model for AUTOSAR for describing signal, expressing timing requirements and determining timing properties by means of the monitoring-based approach, a case study of an integrated, rigorous and close to real-world example has been conducted. The case study, an engine control application, stems from a legacy demonstration project conducted at ETAS GmbH (ETAS DemoCar project [22], [38]) where the functionalities of an engine control application have been developed with the help of ETAS tools. For the case study, it has been re-engineered to an AUTOSAR-compliant ECU system [59].

The engine control application contains several real-time functionalities for which timing requirements can be specified. These must be satisfied by a microcontroller-based realization of the engine control application in terms of an AUTOSAR ECU system. The objective of the case study is to formally express these timing requirements based on precise signal path specifications and to evaluate their degree of fulfillment by means of monitoring experiments.

In this chapter, due to the complexity and size of the engine control application, only a specific excerpt of the complete engine control application, the air system, is presented. A detailed description of the complete case study can be found in [34].

8.2 Internal Combustion Engines and Tasks of an Engine Control Application

In internal combustion engines, chemical energy provided in the form of liquid fuel is transformed into heat and mechanical energy by means of combustion. The combustion is termed internal as it takes place within a combustion chamber. The mechanical energy is produced at the crankshaft of the engine and then transferred via the drive-train to the wheels where it drives the vehicle. The basic working principle of a four-stroke gasoline engine as considered in the case study can be found in standard literature on internal combustion engines, e.g. [42] or [71]. A brief description of the relevant aspects can also be found in [34].

Figure F.1 in Appendix F provides an overview on the mechanical, mechatronic and electronic components of the engine and the engine control application considered in the case study.

The task of an engine control application for an internal combustion engine is to determine important parameters for the control of the combustion processes in the

8 Case Study: Engine Control Application

cylinders such that the chemical energy provided by the fuel is optimally transformed into mechanical energy that drives a vehicle. Important parameters for a combustion process are the injection time, i.e. the time the injector valve shall open to inject fuel in the intake manifold, and the ignition time, i.e. the point in time when a spark should be produced to ignite the air/fuel mixture in the cylinder. The amount of fuel to be injected for a single combustion process depends on the load of air that is available for the combustion process. It must be dimensioned in such a way that the stoichiometric ratio of the air/fuel mixture is maintained ($\lambda = 1$), meaning that the fuel is completely and efficiently burned. In the engine for which the engine control application was designed, the amount of fuel that is to be injected for a combustion process depends on the load of air that is available for the combustion process. The latter is determined by the airflow in the intake system. The airflow is governed by a throttle device whereby the throttle position is influenced by the driver via the accelerator pedal. Due to its physical dynamics, the throttle position must be also controlled, e.g. when the engine is idling in order to prevent it from stalling. In the excerpt of the case study presented in the thesis, the functionality of the air system which controls the position of the throttle based on the driver's wish according to the accelerator pedal is considered as an example functionality towards that application-specific timing requirements can be formulated and for which the degree of fulfillment is to be evaluated.

Figure 8.1 depicts a schematic overview of the engine control application and the relevant input and output signals from the context it is embedded in (driver, vehicle including the engine, vehicle environment).

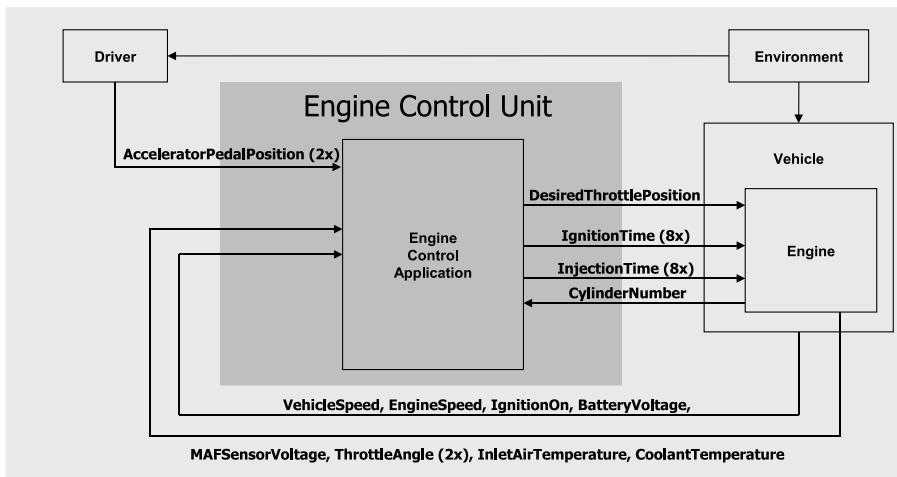


Figure 8.1: Schematic overview of engine control application with relevant input and output signals

8.3 Overview of the Basic Functionalities

The relevant mechatronic components are thus the throttle sensors and actuator as well as the accelerator pedal sensors¹. The corresponding input and output signals are the AcceleratorPedalPositions and the ThrottleAngles acquired by the respective sensors and the DesiredThrottlePosition effectuated by the throttle actuator.

8.3 Overview of the Basic Functionalities

Figure 8.2 shows an overview of the internal structure of the engine control application realizing its basic functionalities.

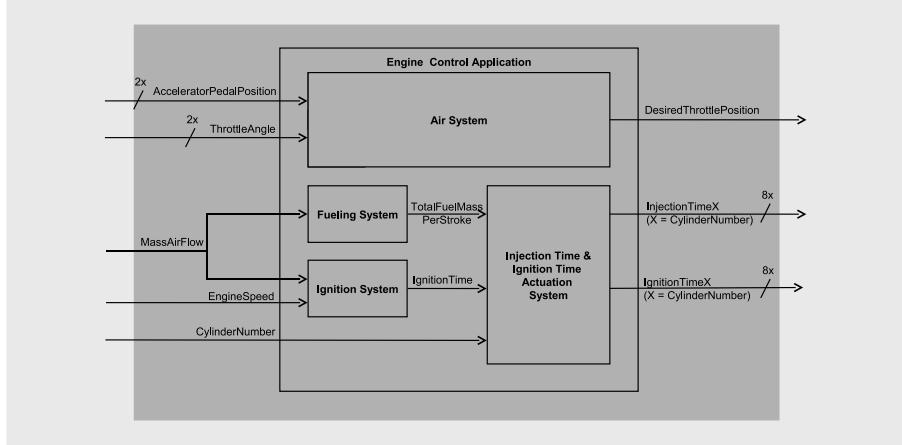


Figure 8.2: Overview of the internal structure of the engine control application and its basic functionalities

The overall functionality of the engine control application is divided into four subsystems:

Air system The air system calculates the desired throttle position based on the current throttle position and the accelerator pedal position. This influences the air flow and subsequently the injected fuel mass and ignition timing for a combustion process.

Fueling system The fueling system calculates the fuel mass to be injected in the intake manifold for a combustion process in a cylinder. This is based on the current mass air flow sensed in the intake system.

Ignition system The ignition system calculates the ignition angle that determines the point in time when an ignition spark is produced to ignite the air/fuel

¹Note that the throttle and accelerator pedal sensors are duplicated due to safety considerations.

8 Case Study: Engine Control Application

mixture in the combustion chamber. This is based on the mass air flow and the current engine speed.

Injection time and ignition time actuation system The injection time and ignition time actuation system delivers the latest values of the two parameters upon a cylinder specific request. The calculation of the two parameters is decoupled from the actuation as the parameters are pre-calculated for all cylinders (sequential injection). This allows a fast response to a cylinder-specific request.

In the case study engine control application, specific functionalities that were traditionally realized as purely mechanical systems are realized as mechatronical systems, i.e., as mechanical system additionally comprising electronics and software. The legislative demands of the E-Gas concept² [1] require the introduction of certain redundancy concepts due to safety considerations when mechanical components are replaced by electronics-based solutions. Such concepts include to redundantly acquire input signals by means of multiple sensors and to compute a voted signal for further processing from the redundantly acquired sensor values. In figure 8.2, the input signals for the accelerator pedal position and the throttle position are thus duplicated.

8.4 Air System

8.4.1 Structure

As shown in figure 8.3, the air system is subdivided into several elementary functions that

- capture input signals from the accelerator pedal and throttle sensors (functions `AcceleratorPedalSensor` and `ThrottleSensor`),
- analyze redundantly captured input values from the sensors and compute a merged input signal (functions `AcceleratorPedalVoter` and `ThrottleSensor`)
- analyze the merged accelerator pedal position input signal and compute a desired throttle position set-point signal (function `PedalFeel`)
- compute a control signal for the adjustment of the current throttle position based on the desired throttle position (function `ThrottleController`)
- bring the computed desired throttle position into effect (function `ThrottleActuator`)

The mechanic part of the throttle device installed in the intake system of the engine is a dynamic system that needs to be controlled. Together with the throttle installed in the engine, the air system thus forms a closed-loop control application.

²The E-Gas concept is an automotive-specific regulation that is compulsory for the electronic connection of the accelerator pedal to the throttle.

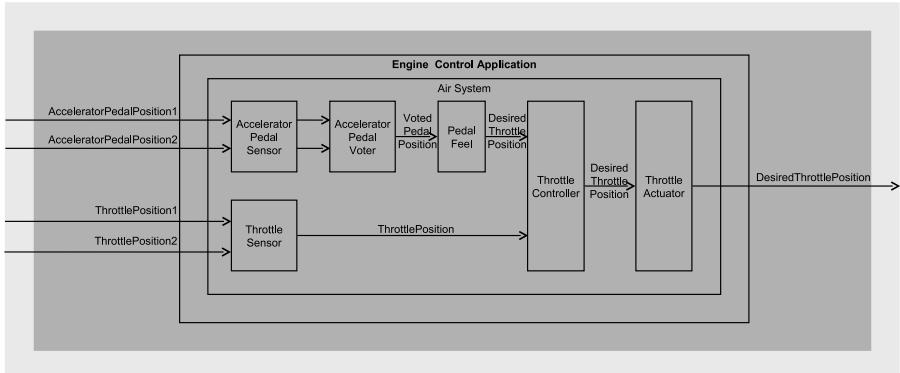


Figure 8.3: Overview on the internal structure of the air system

8.4.2 Signal Paths

The air system contains several signal paths between its input signals and output signals. In principle, two conceptual signal paths can be identified:

- The first conceptual signal path is from the input signal `AcceleratorPedalPosition` to the output signal `DesiredThrottlePosition`. This signal path describes the chain of cause-and-effect on how a change of the accelerator pedal position, i.e. the setpoint determined by the driver, influences the new throttle position.
- The second conceptual signal path is from the input signal `ThrottleSensor` to the output signal `DesiredThrottlePosition`. This signal path describes the chain of cause-and-effect that corresponds to the feedback path of the throttle control application.

Due to the required redundancy concepts, the accelerator pedal position and the current throttle position are sensed by duplicate sensors and delivered as separate input signals. There are thus four concrete signal paths where each two signal paths correspond to one conceptual signal path.

The focus of the excerpt of the case study presented in the thesis is on the signal paths from the two throttle sensors to the throttle actuator. These correspond to the feedback path of the throttle control application and can best be associated with application-specific timing requirements.

8.4.3 Timing Requirements

Each of the identified signal paths of the air system is associated with three different timing requirements. These originate from the analysis of the dynamic behavior of the throttle device such that a control application can be built that adequately influences its behavior. Note that in the thesis we only focus on the feedback path of

8 Case Study: Engine Control Application

the throttle control application, i.e. the signal path from the input signal delivered by the throttle sensor to the output signal of the throttle actuator, and its associated timing requirements. The timing requirements can be formulated as follows:

1. The first timing requirement is a timing requirement on the latency of the signal transformation along the feedback path of the throttle control application. A minimization of the path delay τ is required such that it can be considered as being negligible. This means that the feedback path delay should ideally be 0 ms whereby a deviation of 1 ms is acceptable. This translates to the requirement that the effective path delays must be less than or equal to 1 ms. Formally, this is expressed by

$$\begin{aligned}\tau_{\text{ThrottlePosition1} \rightarrow \text{DesiredThrottlePosition}} &\leq 1 \text{ ms} \\ \tau_{\text{ThrottlePosition2} \rightarrow \text{DesiredThrottlePosition}} &\leq 1 \text{ ms}\end{aligned}$$

- 2.+3. The second and third timing requirement are timing requirements on the latency between consecutive effective sampling and actuation actions. The throttle control application requires the maintenance of a nominal effective sampling interval of 10 ms. A deviation of 1 ms is acceptable (acceptable effective sampling jitter). The same also holds for the effective actuation interval (acceptable effective actuation jitter). These timing requirements are expressed by

$$\begin{aligned}h_{\text{ThrottlePosition1} \rightarrow \text{DesiredThrottlePosition}}^{\text{sampling}} &= 10 \text{ ms} \pm 1 \text{ ms} \\ h_{\text{ThrottlePosition2} \rightarrow \text{DesiredThrottlePosition}}^{\text{sampling}} &= 10 \text{ ms} \pm 1 \text{ ms}\end{aligned}$$

for the effective sampling intervals, and

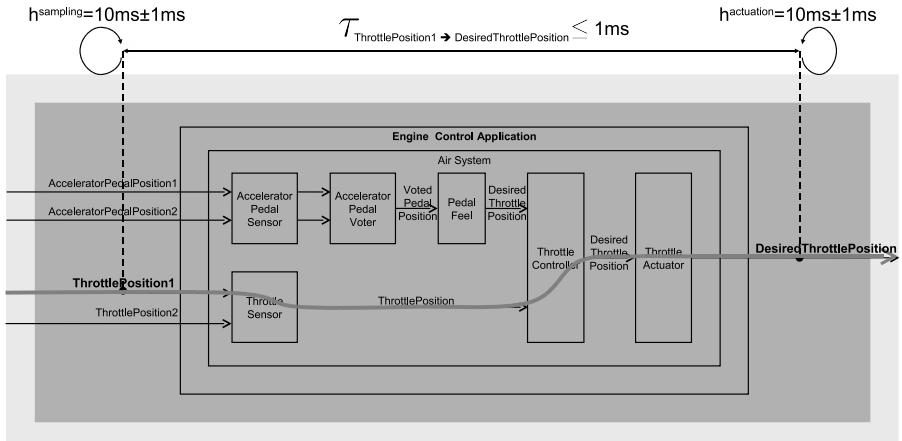
$$\begin{aligned}h_{\text{ThrottlePosition1} \rightarrow \text{DesiredThrottlePosition}}^{\text{actuation}} &= 10 \text{ ms} \pm 1 \text{ ms} \\ h_{\text{ThrottlePosition2} \rightarrow \text{DesiredThrottlePosition}}^{\text{actuation}} &= 10 \text{ ms} \pm 1 \text{ ms}\end{aligned}$$

for the effective actuation intervals.

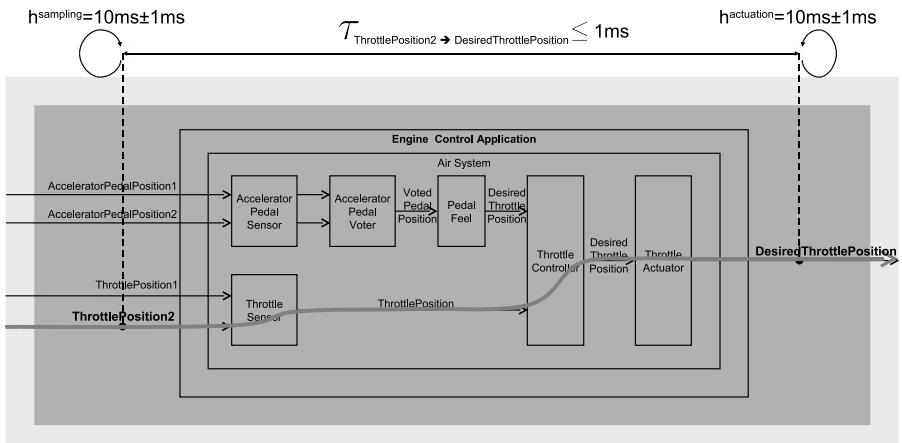
Figures 8.4(a) and 8.4(b) show the signal paths from the input signals ThrottlePosition1 and ThrottlePosition2, respectively, to the output signal DesiredThrottlePosition and the associated timing requirements.

Due to the employed redundancy concept, timing requirements on the synchronization of related input signals also need to be formulated.

The throttle position is provided as two signals from the two redundant throttle sensors. The conversion of the voltage values delivered by the sensors to a percentage value and the determination of a voted throttle position signal is performed by a single function (function `ThrottleSensor`). In order to produce such a voted signal, the two input signals must be synchronized within an interval of 1 ms with respect



- (a) Signal path from input signal ThrottlePosition1 to output signal DesiredThrottlePosition and associated timing requirements



- (b) Signal path from input signal ThrottlePosition2 to output signal DesiredThrottlePosition and associated timing requirements

Figure 8.4: Signal paths from input signals ThrottlePosition1/2 to output signal DesiredThrottlePosition and associated timing requirements

to the voted signal. This is expressed by

$$d_{ThrottlePosition1 \rightarrow ThrottlePosition} \leq 1 \text{ ms}$$

$$d_{ThrottlePosition2 \rightarrow ThrottlePosition} \leq 1 \text{ ms}$$

8 Case Study: Engine Control Application

Figure 8.5 depicts the relevant segments of the signal paths from input signals ThrottlePosition1 and ThrottlePosition2 to the common intermediate signal ThrottlePosition where both signal paths join. The timing requirement that is associated with these is also shown.

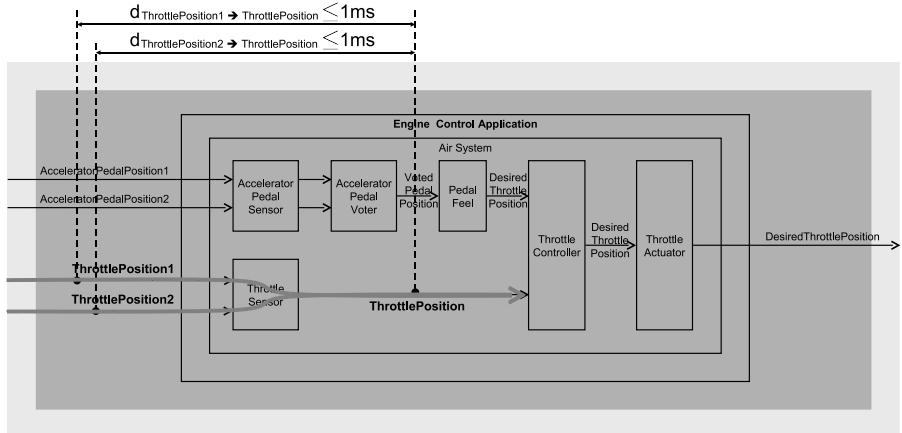


Figure 8.5: Signal paths segments from ThrottlePosition1 and ThrottlePosition2 to ThrottlePosition and associated timing requirements

8.5 AUTOSAR System

The legacy software of the engine control application stemming from the DemoCar project has been re-engineered into an AUTOSAR-compliant ECU system and realized on a hardware platform with an Infineon TriCore1796 microcontroller ([59], [36], [37], [35]). For the individual functions that constitute the engine control application, software components have been developed which have then been integrated into a complete AUTOSAR software architecture. Several design decisions have been taken come to an efficient AUTOSAR software architecture:

- Each function is realized by a single AtomicSoftwareComponent that contains a single RunnableEntity in its InternalBehavior. When being triggered by an RTEEvent, the RunnableEntity first reads the values of the DataElementPrototypes in the RPorts, performs a data transformation of this input data into output data, and writes the output data on the DataElementPrototypes in the PPorts.
- Implicit Sender/Receiver communication is employed by those software components that realize the air system, the fueling system and the ignition timing system.

8.6 Specification of Timing Requirements based on Formal Path Specifications

- No Client/Server communication is employed as the engine control application does not contain timing critical parts that require service-oriented communication.
- The AtomicSoftwareComponentTypes that realize the individual functions are all instantiated only once in the logical software architecture of the AUTOSAR system.

Figure 8.6 shows an overview of the AUTOSAR system realizing the engine control application. Note that only those software components are shown that constitute the basic functionalities of the engine control application, i.e. the air system, the fueling system, the ignition timing system, and the injection time and ignition time actuation system. In the following, only the relevant excerpt for the air system is further discussed and described.

8.6 Specification of Timing Requirements based on Formal Path Specifications

To adequately express the application-specific timing requirements of the throttle control application, the respective signal paths are modeled by means of hierarchical event chains that denote logical PathSpecifications. For this, the reading and writing actions performed by the RunnableEntities are marked with suitable RTEAPIActions for implicit Sender/Receiver communication which are then first aggregated to atomic event chains and then to composite event chains. Figure 8.7 shows the relevant excerpt of the air system for AUTOSAR system of the engine control application. The feedback path from the sensor input signal `ThrottleSensor1Voltage` to the actuator output signal `DesiredThrottlePosition` is described by a logical PathSpecification which in turn is associated with the timing requirements on the effective feedback path delay, sampling and actuation rates. Note that the second feedback path, i.e. from the sensor input signal `ThrottleSensor2Voltage` to the actuator output signal `DesiredThrottlePosition`, is in principle described analogously and is associated with the same timing requirements.

In order to express the timing requirements on synchronized sensor data acquisition, the relevant RTEAPIActions that are part of the two logical PathSpecifications need to be considered in relation to each other. These are the read actions performed by the RunnableEntity of the `ThrottleSensorSWC`. The common JoinPoint with respect to which the synchronization is being measured is the write action performed by the same RunnableEntity. Figure 8.8 shows the two logical PathSpecifications and the InputSynchronizationRequirement that is expressed based on them.

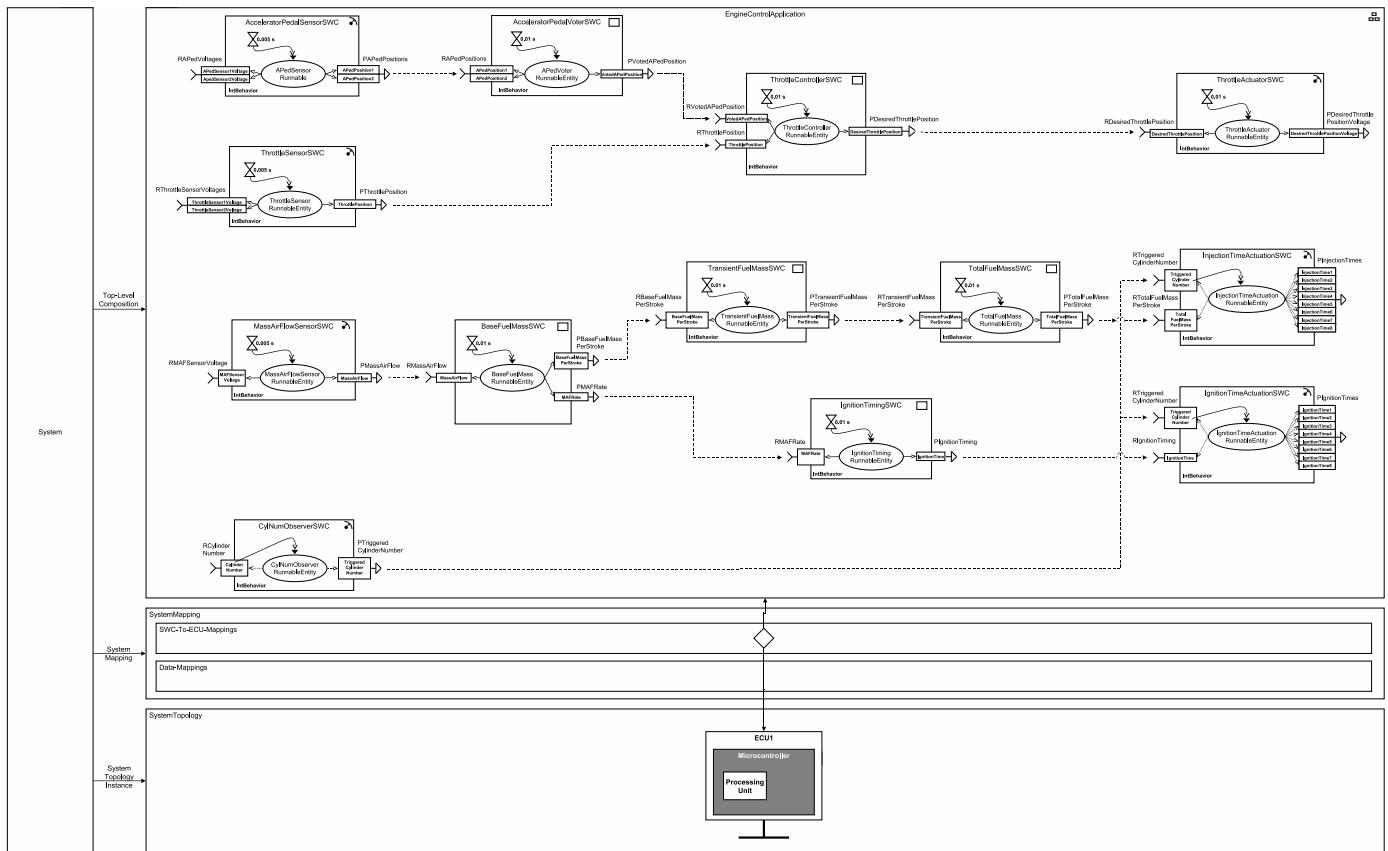


Figure 8.6: Overview of the AUTOSAR system for the engine control application

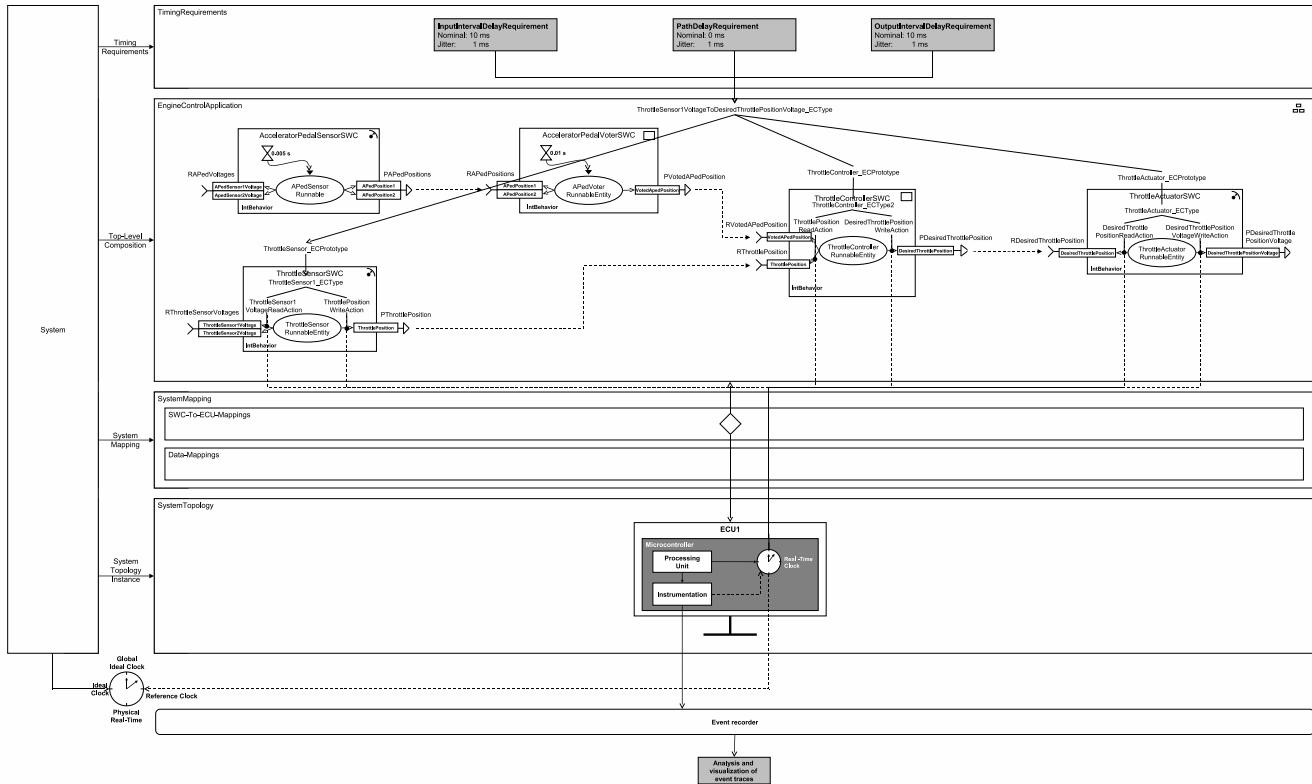


Figure 8.7: Specification of timing requirements on the effective feedback path delay, sampling and actuation rates based on a formal path specification of the feedback path

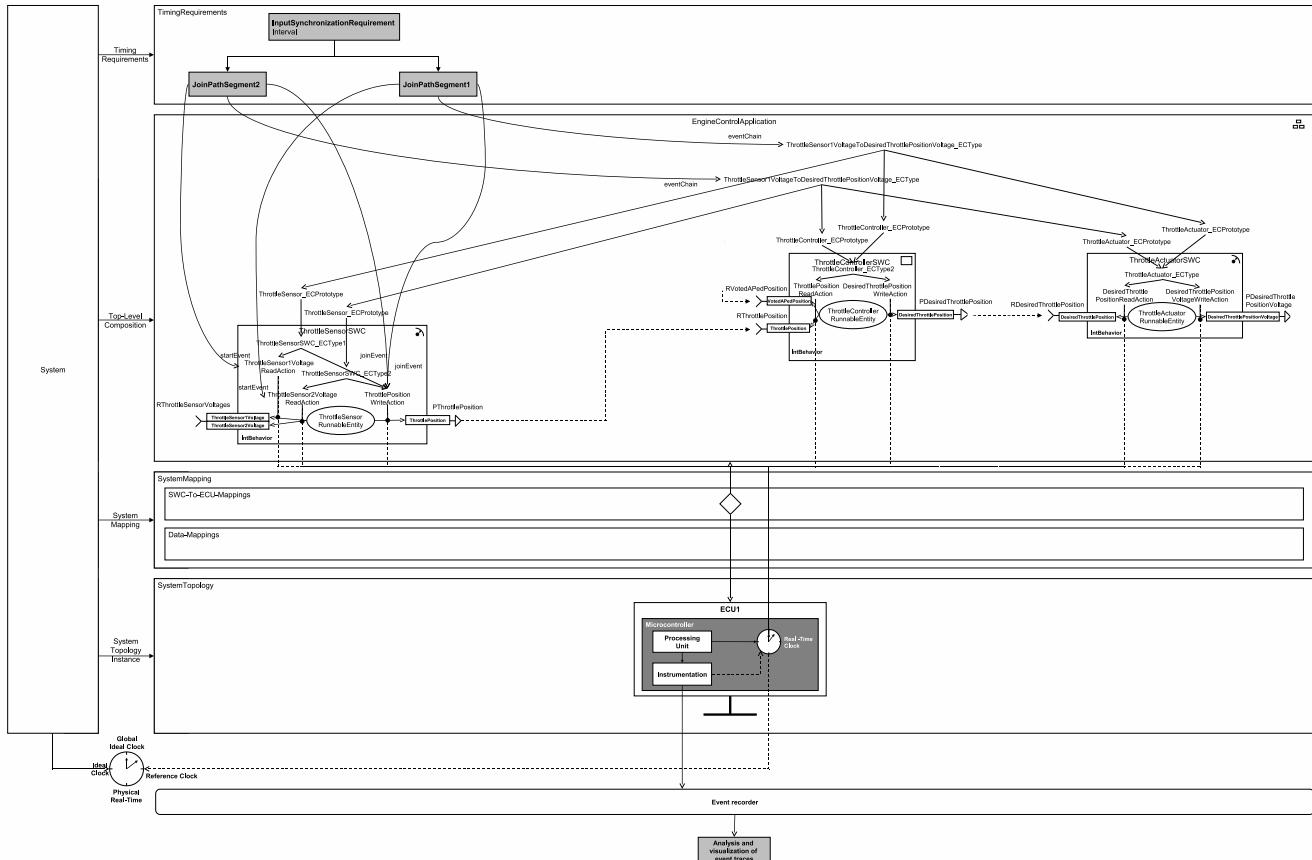


Figure 8.8: Specification of input synchronization timing requirement for synchronized throttle sensor data acquisition

8.7 Derivation of a Monitoring Instrumentation

From the logical PathSpecifications and the information from the system configuration and ECU configuration an instrumentation for monitoring experiments has been obtained.

As a first step, for each logical PathSpecification, the ordered set of RTEAPI-Actions is derived through the application of the flattening algorithm introduced in chapter 6.2. This corresponds to the flat logical PathSpecifications from which the flat concrete PathSpecifications are derived by taking the design decisions of the System Mapping and the ECU configuration into account. Figure 8.9(a) shows the flat logical PathSpecification for the feedback path of the throttle control application. The concrete flat PathSpecification is shown in figure 8.9(b).

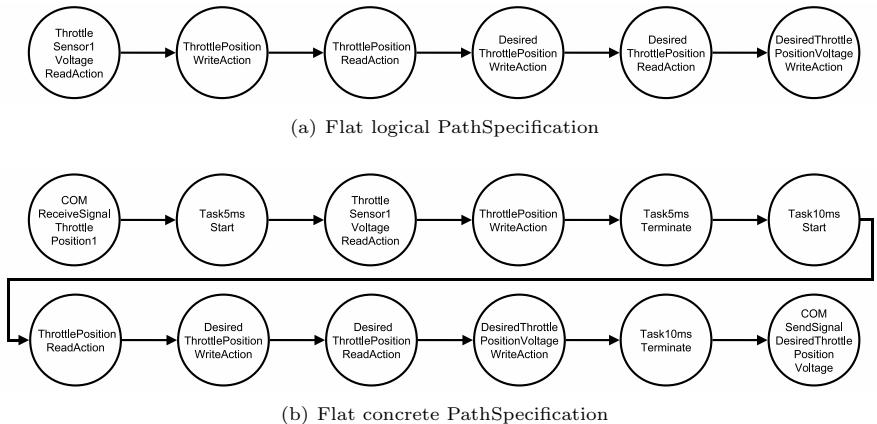


Figure 8.9: Flat logical and concrete PathSpecifications for the feedback path of the throttle control application

From the concrete flat PathSpecifications, an instrumentation for monitoring experiments has then been derived by implementing the corresponding Virtual Function Bus Trace hook functions with event logging functions provided by the RTA-TRACE tool. The source code constituting the monitoring instrumentation has been integrated into the extended ECU software build process in order to obtain a binary that can be used in monitoring experiments.

8.8 Evaluation of the Timing Requirements

For the determination of timing properties, a monitoring experiment has been conducted where an event trace has been recorded with the help of the instrumentation. During the monitoring experiment, the engine control application was operated at

a fixed engine speed of 2000 revolutions per minute (rpm). Through the fixed engine speed, dynamic effects from operating the engine at different speeds could be excluded as potential sources for variations of timing properties. Even trace data was recorded for a period of 800ms due to a limitation of the monitoring tool (RTA-TRACE). Based on the event trace data and the concrete flat PathSpecifications derived from the logical PathSpecifications, the effective PathInstances have been determined with the help of the path analysis algorithms. Based on these, the timing properties could be determined. These are visualized by means of Timing Oscilloscope Diagrams such that the degree of fulfillment of the timing properties could be evaluated.

8.8.1 Path Delays

Figure 8.10 depicts the Timing Oscilloscope Diagrams for the PathDelays of the two feedback paths of the throttle control application, i.e. the signal paths from the input signals ThrottlePosition1 and ThrottlePosition2, respectively, to the output signal DesiredThrottlePosition.

The PathDelays are relatively constant and only marginally vary. The rise of the PathDelays at the beginning is a consequence of the system startup of the AUTOSAR ECU system, more precisely the startup of the real-time operating system and the initialization of the application software. However, the PathDelayRequirement of the throttle control application is not satisfied as the size of the effective PathDelays for the feedback path of approximately 5 ms is too large compared to the demanded size for the effective PathDelays of being smaller than 1 ms. The rationale for this lies in the fact that input signals are provided over a CAN bus to the AUTOSAR ECU system realizing the engine control application. This, however, was required to perform in-the-loop monitoring experiments with a simulate plant for the driver, vehicle and environment as not real engine or vehicle could be used.

There are two alternative solutions for this problem:

- Instead of employing an asynchronous communication over a CAN network, a time-deterministic synchronous vehicle bus technology such as time-triggered CAN or Flexray could be employed. Through this, input signals would be provided at predefined points in time.
- As the feedback path delay is constant, it is possible to include the delay as a parameter in the design of the throttle control application such that it can adequately account for this through by compensating it. This means that the timing requirements need to be adapted, consequently demanding a nominal feedback path delay of approximately 5 ms and a certain tolerance bound.

8 Case Study: Engine Control Application

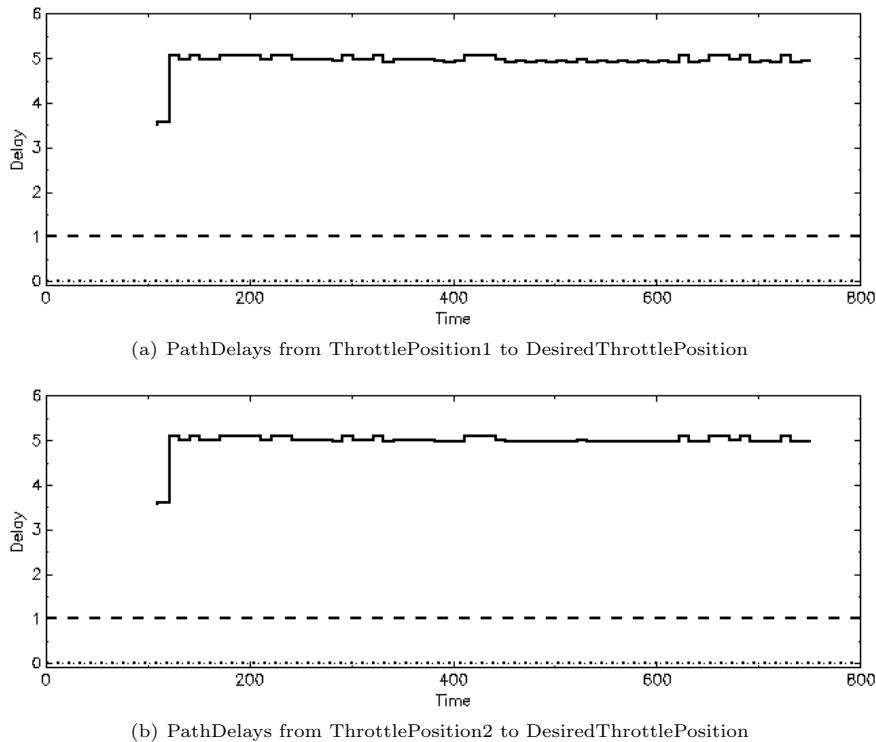


Figure 8.10: Timing Oscilloscope Diagrams for PathDelays

8.8.2 Input Interval Delays

The Timing Oscilloscope Diagrams with the InputIntervalDelays for the signal paths from ThrottlePosition1 and ThrottlePosition2 to DesiredThrottlePosition are shown in figure 8.11.

The InputIntervalDelays correspond to the effective sampling rates h_{sampling} by which input data is acquired from the two throttle sensors. As the sampling rates are almost constant and in almost all cases stay within the demanded tolerance bound, the InputIntervalDelayRequirements are satisfied. The temporary violation at the beginning is a result of the system startup behavior.

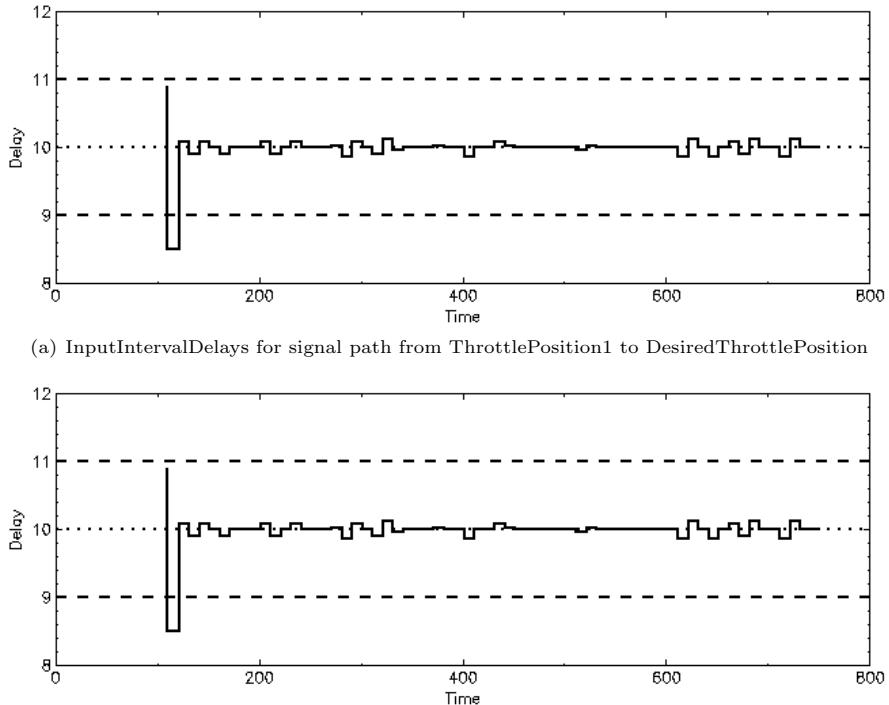


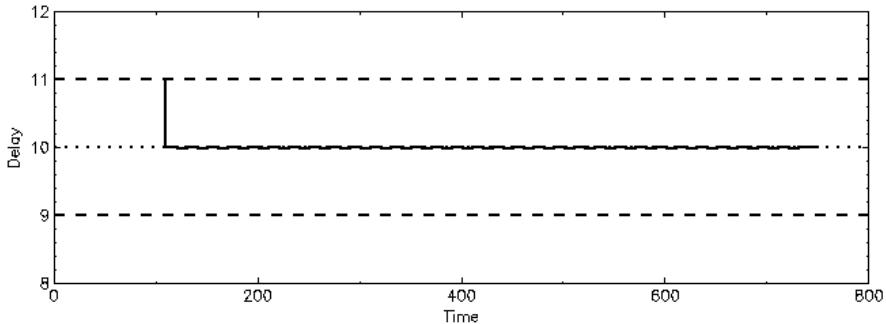
Figure 8.11: Timing Oscilloscope Diagrams for InputIntervalDelays

8.8.3 Output Interval Delays

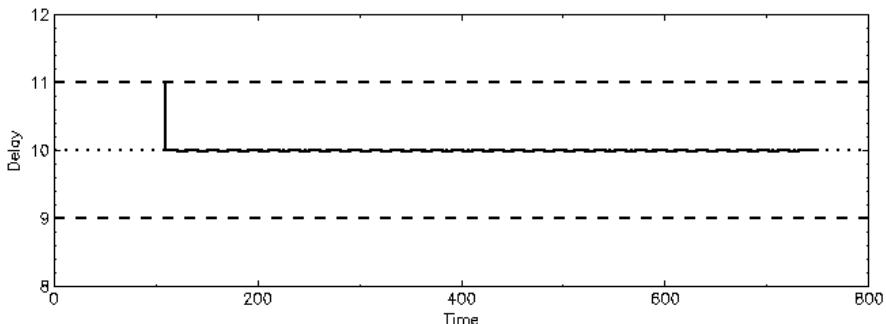
Similar to the previously shown InputIntervalDelays, figure 8.12 depicts the Timing Oscilloscope Diagrams with the OutputIntervalDelays for the signal paths from ThrottlePosition1 and ThrottlePosition2 to DesiredThrottlePosition.

The OutputIntervalDelays correspond to the effective actuation rate $h_{actuation}$ by which output data is effectuated by the throttle actuator. Similar to the sampling rates, the actuation rate is also almost constant and stays within the demanded tolerance bound. The OutputIntervalDelayRequirement is thus satisfied.

8 Case Study: Engine Control Application



(a) InputIntervalDelays for signal path from AcceleratorPedalPosition1 to DesiredThrottlePosition



(b) InputIntervalDelays for signal path from AcceleratorPedalPosition2 to DesiredThrottlePosition

Figure 8.12: Timing Oscilloscope Diagrams for OutputIntervalDelays

8.8.4 Input Synchronization Intervals

The Timing Oscilloscope Diagram for the InputSynchronizationIntervals that represent the effective synchronization between the two sensor data acquisition actions performed by the **ThrottleSensorRE** are shown in figure figure 8.13.

The two sensor data acquisition actions (in the form of the two read actions performed by the RunnableEntity of the **ThrottleSensorSWC**) are quite exactly synchronized. This can be concluded as the two lines for the InputIntervalDelays closely follow each other. However, the size of the IntervalDelays is too large. The rationale for this is the same as for the too large effective PathDelays for the feedback path. I.e., the asynchronous reception of input signals via the CAN network.

8.9 Summary and Conclusion

The results of the monitoring experiment have shown that not all timing requirements are satisfied. Most of the timing properties vary in an irregular pattern. This

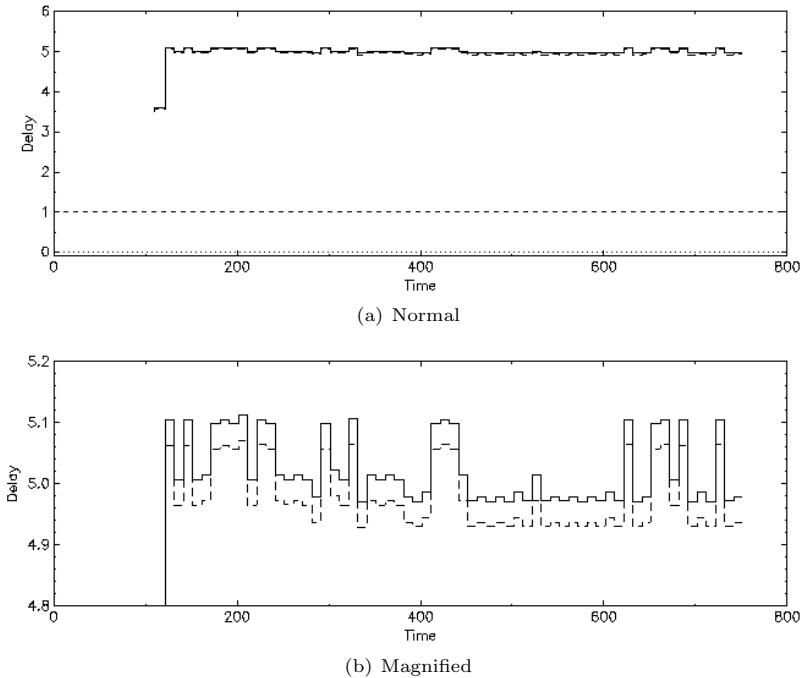


Figure 8.13: Timing Oscilloscope Diagrams for InputSynchronizationIntervals

can be seen in the respective Timing Oscilloscope Diagrams. The reason for the violation of the timing requirements lies in the hardware-in-the-loop (HiL) setup that was designed for the monitoring experiment, more precisely, the source for the timing problems lies in the realization of the CAN bus interface over which input and output signals are exchanged with the simulated engine environment. The reason is that the system signals in the CAN frames are sent asynchronously over the CAN bus by the real-time simulation hardware on which the environment model is executed. The effect is that some system signals arrive earlier than others at the engine control application such that input signals are read at different times. However, through the monitoring experiment and the analysis of the signal paths, these timing problems could be identified. A future improved realization of the engine control application in terms of an AUTOSAR system could employ a time-deterministic network protocol such as the Flexray protocol. Through this, the timing problems of the current setup could be avoided. Another solution would be to account for the non-zero, constant feedback path delay during the control application development.

9 Summary, Extensibility and Transferability

9.1 Summary

The Motivation for the work presented in this thesis has been that automotive embedded applications are real-time applications that have timing requirements which need to be satisfied such that the vehicle functions realized by the applications operate as expected. Violations of the timing requirements can lead to undesired behavior or even complete failure. In order to properly account for potential timing problems during the development and to identify the sources for the latter during the runtime of such systems, it is firstly required that the timing requirements are adequately described and secondly required that corresponding timing properties are determined such that the degree of fulfillment of the former can be evaluated.

Control Applications are inherently real-time applications where the timing requirements originate from discrete-time realizations of process descriptions formulated in the continuous-time domain. The timing requirements of closed-loop control applications refer to the feedback path that can be described between a sensor and an actuator, and which is internal to the control applications. These timing requirements demand that the feedback path delay (τ_{SCA}) is either negligibly small or maintained constant, and that the effective sampling and actuation intervals ($h_{sampling}, h_{actuation}$) are maintained constant whereby in all cases acceptable tolerances can further be formulated. If multiple sensors or actuators are employed, the related sampling actions at the sensors or the related actuation actions at the actuators must be synchronized.

Timing Models for simulation- and monitoring-based performance evaluation approaches are based on events that mark interesting places in a system that can be observed during the runtime of the system. Events provide an adequate abstraction of the system behavior such that timing properties can be determined. The causal and temporal order relations that are defined on sets of events provide the necessary basis for the determination of safe timing properties. In order to determine a valid temporal order on a set of events recorded from a parallel or distributed computer system obtained through simulation- or monitoring-based approaches, a formal notion of time is required. This is achieved through the employment of a formal clock model by which the behavior of any types of clocks can be precisely characterized based on an ideal clock. Events are then bound to formally characterized clocks such that the time-stamps of instances of clocks which are readings of the clocks can be unambiguously ordered according to the temporal order relation. This enables safe statements of causal and temporal relations among events that occurred in a parallel

9 Summary, Extensibility and Transferability

and distributed computer system and consequently the determination of valid and meaningful timing properties.

AUTOSAR is a joint initiative of the automotive industry targeting a rigorous development approach for automotive embedded systems through standardization of non-competitive concepts. While AUTOSAR provides all necessary concepts for realizing an automotive E/E system, it has so far not been possible to adequately describe the timing requirements of the real-time applications realized by the AUTOSAR system. While timing requirements of control applications must be described based on signal path descriptions, for the latter, AUTOSAR does not provide adequate concepts.

Overview of work In the work presented in this theses, the three aspects that have so far been only considered separately, i.e. (1) control applications as important class of real-time applications from which timing requirements originate from, (2) AUTOSAR providing standardized concepts for the realization of the latter in automotive electric/electronic systems, and (3) timing models based on events and which support performance evaluations by means of simulation- or monitoring-based approaches are brought together. The result is a Timing Model for AUTOSAR that allows the precise description of the application-specific timing requirements for control applications based on specifications of signal paths and the determination of suitable timing properties by means of simulation- or monitoring-based approaches such that the degree of fulfillment of the timing requirements can be evaluated. The link between the specification concepts for timing requirements and the validation concepts by means of timing properties is that event trace data based on which the timing properties are determined is obtained from instrumentations of the AUTOSAR ECU systems whereby the instrumentation is directly derived from the path specifications to which the timing requirements refer. Figure 9.1 recapitulates the overview figure of the work presented in this thesis.

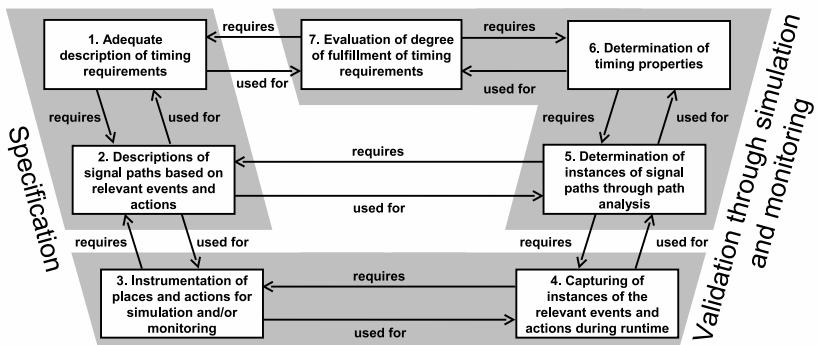


Figure 9.1: Overview of work

Description of timing requirements based on signal path specifications
In order to adequately describe application-specific timing requirements, a concept

is required for the description of signal paths to which the timing requirements refer. To address the problem of describing signal paths in hierarchical software architectures such as AUTOSAR, a concept that seamlessly integrates with the existing AUTOSAR concepts has been introduced in this thesis. The concept is based on the notion of event and action classes that can be identified and monitored in an AUTOSAR system and which are relevant for describing a signal path. As concrete event and action classes that can be marked in an AUTOSAR system, the trace events that have already informally been defined by AUTOSAR in the Virtual Function Bus Tracing Mechanism have been formally introduced into AUTOSAR. These AUTOSAR-specific event and action classes (RTEAPIActions, RunnableEntityEvents, OSEvents and COMEvents) have a direct counterpart in the source code of the instrumentation of an AUTOSAR ECU system.

In order to give instances of events and actions that occur during the runtime of an AUTOSAR system a definite temporal context, a formal notion of time has been introduced. This has been achieved by adapting the concepts of the formal clock model developed by Münzenberger et al. [6]. A globally visible ideal clock has been introduced to which event and action classes are bound during the planning phase (Virtual Function Bus view). In the realization phase (System view), the formal clock model allows the precise characterization of the behavior of the real-time clocks employed in the single ECUs of an AUTOSAR system. The event and action classes are then bound to the real-time clocks of the single ECUs such that is clear in which temporal contexts instances of event and action classes occur.

To describe a logical signal path within the software architecture of an AUTOSAR system, RTEAPIActions that mark the points of interaction of the application software and the Runtime Environment of an AUTOSAR ECU system are aggregated to hierarchical event chains. On the highest level of the software architecture of an AUTOSAR system, a hierarchical event chain denotes a precise and unambiguous specification of a signal path from a system input (sensor) to a system output (actuator) with RTEAPIActions residing on the leaf-level of the hierarchical event chain.

To express application-specific timing requirements, one or more logical signal path specifications can then be associated with parameters that adequately express the timing requirements. For the latter, adequate concepts have been introduced. Timing requirements also implicitly refer to the globally visible ideal clock of an AUTOSAR system. Through this, timing requirement are also formulated in a definite temporal context.

All concepts for describing signal paths and timing requirements have also been formally described in terms of a UML2 meta-model specification. This ensures that the concepts seamlessly integrate with the existing AUTOSAR concepts and can consequently be used during the specification of an AUTOSAR system.

Derivation of instrumentations for recording of event traces in simulation- and monitoring-based approaches In order to determine timing properties by means of simulation and monitoring-based approaches, an instrumentation of the ECUs is required that records the instances of event and action classes.

9 Summary, Extensibility and Transferability

To obtain such an instrumentation, the hook functions provided by the Virtual Function Bus Tracing Mechanism which correspond to the AUTOSAR-specific event and action classes need to be implemented with logging functions provided by a event-logging mechanism.

The derivation of an instrumentation of an AUTOSAR ECU system must take the decisions from the system configuration (i.e., the SWC-to-ECU mapping and the data-mappings) and the ECU configuration (i.e., the OS and RTE configurations) into account. The logical PathSpecifications from the planning phase (Virtual Function Bus view) based on which the application-specific timing requirements are formulated must be consequently augmented to concrete PathSpecifications that capture the true causal relations in the concrete realization of the AUTOSAR system (System view). For this, OSEvents and COMEvents are inserted into a logical PathSpecification based on the information of the system configuration and ECU configuration. Through this, intra-ECU and inter-ECU communication as well as intra-task and inter-task communication can correctly be accounted for during path analyses such that their effects on the effective path instances and consequently the timing properties are correctly handled. The places in the source code where the logging of events and actions must be performed are the trace hook functions provided by the Virtual Function Bus Tracing mechanism.

With the help of the instrumentation, event traces can be recorded during simulation- or monitoring experiments. These are then be used for the determination of timing properties based on signal path instances.

Determination of timing properties based on instances of signal paths and visualization in application-specific diagrams In order to determine timing properties that directly fit to the timing requirements it has been identified that it is required to determine instances of signal paths based on which these timing properties are then determined. The determination of instances of signal paths, however, is complicated due to the fact that instances of event and action classes can occur independent from each other and in an arbitrary number of times. It is then not directly clear which instance of an event or action class can be interpreted as the cause for the instance of another event or action class.

To address these issues, algorithms that allow a determination of effective path instances have been introduced. Based on these effective path instances, timing properties that address the timing requirements can be determined.

To adequately visualize timing requirements and timing properties, a new form of runtime-oriented diagrams has been introduced, the so-called Timing Oscilloscope Diagrams. A Timing Oscilloscope Diagram shows the development of the determined timing properties over time such that the degree of fulfillment of the timing requirements can then be directly evaluated. Timing Oscilloscope Diagrams are a viable means for control engineers who are used to analyze the development of signals over time in the value domain. The new type of diagram does the same for timing properties, i.e. in the time domain.

A Case Study of an legacy engine control application has been conducted to demonstrate the applicability of the concepts introduced in the thesis and the

monitoring-based approach for the determination of timing properties. For this, the application software of a legacy engine control application system has been re-engineered into an AUTOSAR-compliant ECU system. Then, the concepts for describing the signal paths of the contained real-time applications have been applied. The formal signal path specifications have then been associated with their corresponding timing requirements. From the formal path specifications, an instrumentation could be derived that seamlessly integrates with the Runtime Environment of the AUTOSAR ECU system.

The case study has shown that the concepts of the Timing Model for AUTOSAR presented in the thesis are applicable also to complex AUTOSAR systems, and that simulation and monitoring-based approaches can be employed to evaluate the degree of fulfillment of timing requirements for real-time applications, especially control applications, realized in an AUTOSAR system.

As a future work, it is envisioned to contribute the results of this thesis to the AUTOSAR standardization initiative.

9.2 Extensibility

In the Timing Model for AUTOSAR, the description of signal paths is based on AUTOSAR-specific event and action classes that are introduced based on the Trace events defined by the Virtual Function Bus Tracing Mechanism. The RTEAPI-Actions are action classes by which the interaction of a RunnableEntity with the RTE can be monitored. The concepts for RTEAPIActions have been introduced into the AUTOSAR software component technology in such a way that they are part of the InternalBehavior of an AtomicSoftwareComponentType. They are used to describe a logical signal path in the application software of an AUTOSAR system during the planning phase (Virtual Function Bus view).

The description of a signal path by means of the introduced concepts thus requires that the InternalBehavior of an AtomicSoftwareComponentType is defined. It is thus not possible to describe a signal path through a software component only based on its structural description, i.e., its DataElementPrototypes or OperationPrototypes contained in the respective PortPrototypes.

From a practical point of view, however, the latter could be desirable as the InternalBehavior is specified firstly independently of the structural description of an AtomicSoftwareComponentType and secondly also prior to a concrete InternalBehavior.

A tentative solution could be to adapt the idea of flow specifications and flow implementations from AADL. In AADL, a flow specification describes a signal path by referring to the externally visible features of a component, i.e., the incoming and outgoing flow ports at the structural interface of a component. A flow implementation realizes a flow specification based on the internal implementation of the component, i.e., based on the flow specifications of the contained processes and subcomponents.

A similar approach could be developed for AUTOSAR. The idea is to distinguish between *external* logical PathSpecifications and *internal* logical PathSpecifications.

9 Summary, Extensibility and Transferability

The additional concepts required for that could be designed in such a way that they support a top-down, decompositional approach for external logical PathSpecifications. Compared to that, the hierarchical event chain concept introduced in the thesis supports a bottom-up, compositional approach for internal logical PathSpecifications. Figures 9.2 and 9.3 show how the concepts could practically be employed compared to the concepts introduced in the thesis.

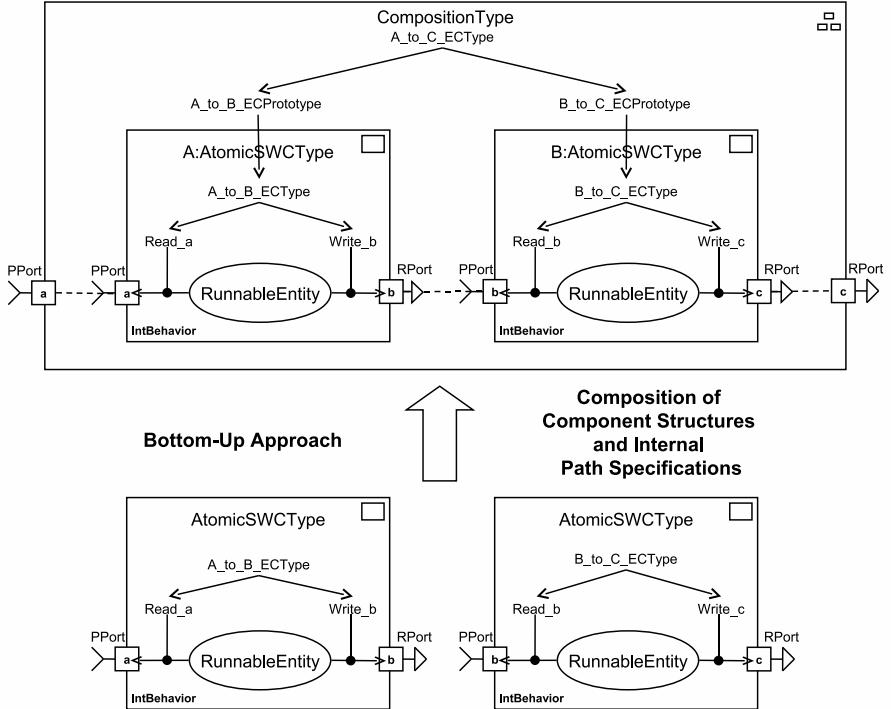


Figure 9.2: Concepts of the Timing Model for AUTOSAR: Support of a compositional bottom-up approach for internal PathSpecifications

The concepts for a top-down, decompositional approach are complementary to the hierarchical event chains concept presented in the thesis which supports a bottom-up, compositional approach. The idea is that an external logical PathSpecification is *realized* by an internal logical PathSpecification. The latter have RTEAPIActions on their leaf level which have a direct counterpart in the instrumentation used for simulation or monitoring of an AUTOSAR system. Figure 9.4 shows the relation between the two concepts.

A drawback of the decompositional top-down approach is that timing properties cannot be determined by means of the simulation- or monitoring-based approaches.

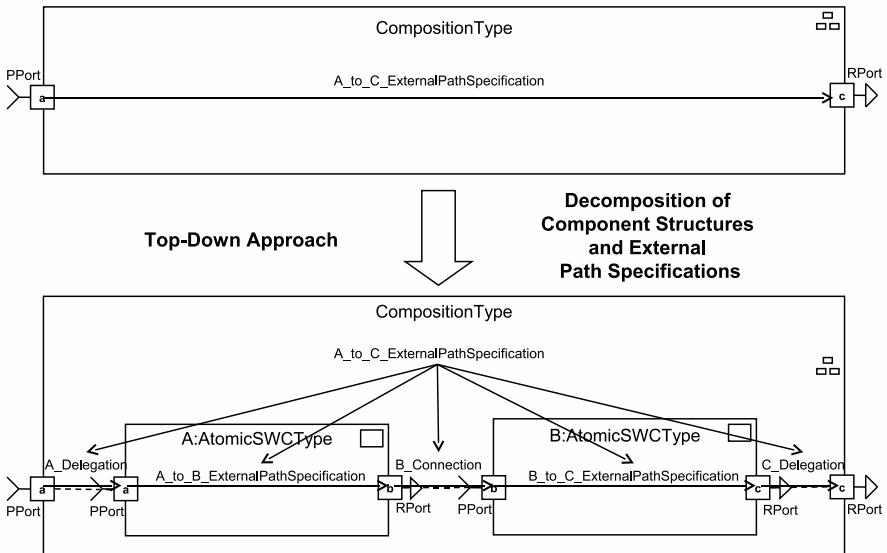


Figure 9.3: Additional concepts: Support of a decompositional, top-down approach for external PathSpecifications

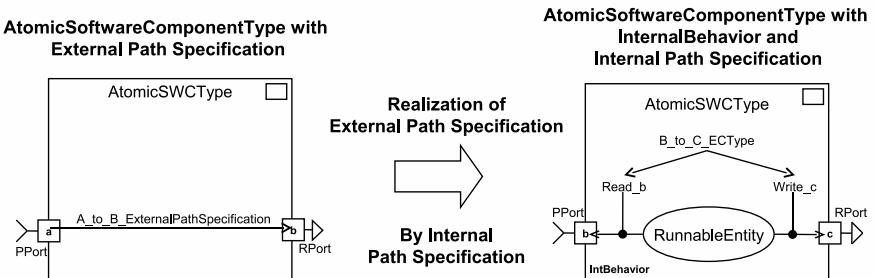


Figure 9.4: External logical Path Specification vs. Internal logical Path Specification

The reason is that a description of a signal path solely based on the structural descriptions of software components is not sufficient for the generation of an instrumentation. It is not clear which Virtual Function Bus Trace hook functions need to be implemented such that the instances of the relevant event and action classes can be recorded. Furthermore, there can be multiple different internal PathSpecifications for one and the same external PathSpecification. The latter depends on the InternalBehavior that is selected for an AtomicSoftwareComponentType.

9.3 Transferability

The results of the work presented in the thesis can also be transferred to other existing works and approaches.

Other Modeling Languages The concepts developed as part of the Timing Model for AUTOSAR could be adapted to other modeling languages such as the EAST-ADL2 or the AADL described in chapter 3.

The EAST-ADL2, an architecture description for automotive embedded systems, provides a functional modeling concept that allows the specification of logical function architectures based on simplified modeling concepts compared to AUTOSAR. For functionalities modeled by means of these concepts it is also required to express timing requirements based on signal path descriptions. As the concepts for functional modeling also employ the type/prototype concept from AUTOSAR, the concepts for describing signal paths by means of hierarchical event chains could be directly adopted.

While the AADL incorporates a concept for modeling signal paths by means of flow specifications and flow implementations, the AADL flow latency analysis framework is not yet fully capable of determining timing properties for all possible configurations of an AADL model. The path analysis algorithms that were introduced in chapter 7 could be adapted to AADL such that timing properties of flows for all possible configurations can be determined.

Other Domains The AUTOSAR standard currently only addresses automotive embedded systems. When AUTOSAR is applied to other domains such as the aerospace domain or the automation domain, the concepts of the Timing Model for AUTOSAR can also be applied there. This means that the results of the thesis have a broader applicability as the results are not limited to automotive embedded real-time systems.

A AUTOSAR Meta-Model Excerpts

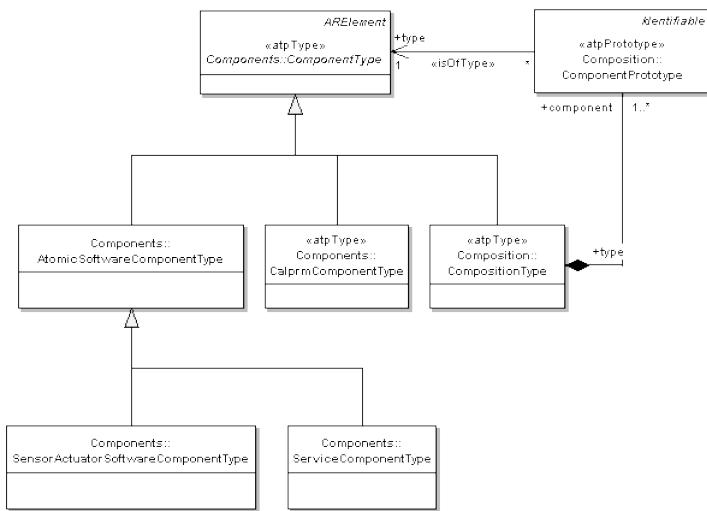


Figure A.1: Meta-model excerpt for components

A AUTOSAR Meta-Model Excerpts

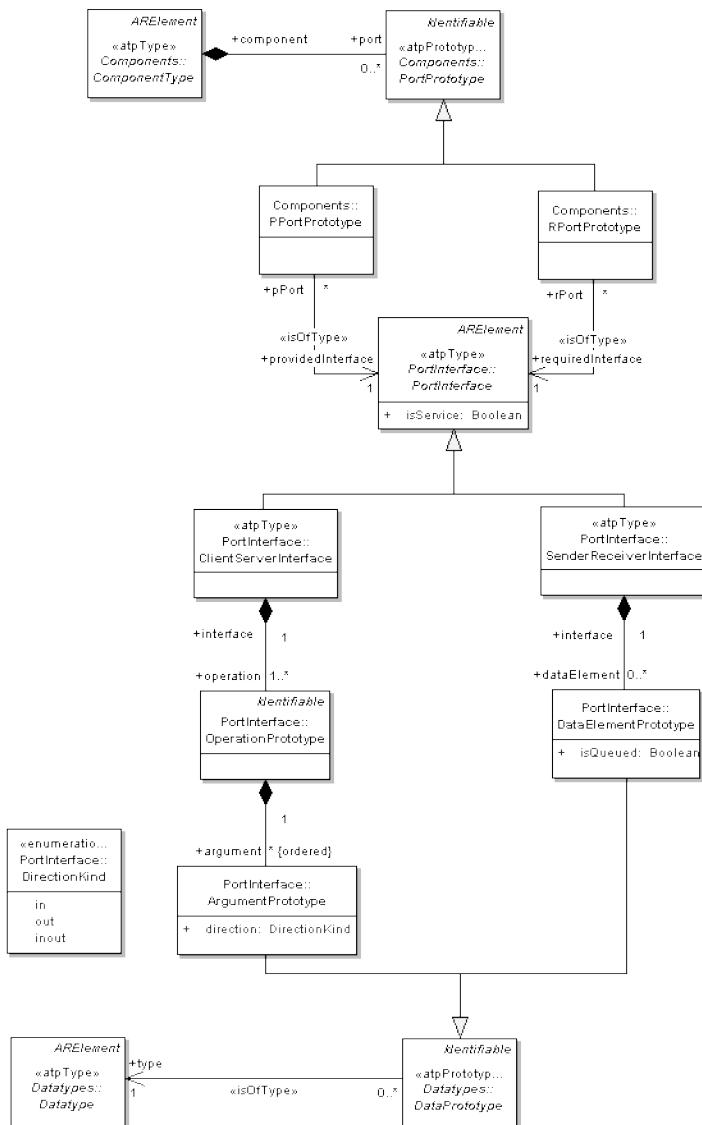


Figure A.2: Meta-model excerpt for ports and interfaces

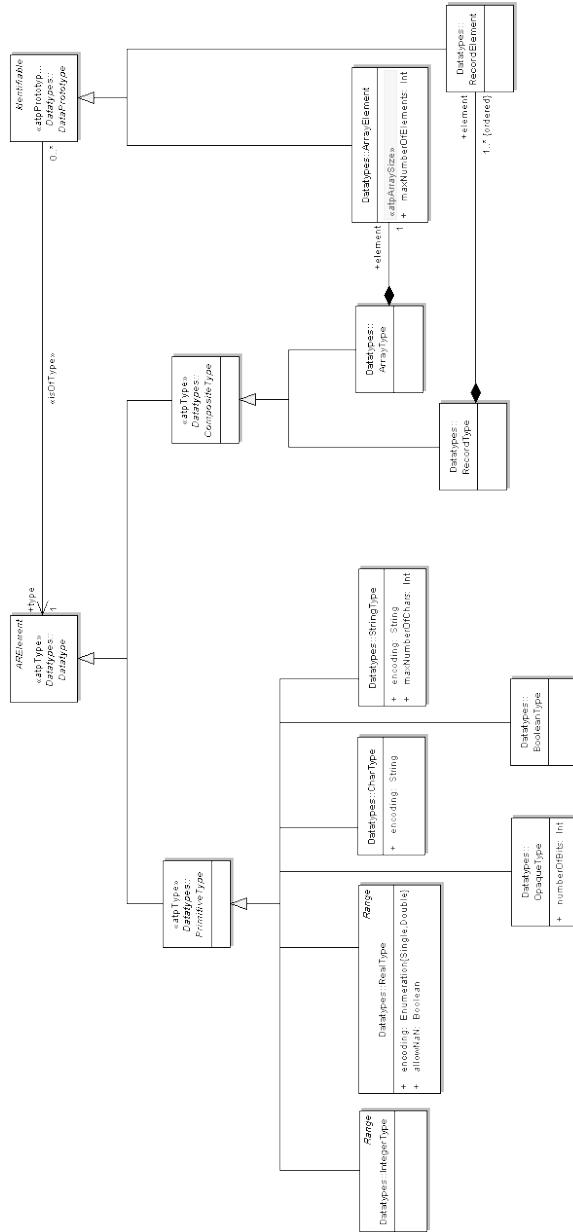


Figure A.3: Meta-model excerpt for data types

A AUTOSAR Meta-Model Excerpts

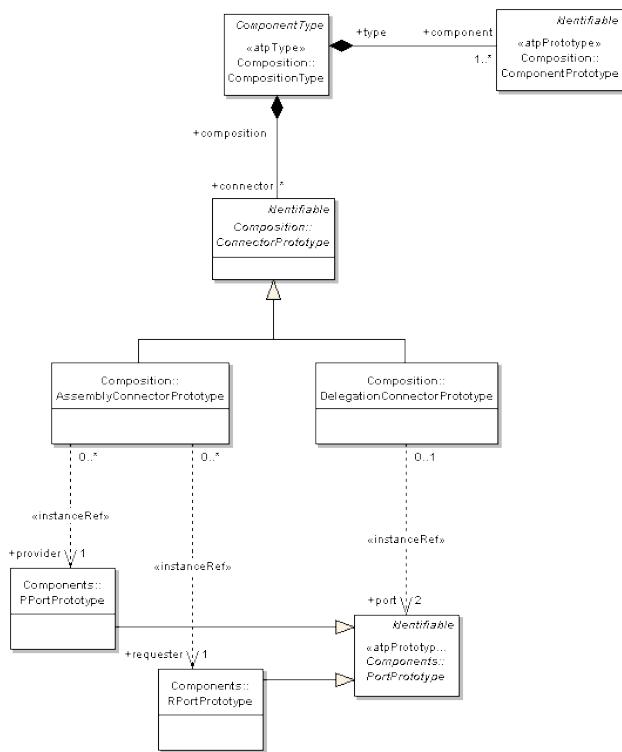


Figure A.4: Meta-model excerpt for connectors

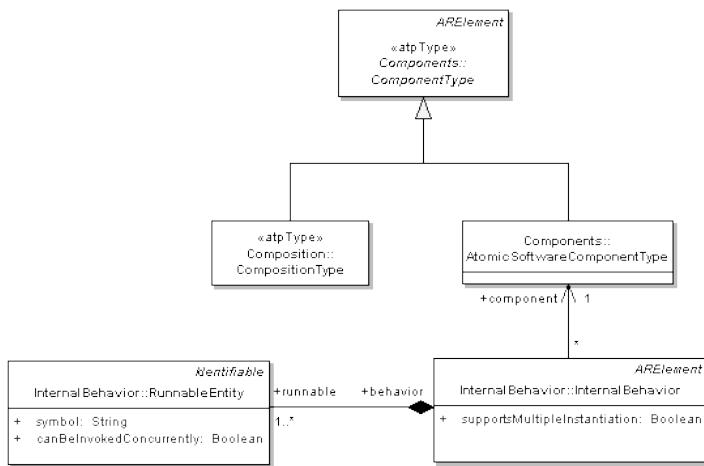


Figure A.5: Meta-model excerpt for internal behavior and RunnableEntity

A AUTOSAR Meta-Model Excerpts

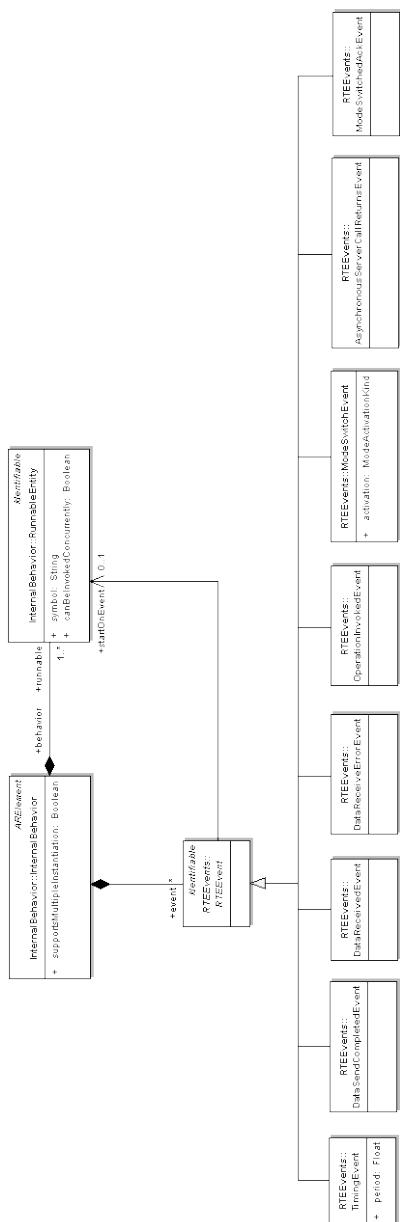


Figure A.6: Meta-model excerpt for RTEEvents

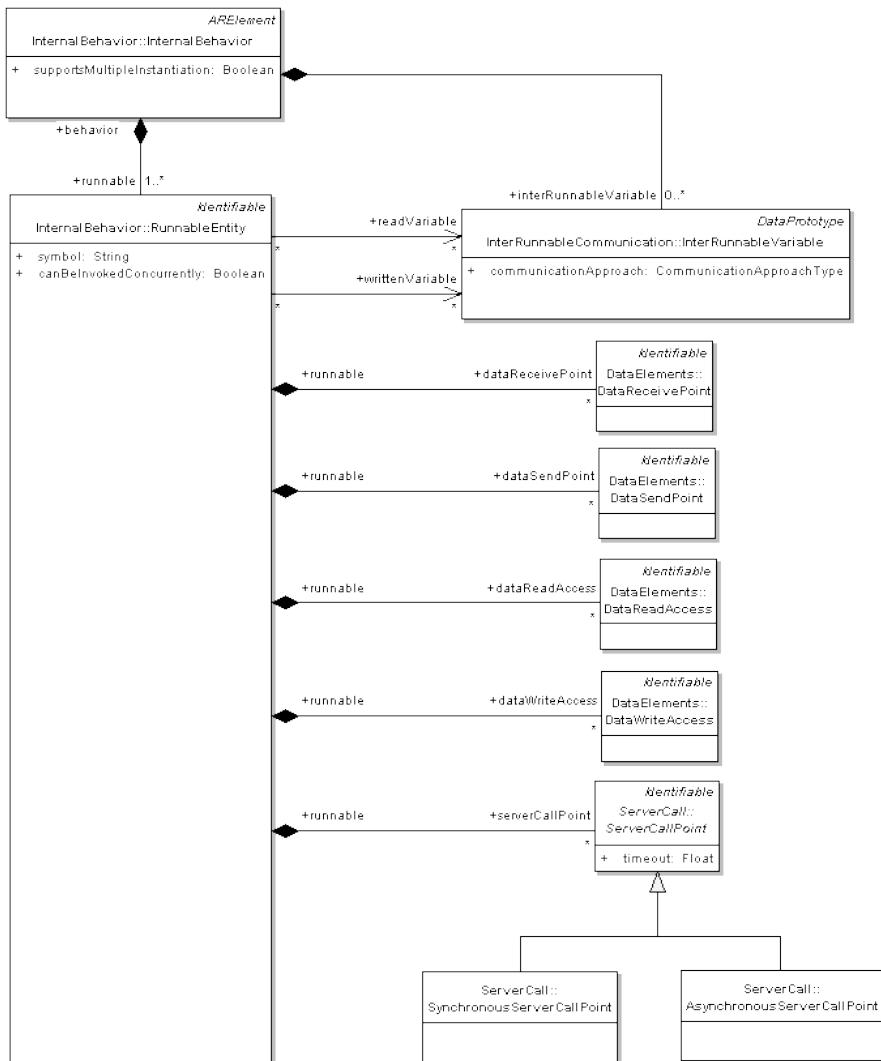


Figure A.7: Meta-model excerpt for communication patterns

B Graphical Notation for AUTOSAR

Graphical representation	Description
	AtomicSoftwareComponentType (specialization of ComponentType). In the lower left corner, the name of the associated InternalBehavior is specified, in the lower right corner, the name of the associated Implementation is specified.
	CompositionType (specialization of ComponentType).
	SensorActuatorSWCType (specialization of AtomicSoftwareComponentType).

Table B.1: ComponentTypes

Graphical representation	Description
	ComponentPrototype: Instantiated ComponentType in the context of a CompositionType .

Table B.2: CompositionTypes and ComponentPrototypes

B Graphical Notation for AUTOSAR

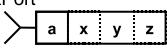
Graphical representation	Description
RPort 	RPortPrototype typed by a SenderReceiverInterface with one DataElementPrototype .
PPort 	PPortPrototype typed by a SenderReceiverInterface with one DataElementPrototype .
RPort 	RPortPrototype being typed by a SenderReceiverInterface with multiple DataElementPrototypes . PPortPrototype is analogical.
RPort 	RPortPrototype RPort typed by a SenderReceiverInterface with a DataElementPrototype a of composite data type ArrayType . The size of the ArrayType is denoted by n . PPortPrototype is analogical.
RPort 	RPortPrototype RPort typed by a SenderReceiverInterface with a DataElementPrototype a of composite data type RecordType . The RecordType contains RecordElements x , y and z . PPortPrototype is analogical.

Table B.3: RequiredPortPrototypes and ProvidedPortPrototypes of Sender/Receiver communication

Graphical representation	Description
RPort 	RPortPrototype RPort typed by a ClientServerInterface with one OperationPrototype op .
PPort 	PPortPrototype PPort typed by a ClientServerInterface with one OperationPrototype op .
RPort 	RPortPrototype RPort typed by a ClientServerInterface with multiple OperationPrototypes op1 , op2 and op3 . PPortPrototype is analogical.

Table B.4: RequiredPortPrototypes and ProvidedPortPrototypes of Client/Server communication

Graphical representation	Description
	AssemblyConnectorPrototype: Connection between a PPortPrototype (PPort at cp1:CT1) and a RPortPrototype (RPort at cp2:CT2) according to the Sender/Receiver communication pattern.
	DelegationConnectorPrototype: Signal-flow delegations in Sender/Receiver communication from the outer port of a CompositionType to the inner port of a ComponentPrototype and vice versa.
	AssemblyConnectorPrototype: Connection between a PPortPrototype (PPort at cp1:CT1) and a PPortPrototype (RPort at cp2:CT2) according to the Client/Server communication pattern.
	DelegationConnectorPrototype: Service delegation in Client/Server communication from the outer port of a CompositionType to the inner port of a ComponentPrototype and vice versa.

Table B.5: AssemblyConnectorPrototypes and DelegationConnectorPrototypes

Graphical representation	Description
	InterRunnableVariable a specified within the InternalBehavior of an AtomicSoftware-ComponentType AtomicSWCType .

Table B.6: InterRunnableVariables

Graphical representation	Description
	RunnableEntity <code>WriterRE</code> specifies a DataWriteAccess <code>dwa</code> to DataElementPrototype <code>a</code> in the SenderReceiverInterface of PPortPrototype <code>PPort</code>
	RunnableEntity <code>ReaderRE</code> specifies a DataReadAccess <code>dra</code> to DataElementPrototype <code>a</code> in the SenderReceiverInterface of RPortPrototype <code>RPort</code>

Table B.7: Implicit Sender/Receiver communication

Graphical representation	Description
	RunnableEntity <code>WriterRE</code> specifies a DataSendPoint <code>dsp</code> to DataElementPrototype <code>a</code> in the SenderReceiverInterface of PPortPrototype <code>PPort</code> .
	RunnableEntity <code>ReaderRE</code> specifies a DataReceivePoint <code>drp</code> to DataElementPrototype <code>a</code> in the SenderReceiverInterface of RPortPrototype <code>RPort</code> .

Table B.8: Explicit Sender/Receiver communication (Unqueued, i.e. data semantics)

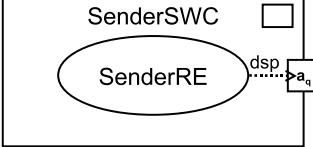
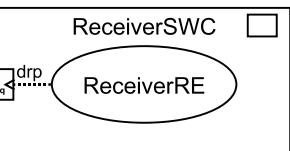
Graphical representation	Description
	RunnableEntity SenderRE specifies a DataSendPoint dsp to DataElementPrototype a in the SenderReceiverInterface of PPortPrototype PPort . The index q of DataElementPrototype a indicates that the communication is queued .
	RunnableEntity ReceiverRE specifies a DataReceivePoint drp to DataElementPrototype a in the SenderReceiverInterface of RPortPrototype RPort . The index q of DataElementPrototype a indicates that the communication is queued . The length of the queue is specified by the QueuedReceiverComSpec and given by $ a_q = n$.

Table B.9: Explicit Sender/Receiver communication (Queued, i.e. event semantics)

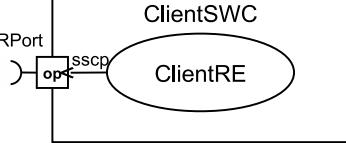
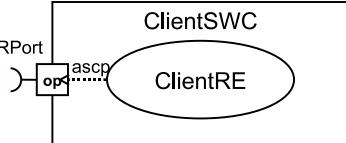
Graphical representation	Description
	RunnableEntity ClientRE specifies a SynchronousServerCallPoint sscp to OperationPrototype op in the ClientServerInterface of PPortPrototype RPort .
	RunnableEntity ClientRE specifies an AsynchronousServerCallPoint ascp to OperationPrototype op in the ClientServerInterface of PPortPrototype RPort .

Table B.10: Synchronous and Asynchronous Client/Server communication (client side)

Graphical representation	Description
	RunnableEntity WriterRE specifies an implicit writing access <i>iw</i> to Inter-RunnableVariable <i>a</i> of the AtomicSoftwareComponentType SWC.
	RunnableEntity ReaderRE specifies an implicit reading access <i>ir</i> to Inter-RunnableVariable <i>a</i> of the AtomicSoftwareComponentType SWC.

Table B.11: Implicit Inter-Runnable communication

Graphical representation	Description
	RunnableEntity WriterRE specifies an explicit writing access <i>ew</i> to Inter-RunnableVariable <i>a</i> of the AtomicSoftwareComponentType SWC.
	RunnableEntity ReaderRE specifies an explicit reading access <i>er</i> to Inter-RunnableVariable <i>a</i> of the AtomicSoftwareComponentType SWC.

Table B.12: Explicit Inter-Runnable communication

Graphical representation	Description
	Specification of a TimingEvent to trigger the referenced RunnableEntity RunnableEntity at a fixed rate. Additionally, an activation offset can be specified.
	Specification of a DataReceivedEvent to trigger the referenced RunnableEntity RunnableEntity on reception of a new data value of DataElementPrototype a on RPort-Prototype RPort .
	Specification of a DataReceivedEvent to trigger the referenced RunnableEntity RunnableEntity on erroneous reception of a new data value of DataElementPrototype a on RPortPrototype RPort , or when a timeout occurred.
	Specification of a DataSendCompletedEvent to trigger the referenced RunnableEntity RunnableEntity on successful sending of a new data value of DataElementPrototype b on PPortPrototype PPort .

Table B.13: RTE Events

Graphical representation

Description

	<p>Specification of a OperationInvokedEvent to trigger the referenced Runnable-Entity RunnableEntity on the invocation of operation op on RPortPrototype RPort.</p>
	<p>Specification of an AsynchronousServerCallReturnsEvent to trigger the referenced RunnableEntity RunnableEntity on the return of a previously initiated asynchronous server call of operation op on PPortPrototype PPort.</p>
	<p>Specification of an ModeSwitchEvent to trigger the referenced RunnableEntity RunnableEntity on the notification of a mode switch by the ECU state manager. The mode switch notification is performed via the mode switch port ModePort.</p>
	<p>Specification of an ModeSwitchAck-Event to trigger the referenced Runnable-Entity RunnableEntity on the notification of a successfully performed mode switch by the ECU state manager. The mode switch notification is performed via the mode switch port ModePort.</p>

Table B.14: RTE Events (cont.)

C Description of AUTOSAR Runtime Environment API Functions and Macros

In an implementation, the specifications of how the RunnableEntities intend to access the DataElementPrototypes in the PortPrototypes are translated into RTE API functions and macros by an RTE generator tool. In the contract phase, the RTE generator tool reads the specification of an AtomicSoftwareComponentType including a corresponding InternalBehavior description and generates a header file with the RTE API functions and macros that are contracted between the specifier and the implementor of the software component. The concrete RTE API function depends on the concrete specification of the intended data access, i.e., whether a DataReceivePoint or DataReadAccess, or whether a DataSendPoint or DataWriteAccess is specified.

With respect to the AUTOSAR development methodology, the RTE API functions are contracted for a single AtomicSoftwareComponent *Type* in the RTE contract phase while their final implementation is generated in the the RTE generation phase (see section 4.3.3).

The tables in the following sections provide an overview on the RTE API functions.

C.1 Sender/Receiver Communication

Table C.1 shows the RTE API functions for providing or requesting action in Sender/Receiver communication.

RTE API function signature	Description
<code>void Rte_IWrite_<re>_<p>_<d></code> <code>([IN Rte_Instance],</code> <code> IN <type>)</code>	Implicit write action with data semantics (DataWriteAccess)
<code><return> Rte_IRead_<re>_<p>_<d></code> <code>([IN Rte_Instance <instance>])</code>	Implicit read action with data semantics (DataReadAccess)
<code>Std_ReturnType Rte_Write_<p>_<d></code> <code>([IN Rte_Instance <instance>],</code> <code> IN <data>)</code>	Explicit write action with data semantics (DataSendPoint)
<code>Std_ReturnType Rte_Read_<p>_<d></code> <code>([IN Rte_Instance <instance>],</code> <code> OUT <data>)</code>	Explicit read action with data semantics (DataReceivePoint)
<code>Std_ReturnType Rte_Send_<p>_<d></code> <code>([IN Rte_Instance <instance>],</code> <code> IN <data>)</code>	Explicit send action with event semantics (DataSendPoint)
<code>Std_ReturnType Rte_Receive_<p>_<d></code> <code>([IN Rte_Instance <instance>],</code> <code> OUT <data>)</code>	Explicit receive with event semantics (DataReceivePoint)
<code><re></code>	Name of the RunnableEntity
<code><p></code>	Name of the PortPrototype
<code><d></code>	Name of DataElementPrototype in the Sender-ReceiverInterface which types the PortPrototype
<code><data></code>	Value of DataElementPrototype being passed to write action
<code><return></code>	Value of DataElementPrototype being returned from read action
<code><instance></code>	Software-component instance handle provided by RTE
<code>Std_ReturnType</code>	Standard return type from RTE indicating the success status of an invocation

Table C.1: RTE API functions for Sender/Receiver communication

C.2 Inter-Runnable Communication

Table C.2 shows the RTE API functions which are contracted (RTE contract phase) or generated (RTE generation phase) for a read or write action in Inter-Runnable communication.

RTE API function	Description
<code>void Rte_IrvIWrite_<re>_<irv> ([IN RTE_Instance <instance>], IN <data>) <return> Rte_IrvIRead_<re>_<irv> ([IN RTE_Instance <instance>])</code>	Implicit write action (reference to IRV with implicit communication approach)
<code>void Rte_IrvWrite_<re>_<irv> ([IN RTE_Instance <instance>], IN <data>)</code>	Explicit write action (reference to IRV with explicit communication approach)
<code><return> Rte_IrvRead_<re>_<irv> ([IN RTE_Instance <instance>])</code>	Explicit read action (reference to IRV with explicit communication approach)
<code><re></code>	Name of the RunnableEntity
<code><irv></code>	Name of the InterRunnableVariable
<code><data></code>	Value of InterRunnableVariable being passed to write action
<code><return></code>	Value of InterRunnableVariable being returned from read action
<code><instance></code>	Software-component instance handle provided by the RTE

Table C.2: RTE API functions for Inter-Runnable communication

C.3 Client/Server Communication

Table C.3 shows the RTE API functions which are contracted (RTE contract phase) or generated (RTE generation phase) for a server call action in both synchronous and asynchronous Client/Server communication as well as the RTE API function that is to be used by the client for fetching the result in asynchronous Client/Server communication.

RTE API function	Description
<pre>Std_ReturnType Rte_Call_<p>_<o> ([IN Rte_Instance <instance>], [IN IN/OUT OUT] <param_1>... [IN IN/OUT OUT] <param_n>)</pre> <pre>Std_ReturnType Rte_Result_<p>_<o> ([IN Rte_Instance <instance>], [OUT <param 1>]... [OUT <param n>])</pre>	<p>Synchronous or asynchronous server call (asynchronous: no OUT or IN/OUT arguments) (SynchronousServerCallPoint or AsynchronousServerCallPoint)</p> <p>Collect results from asynchronous server call (AsynchronousServerCallPoint)</p>
<p> <o> <param> <instance> Std_ReturnType	Name of the PortPrototype Name of the OperationPrototype Value of an IN, IN/OUT or OUT parameter to the operation call Software-component instance handle provided by RTE Standard return type from RTE indicating the success status of an invocation

Table C.3: RTE API functions for Client/Server communication

D Amendments to the Virtual Function Bus

Tracing

Mechanism

D.1 Monitoring of Signal (Group) Transmission and Reception via COM Events

The VFB Tracing Mechanism defines several hook functions to monitor the interaction of software components which are deployed onto distinct ECUs. The hook functions resemble events corresponding to the transmission and reception of signals and signal groups via the remote communication services that are abstracted by the RTE and implemented by the COM stack. The hook functions which enable the monitoring of remote communication are hook functions at the interface of the RTE to the COM module.

The VFB Tracing Mechanism defines the following hook functions:

- The signal transmission hook `Rte_ComHook_<signalName>.SigTx` is called just before the RTE invokes the `Com_SendSignal` API function (transmission of primitive data) or the `Com_SendSignalGroup` API function (transmission of complex data) provided by the COM module. This hook thus resembles a signal transmission request event by a software component making use of the remote communication service provided by the COM module.
- The signal reception hook `Rte_ComHook_<signalName>.SigRx` is called directly after the RTE has successfully received a signal or signal group through `Com_ReceiveSignal`. This hook thus resembles a signal reception event notifying the RTE of newly available data.
- Both the RTE of a sender and receiver are notified by the COM module upon successful signal transmission or newly available data for signal reception. The COM notification hook is called when the COM module invokes a COM-callback function `Rte_ComCbk` offered by the RTE, notifying the RTE that a signal has been received via the COM service. For signal transmission, depending on the success status of the remote communication, four occurrences of the COM notification hook exist to indicate the exact status of the signal reception:
 - Inv** to indicate the invalidation of a signal to a receiver
 - TAck** to indicate an acknowledgement of a signal transmission to a receiver

TErr to indicate an error of a signal transmission to a receiver

TOut to indicate a timeout of a signal transmission to a receiver.

- The signal invalidation hook is called just before the RTE invokes the `COM_InvalidateSignal` or `COM_InvalidateShadowSignal` function provided by the COM module. This hook thus resembles a signal invalidation request event by the RTE.

In the following, the sequence of actions in terms of the API calls that are involved in a transmission and reception through the RTE and via COM are described. This gives an insight into which COM events are required to be monitored with the help of the respective hook functions such that a communication between RunnableEntities of software components deployed on distinct ECUs can be monitored. As individual signals and signal groups are treated differently, both cases are explained separately.

Signal Transmission and Signal Reception

Figure D.1 depicts an overview of the RTE and communication services layer of an AUTOSAR ECU, highlighting the API functions involved in signal transmission and signal reception.

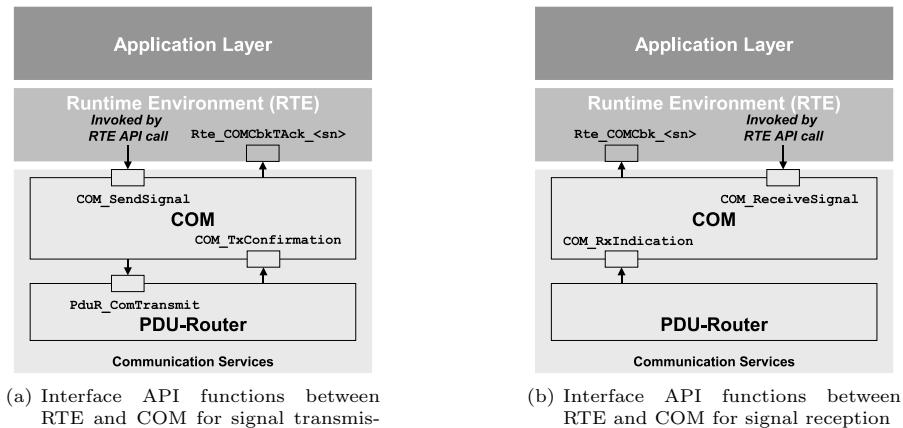


Figure D.1: API functions of RTE and COM involved in remote signal communication (primitive data)

Steps of a Signal Transmission Process When a signal is to be transmitted via COM, the RTE calls the function `COM_SendSignal`, handing over the signal to be sent identified through its unique SignalID. This can be monitored by means of

D.1 Monitoring of Signal (Group) Transmission and Reception via COM Events

the `Rte_ComHook_<signalName>.SigTx` hook. The COM module marshals the signal into an I-PDU and passes the latter to the PDU-Router. The PDU-Router hands the I-PDU over to the next basic software module, being notified upon successful signal transmission by the lower basic software module. The success of the signal transmission is then notified to the COM module which itself invokes the RTE signal callback function `Rte_COMCbkTAck_<sn>` as an acknowledgement of successful signal transmission to the RTE. This can be monitored by means of the `Rte_ComHookTAck_<signalName>` hook.

Steps of a Signal Reception Process When a signal is received via COM, the COM module indicates that a new signal has arrived via the RTE signal callback function `Rte_COMCbk_<sn>`. The `Rte_ComHookTAck_<signalName>` hook can be used to monitor this event. The RTE can then actively receive the signal by calling the `COM_ReceiveSignal` function. This can be monitored by means of the `Rte_ComHook_<signal>.SigRx` hook function which is invoked by the RTE.

Signal Group Transmission and Signal Reception

Figure D.2 depicts an overview of the RTE and communication services layer of an AUTOSAR ECU, highlighting the API functions involved in signal group transmission and signal group reception. Complex data types such as records or arrays consist of multiple individual data items which must be considered as a group. In remote communication, the individual data items must thus be marshalled and unmarshalled together.

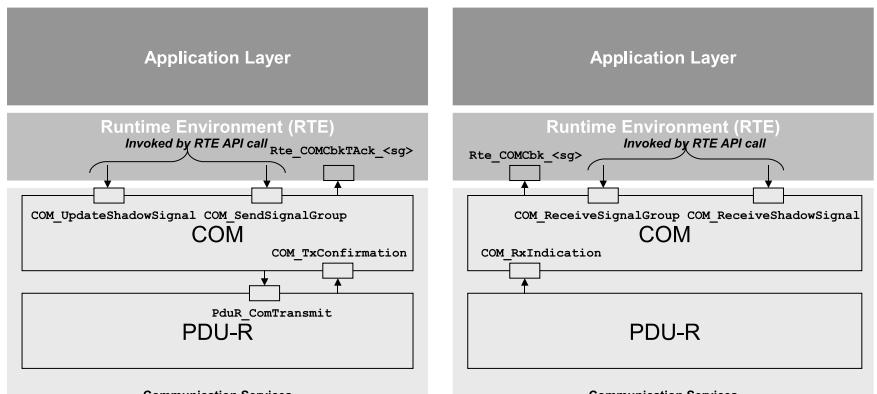


Figure D.2: API functions of RTE and COM involved in remote signal communication (complex data)

Steps of a Signal Group Transmission Process When a complex data item is to be sent via COM, the RTE first calls `COM_UpdateShadowSignal` repeatedly for each entry of the complex data item and then calls `COM_SendSignalGroup` to notify the COM module that the signals of the signal group shall be transmitted. The latter can be monitored by means of the `Rte_ComHook_<signalgroup>.SigTx` hook. The COM module then uses the service provided by the PDU-Router module to transmit the I-PDU. When the signal group has been sent successfully, the COM module is notified by the PDU-Router module. The COM module then notifies the RTE by invoking the RTE callback function for transmission acknowledgement of the signal group, `Rte_COMCbkTAck_<sg>`. This can be monitored by means of the `Rte_ComHook_TAck_<signalgroup>` hook on the sending ECU.

Steps of a Signal Group Reception Process When a signal group is received by the COM module, the COM module can indicate this to the RTE by invoking the signal group callback function `Rte_COMCbk_<sg>`. This can be monitored by means of the `Rte_ComHook_TAck_<signalgroup>` hook on the receiving ECU. The RTE can then receive the signal group by repeatedly calling the `COM_ReceiveShadowSignal` function in order to copy the data of the individual signals of the signal group to the entries of the complex data item. When all entries have been copied, the `COM_ReceiveSignalGroup` is called which indicates that the process of receiving the signal group has finished. This can be monitored by means of the `Rte_ComHook_<signalgroup>.SigRx` hook on the receiving ECU.

E Amendments to AUTOSAR Timing Model

E.1 Amendments to Definitions for Event Classes and Action Classes

E.1.1 COM Events for Transmission and Reception of System Signal Groups

To monitor the transmission and reception of signal groups, COMSignalGroupEvents are introduced.

Definition 39 A *COMSignalGroupEvent* represents a COM signal group event that can be observed when a system signal group is transmitted to or received from the COM module by the RTE.

The COMSignalGroupEvent has a reference to the respective system signal group.

Definition 40

- A *COMSendSignalGroupEvent* represents a COM signal group event that can be observed when a system signal group is transmitted to the COM module by the RTE.
- A *COMReceiveSignalGroupEvent* represents a COM signal group event that can be observed when a system signal group is received from the COM module by the RTE.

For COMSignalGroupEvents, there is also no graphical representation as they are also not directly intended for modeling.

E.2 Meta-Model Specifications for AUTOSAR-specific Event and Action Classes

This section provides meta-model specifications for the AUTOSAR-specific event classes and action classes in terms of UML2 class diagrams. The classes with the white background stem from the original AUTOSAR R2.1 meta-model, the classes with the grayish background are complements of our Timing Model for AUTOSAR.

E.2.1 RunnableEntity Events

Figure E.1 depicts the meta-model specification for the RunnableEntityEvents.

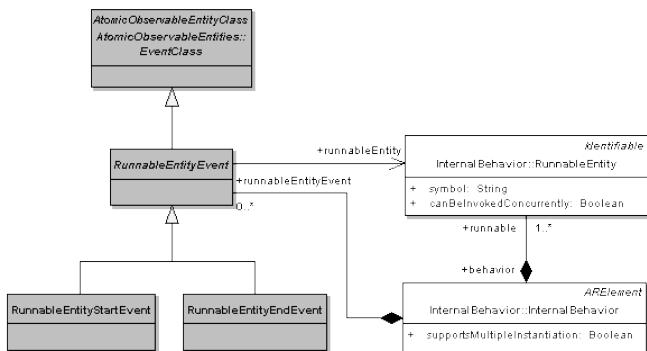


Figure E.1: Meta-model specification of RunnableEntityEvents

A RunnableEntityEvent has an explicit reference to a RunnableEntity. It is defined in the context of the InternalBehavior of an AtomicSoftwareComponentType.

E.2.2 Runtime Environment (RTE) API Actions

Abstract superclass for concrete RTEAPIActions

Figure E.2 depicts the meta-model specification for the abstract superclass RTEAPIAction. All RTEAPIActions are specified in the context of the InternalBehavior of an AtomicSoftwareComponentType.

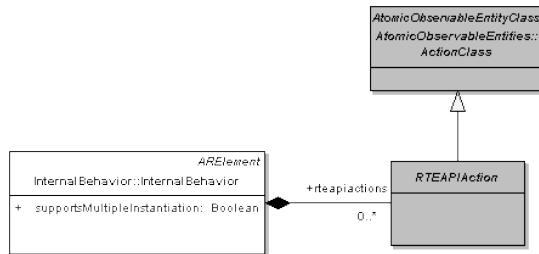


Figure E.2: Meta-model specification of abstract superclass RTEAPIAction

Concrete RTEAPIActions for Sender/Receiver communication

Figure E.3 shows the meta-model specification for the abstract superclasses of Sender/Receiver communication.

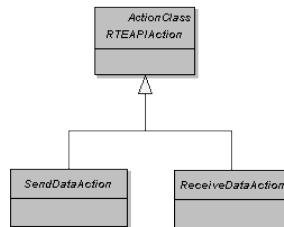
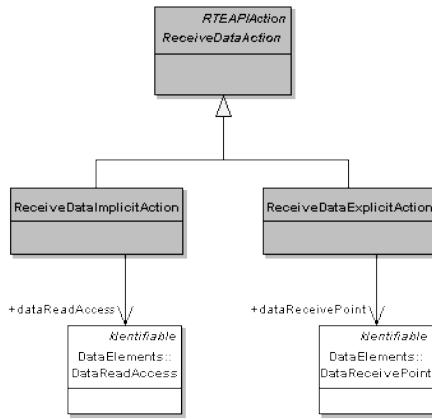


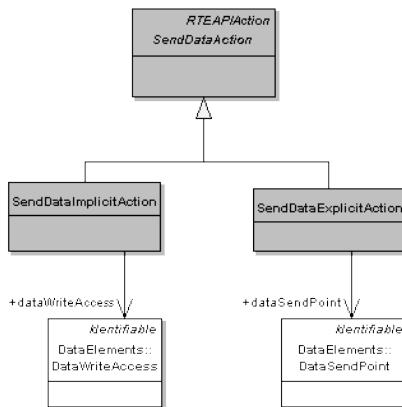
Figure E.3: Meta-model specification of abstract superclasses SendDataAction and ReceiveDataAction

E Amendments to AUTOSAR Timing Model

Figure E.4 shows the meta-model specification for the concrete RTEAPIEvents of Sender/Receiver communication.



(a) **ReceiveDataImplicitAction** and **ReceiveDataExplicitAction**



(b) **SendDataImplicitAction** and **SendDataExplicitAction**

Figure E.4: Meta-model specification for concrete RTEAPIActions or Sender/Receiver communication

Concrete RTEAPIActions for Interrunnable Communication

Figure E.5 shows the meta-model specification for the concrete RTEAPIActions of Interrunnable communication.

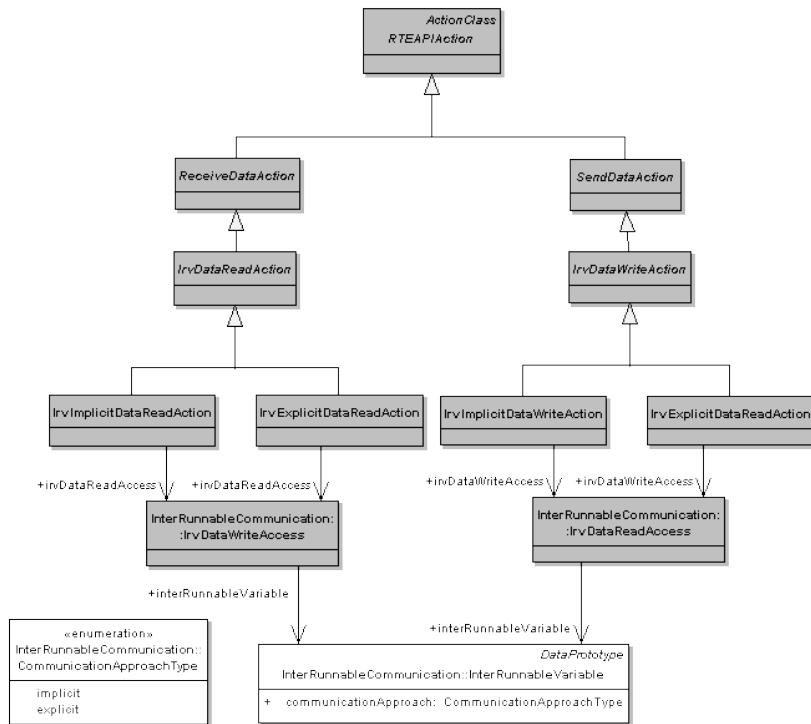


Figure E.5: Meta-model specification for the concrete RTEAPIActions of Interrunnable communication

E Amendments to AUTOSAR Timing Model

Concrete RTEAPIActions for Client/Server communication

Figure E.6 depicts the meta-model specification for the abstract superclasses for Client/Server communication.

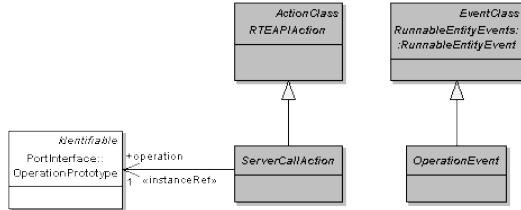


Figure E.6: Meta-model specification of abstract superclasses ServerCallAction and OperationEvent for Client/Server communication

Note that the `ServerCallAction`, compared to the `OperationEvent`, has an additional reference to an `OperationPrototype`. This is required as a `ServerCallPoint` can reference multiple `OperationPrototypes`.

Client Side

Figure E.7 depicts the meta-model specification for the concrete ServerCallActions for the client side of a Client/Server communication.

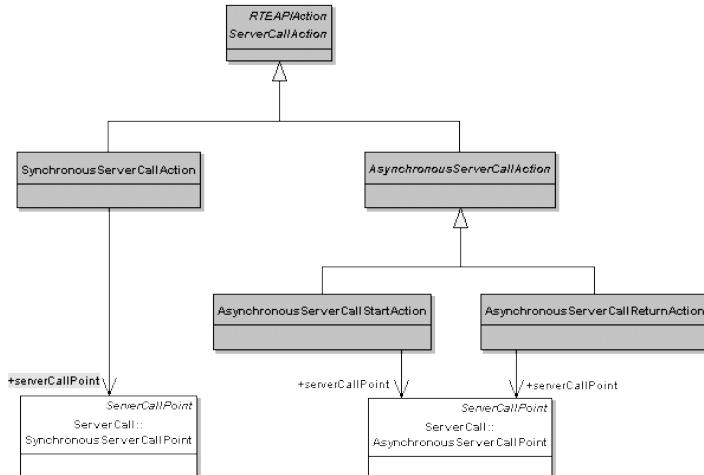


Figure E.7: Meta-model specification for concrete RTEAPIActions on client side

E.2 Meta-Model Specifications for AUTOSAR-specific Event and Action Classes

Note that for asynchronous Client/Server communication, there are two action classes. The AsynchronousServerCallStartAction and the AsynchronousServerCallReturnAction both point to the same AsynchronousServerCallPoint.

Server side

Figure E.8 shows the meta-model specification for the event classes for the server side of a Client/Server communication. Note that these are not RTEAPIActions but event classes that are derived from RunnableEntityEvent. Although these event classes are not directly required for the specification of signal paths, they can be monitored by means of the corresponding Virtual Functional Bus trace hook functions.

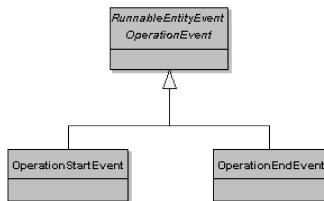


Figure E.8: Meta-model specification of concrete EventClasses for server side

E.2.3 Communication (COM) Events

COMEEvents for Transmission and Reception of System Signals

Figure E.9 depicts the meta-model specification of COMSignalEvents.

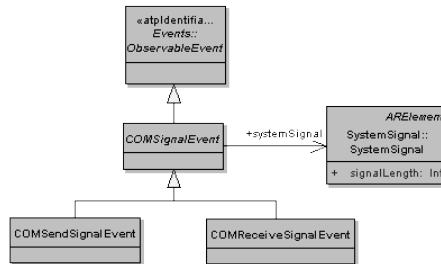


Figure E.9: Meta-model specification of COMSignalEvents for Inter-ECU communication

COMEEvents for Transmission and Reception of System Signal Groups

Figure E.10 depicts the meta-model specification of COMSignalGroupEvents.

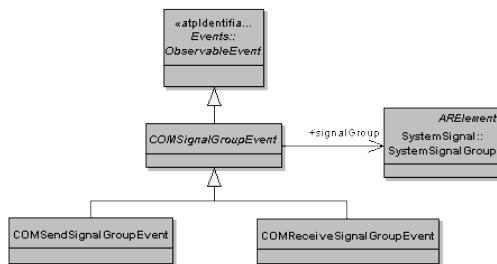


Figure E.10: Meta-model specification of COMSignalGroupEvents for Inter-ECU communication

E.2.4 Operating System (OS) Events

Figure E.11 depicts the meta-model specification for OS-task related OSEvents, so-called TaskEvents.

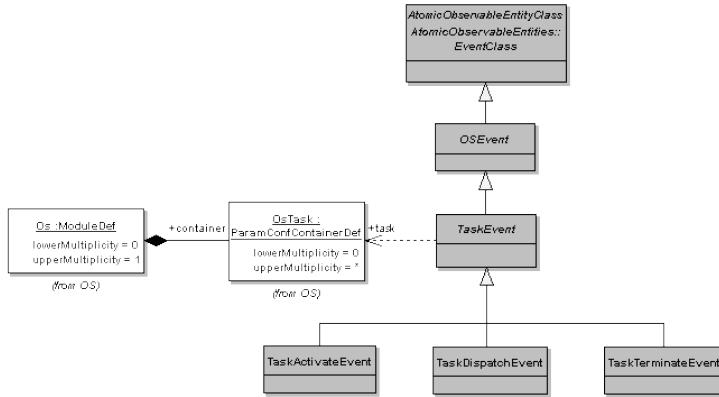


Figure E.11: Meta-model specification of TaskEvents

Figure E.12 depicts the meta-model specification for OS-event related OSEvents, so-called OSEventEvents.

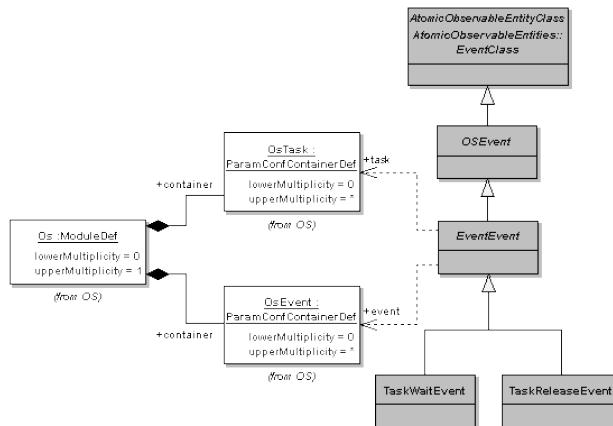


Figure E.12: Meta-model specification of EventEvents

E.3 Overview on AUTOSAR-specific Event and Action Classes

Scope	Concrete Event Class	Referenced AUTO-SAR entity
Life cycle of RunnableEntities	RunnableEntityStartEvent RunnableEntityEndEvent	RunnableEntity

Table E.1: Overview of RunnableEntityEvents

Scope	Concrete Event Class	Referenced AUTO-SAR entity
Life cycle of OS-tasks	TaskActivateEvent TaskDispatchEvent TaskTerminateEvent	OsTask
	TaskWaitEvent TaskReleaseEvent	OsEvent

Table E.2: Overview of OSEvents

Scope	Concrete Event Class	Referenced AUTO-SAR entity
Inter-ECU communication	COMSendSignalEvent COMReceiveSignalEvent	SystemSignal
	COMSendSignalGroupEvent COMReceiveSignalGroupEvent	SystemSignalGroup

Table E.3: Overview of COMEvents

Scope	Communication Pattern	Concrete Action Class	Referenced AUTOSAR entity
with RTE Interaction of RunnableBmittities	Sender/Receiver communication	implicit ReceiveDataImplicitAction SendDataImplicitAction	DataReadAccess DataWriteAccess
		explicit ReceiveDataExplicitAction SendDataExplicitAction	DataReceivePoint DataSendPoint
	Interrunnable communication	implicit IrvDataReadImplicitAction IrvDataWriteImplicitAction	IrvDataReadAccess (implicit) Irv DataWriteAccess (implicit)
		explicit IrvDataReadExplicitAction IrvDataWriteExplicitAction	Irv DataReadAccess (explicit) Irv DataWriteAccess (explicit)
Client/Server communication	synchronous client	SynchronousServerCallAction	SynchronousServerCallPoint
	asynchronous client	AsynchronousServerCallStartAction AsynchronousServerCallReturnAction	AsynchronousServerCallPoint
	server	OperationStartEvent ^a OperationEndEvent ^b	RunnableEntity

Table E.4: Overview of RTEAPIActions

^aThis is not an RTEAPIAction, but a specialization of RunnableEntityStartEvent

^bThis is not an RTEAPIAction, but a specialization of RunnableEntityEndEvent

E.4 Meta-Model Specification for Hierarchical Event Chains

Figure E.13 depicts the meta-model specification for the definition of hierarchical order relations, so-called event chains, between event classes and action classes.

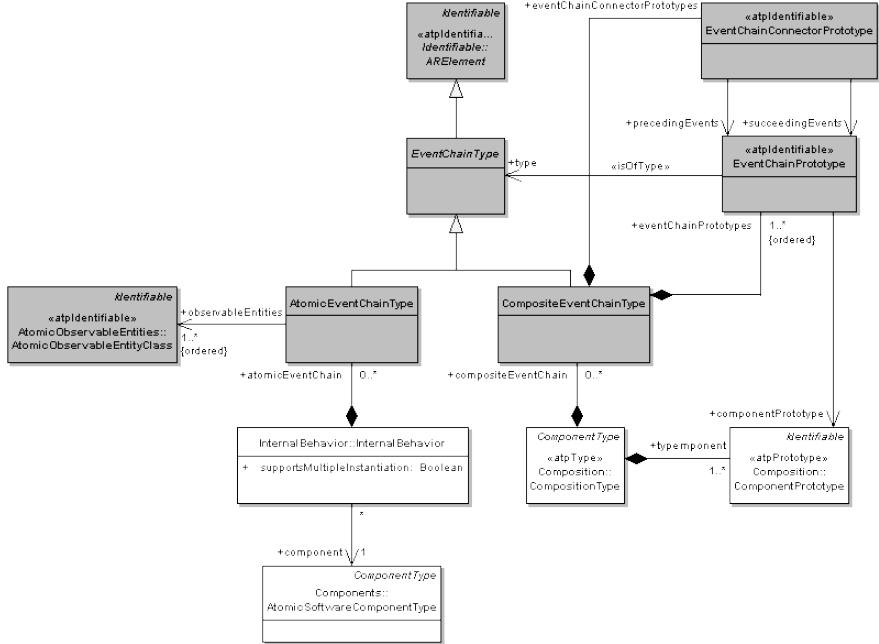


Figure E.13: Meta-model specification for hierarchical event chains

The integration of the concepts for hierarchical event chains with the existing AUTOSAR concepts is achieved in the following way:

- Through the inheritance from the AUTOSAR base class `Identifiable`, the class `EventChainType` and all its subclasses are integrated into the AUTOSAR meta-model and can be identified by means of their AUTOSAR short-names.
- Through the establishment of a composition relation between class `InternalBehavior` and class `AtomicEventChainType`, the latter have a defined context in which they must be specified. Consequently, `AtomicEventChainTypes` are part of the `InternalBehavior` description of an `AtomicSoftwareComponentType`.
- Through the establishment of a composition relation between class `CompositionType` and class `CompositeEventChainType`, the latter also have a defined context in which they must be specified.

E.5 Meta-Model Specifications for Timing Requirements

E.5.1 Abstract Superclasses for Timing Requirements

Figure E.14 depicts the meta-model specification for application-specific timing requirements in AUTOSAR. The central entity is the abstract superclass **TimingRequirement** from which all concrete classes **TimingRequirements** are derived to express application-specific timing requirements, and through which the integration with existing AUTOSAR concepts is achieved.

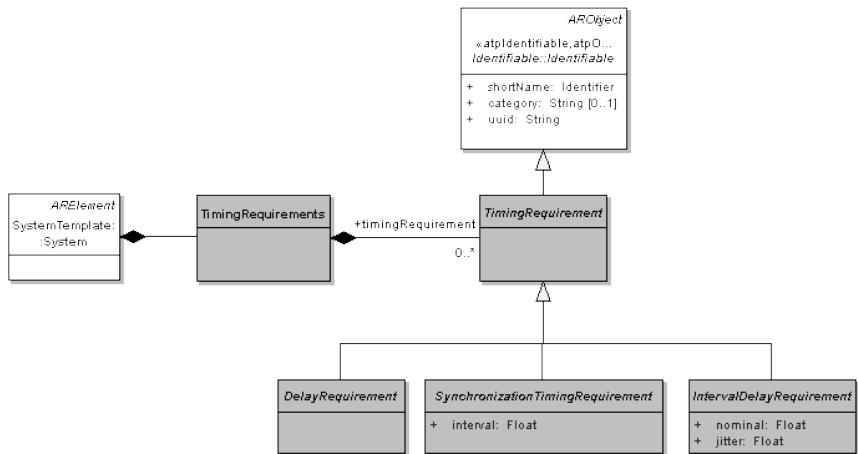


Figure E.14: Meta-model specification for timing requirements

The integration of the timing requirements with existing AUTOSAR concepts is two-fold. In order to treat the timing requirements as AUTOSAR modeling artifacts, a specialization-relationship of the class **TimingRequirement** to the class **Identifiable** from the existing AUTOSAR meta-model is defined. Through this, **TimingRequirements** can be identified and thus also referenced by their shortname. As **Identifiable** artifacts cannot live stand-alone such as **ARObjects** in an **ARPackage**, a context needs to be defined in which they are to be described. For this purpose, a composition-relationship between the AUTOSAR meta-model class **System** and the abstract class **TimingRequirement** is defined.

E.5.2 Delay Requirements

Figure E.15 depicts the meta-model specification for **DelayRequirements**.

As can be seen in the figure, a **DelayRequirement** references a single **EventChainType**. We assume that this **EventChainType** is a **PathSpecification**. The entity **ReactionTimeDelayRequirement** shown in the meta-model specification resembles the

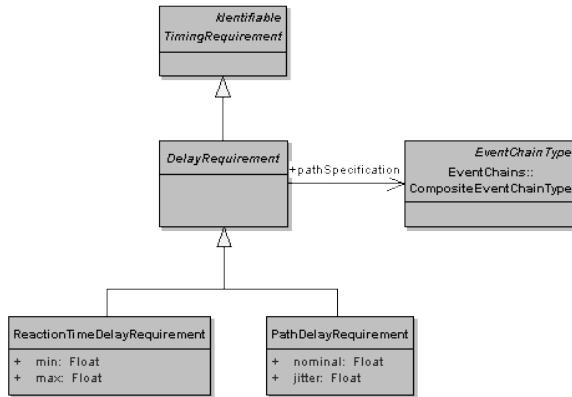


Figure E.15: Meta-model specification for Delay Requirements

notion of delay requirements for reactive real-time applications. The entity *PathDelayRequirement* shown in the meta-model specification resembles the notion of delay requirements for control applications.

E.5.3 Interval Delay Requirements

Figure E.16 depicts the meta-model specification for the *IntervalDelayRequirement*.

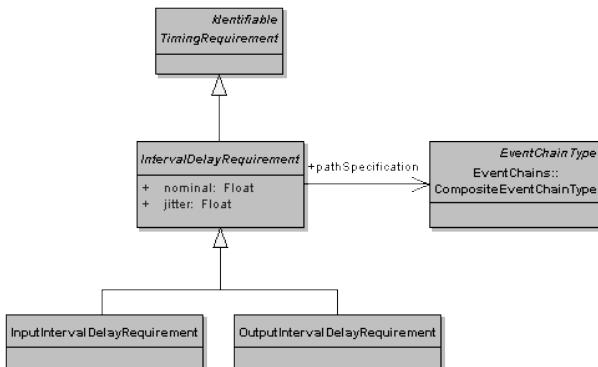


Figure E.16: Meta-model specification for IntervalDelay Requirements

The *IntervalDelayRequirement* has a direct reference to an *EventChainType* that denotes a *PathSpecification*.

E.5.4 Synchronization Requirements

Figure E.17 depicts the meta-model specification to express these two kinds of SynchronizationRequirements.

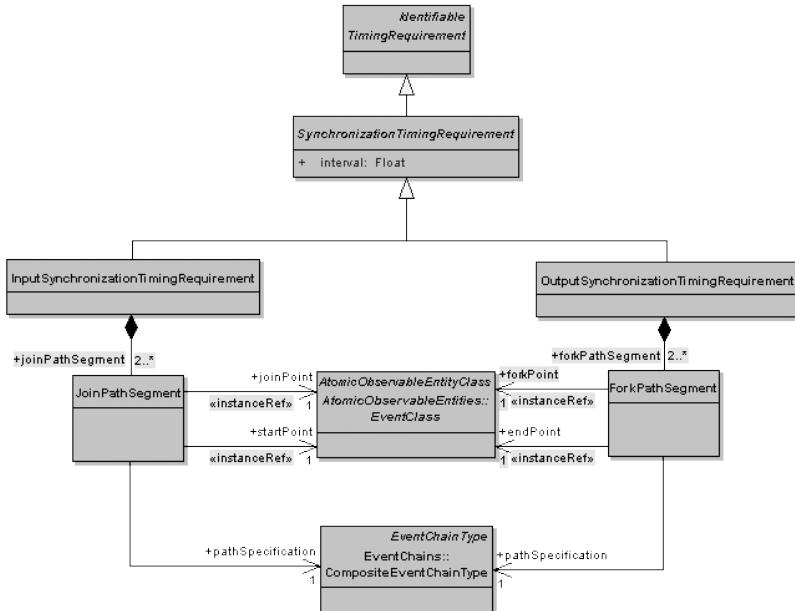
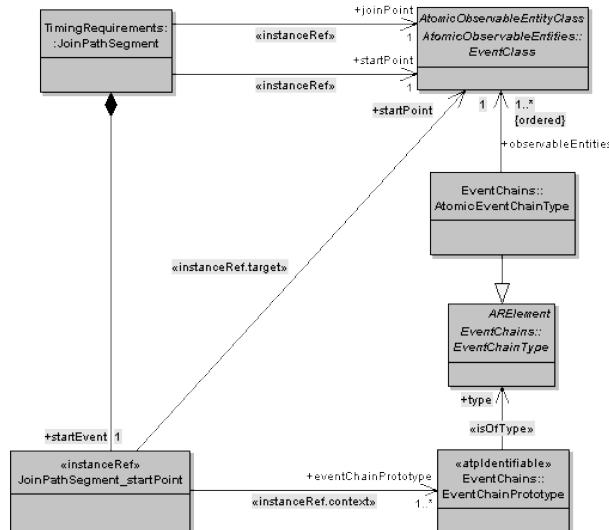
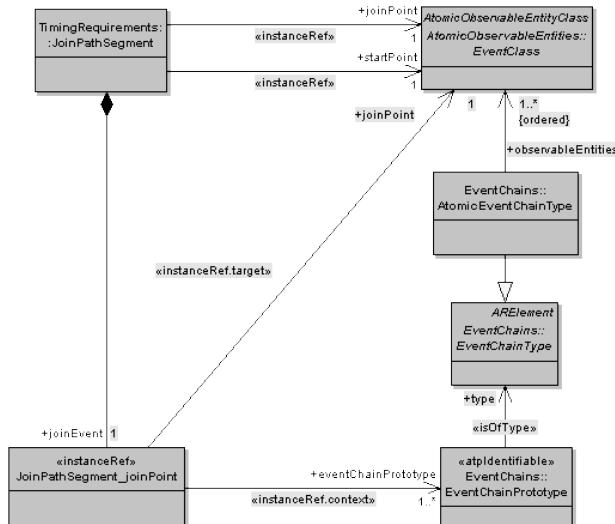


Figure E.17: Meta-model specification for SynchronizationRequirements

E Amendments to AUTOSAR Timing Model



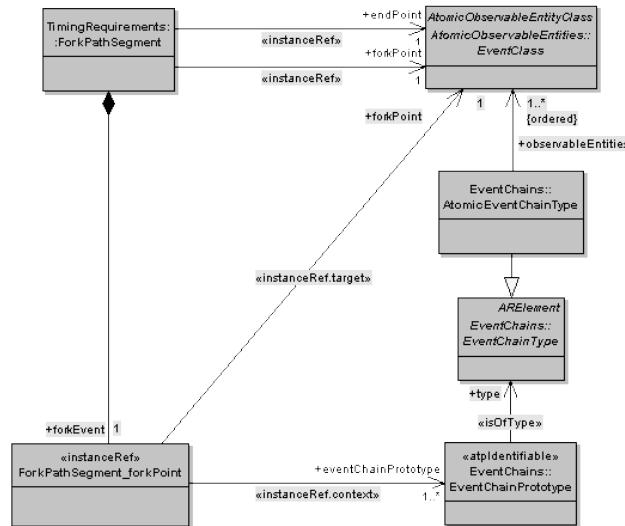
(a) Instance reference for StartPoint of a JoinPathSegment



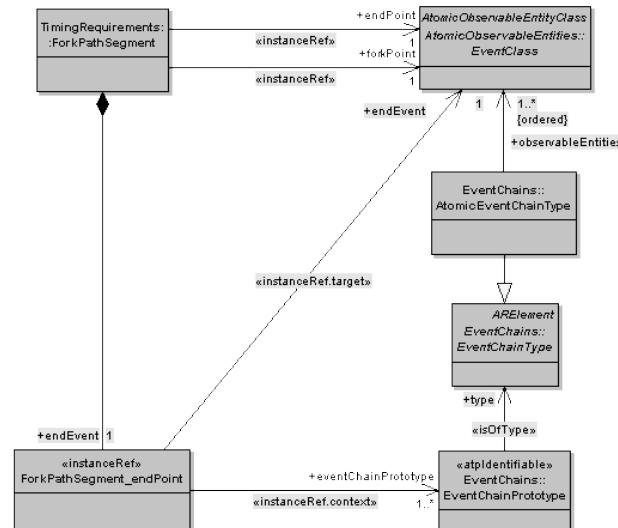
(b) Instance reference for JoinPoint of a JoinPathSegment

Figure E.18: Meta-model specification for InputSynchronizationRequirement

E.5 Meta-Model Specifications for Timing Requirements



(a) Instance reference for ForkPoint of a ForkPathSegment



(b) Instance reference for EndPoint of a ForkPathSegment

Figure E.19: Meta-model specification for OutputSynchronizationRequirement

E.6 Amendments to Derivation of Concrete Path Specifications based on Flat Logical Path Specifications

E.6.1 Intra-Task Communication

Interrunable Communication

Figure E.20 depicts an example logical communication between two RunnableEntities employing Interrunnable communication. RE1 performs a write action on InterRunnableVariable *a* that is marked by a IrvDataWriteAction Write_a. RE2 performs a read action on InterRunnableVariable *a* that is marked by a IrvDataReadAction Read_a.

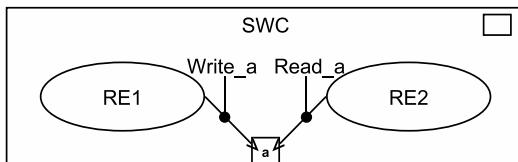


Figure E.20: Example logical Interrunnable communication

Table E.5 provides an overview on how the logical causal relation between two RTEAPIActions involved in Interrunnable communication (i.e., IrvDataWriteAction Write_a, IrvDataReadAction Read_a) translates into concrete causal relations.

RTEAPI-Action of RE1	RTEAPI-Action of RE2	Concrete Causal Relations	Rationale	Figure
IrvDataWrite-ExplicitAction	IrvDataRead-ExplicitAction	Write_a → Read_a	Direct communication over RTE	E.21
IrvDataWrite-ImplicitAction	IrvDataRead-ImplicitAction	Write_a → Read_a	Direct communication over OS-task buffer	E.22

Table E.5: Translation of logical causal relations to concrete causal relations (Intra-Task Interrunnable communication)

Figures E.21 and E.22 provide example execution scenarios for intra-task Interrunnable communication for the two different cases of implicit and explicit data access.

Figure E.21 depicts an execution scenario with explicit read and write actions. The communication is established by writing and reading directly to the InterRunnableVariable of the Component Instance Data Structure (CIDS) the RTE.

E.6 Amendments to Derivation of Concrete Path Specifications

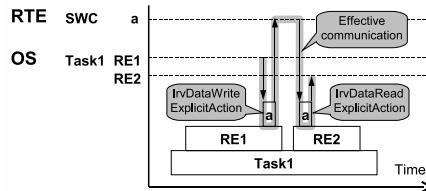


Figure E.21: Intra-Task Interrunnable Communication: Explicit write and explicit read actions

Figure E.22 depicts an execution scenario with implicit read and write actions. The communication is established by writing and reading to the local copy of the InterRunnableVariable in the OS-task.

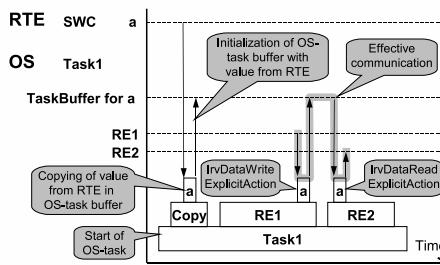


Figure E.22: Intra-Task Interrunnable Communication: Implicit write and Implicit read actions

The copying of data between the RTE and the OS-task are respected for the cases where implicit data access is involved.

E.6.2 Special Cases of Intra-Task Communication

Sender/Receiver Communication

Figure E.23 depicts an execution scenario for intra-task Sender/Receiver communication with implicit write and explicit read action.

The communication is established over the task buffer for DataElementPrototype a and the RTE and spread over two executions of the OS-task. For the copying of data between the task buffer and the RTE, the termination of the OS-task can be used as an alibi place for when the copying process has finished and the values are effective in the RTE. This value is accessed by the explicit read action performed by RE2.

Figure E.24 depicts an execution scenario for intra-task Sender/Receiver communication with explicit write and implicit read action.

E Amendments to AUTOSAR Timing Model

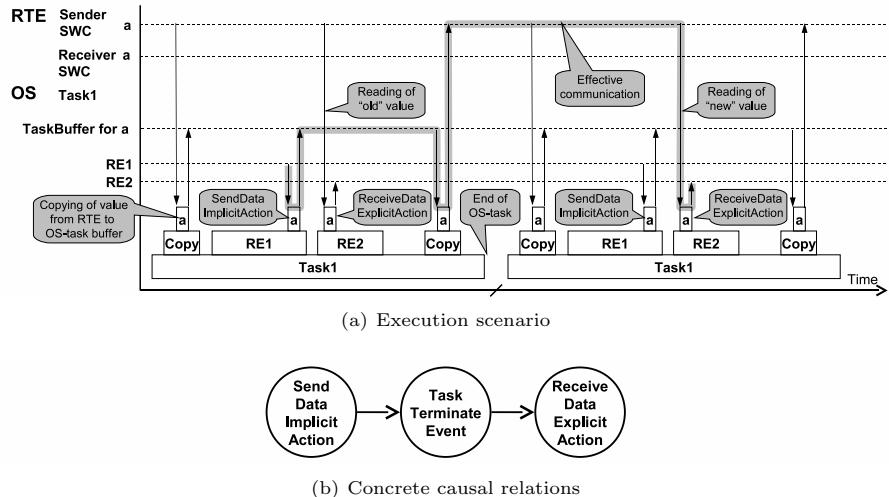


Figure E.23: Intra-task Sender/Receiver communication: Implicit write and explicit read actions

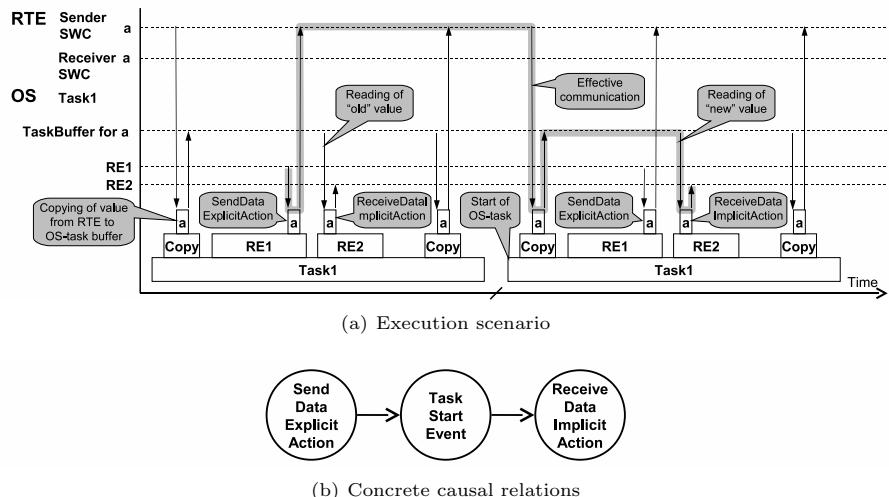


Figure E.24: Intra-Task Sender/Receiver Communication: Explicit write and Implicit read actions

Similar to the previous execution scenario, the communication is established over the task buffer for DataElementPrototype a and the RTE and spread over two executions of the OS-task. For the copying of data between the task buffer and the RTE, the start of the OS-task can be used as an alibi place for when the copying process has started and the values are effective in the task buffer.

E.6.3 Special Cases of Inter-Task Communication

Sender/Receiver Communication

Figure E.25 depicts an execution scenario for inter-task Sender/Receiver communication with implicit write and explicit read action. The communication is established over the task buffer for DataElementPrototype a and the RTE. For the copying of data between the task buffer and the RTE, the termination of the OS-task can be used as an alibi place for when the copying process has finished and the values are effective in the RTE. This value is accessed by the explicit read action performed by RE2.

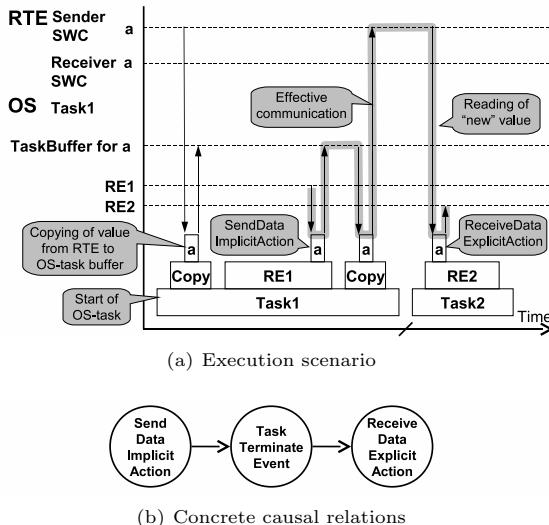


Figure E.25: Inter-task Sender/Receiver communication: Implicit write and explicit read actions

Figure E.26 depicts an execution scenario for inter-task Sender/Receiver communication with explicit write and implicit read action. Similar to the previous execution scenario, the communication is established over the task buffer for DataElementPrototype a and the RTE. For the copying of data between the task buffer and the RTE, the start of the OS-task can be used as an alibi place for when the copying process has started and the values are effective in the task buffer,

the start of the OS-task can be used as an alibi place for when the copying process has started and the values are effective in the task buffer.

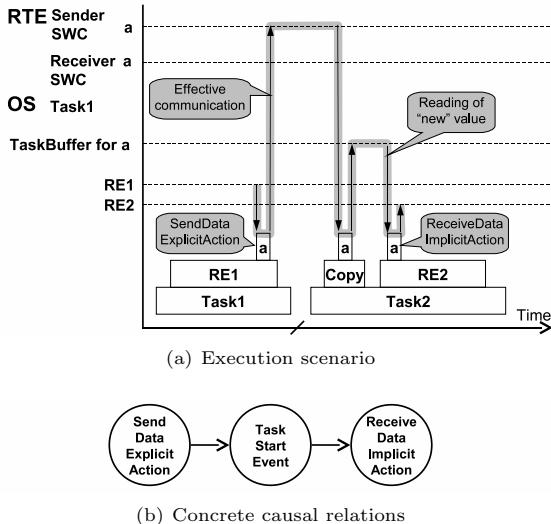


Figure E.26: Inter-task Sender/Receiver communication: Explicit write and implicit read actions

E.6.4 Special Cases of Inter-ECU Communication

Sender/Receiver Communication

Figure E.27 depicts an execution scenario for inter-ECU Sender/Receiver communication with implicit write and explicit read action. The communication is established over the RTE, via the task buffer for DataElementPrototype *a* and the communication service on the sender ECU side, and directly over the communication service and the RTE on the receiver ECU side. The RunnableEntity RE1 of Sender-SWC performs an implicit writing action on DataElementPrototype *a*, marked by the SendDataImplicitAction. The transmission of the system signal Sig.*a*, however, is delayed until the value of the task buffer is copied to the RTE just before the end of the OS-task. The transmission of the system signal Sig.*a* is marked by a COMSendSignalEvent. When the system signal Sig.*a* is received on the receiver side by the respective communication service, this value is automatically provided to the RTE. When the RunnableEntity RE2 then performs an explicit reading action on the DataElementPrototype *a* (marked by an ReceiveDataExplicitAction), the latest value of the received system signal Sig.*a* is copied to the RTE from which it is then read by the RunnableEntity.

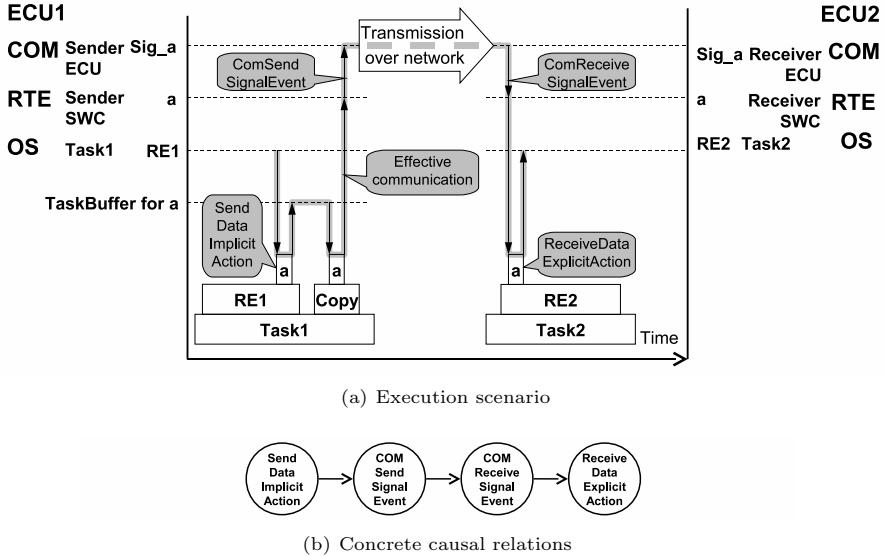


Figure E.27: Inter-ECU Sender/Receiver communication: Implicit write and explicit read actions

Figure E.28 depicts an execution scenario for inter-ECU Sender/Receiver communication with explicit write and implicit read action. Similar to the previous execution scenario, the communication is established over the communication service, the task buffer for DataElementPrototype a and the RTE, however, this time on the receiver ECU side. The RunnableEntity RE1 of the SenderSWC performs an explicit writing action on DataElementPrototype a , marked by the `SendDataExplicitAction`, where the written value is also directly handed over to the communication service by invoking the `COMSendSignal` function. This leads to the transmission of the system signal Sig_a to which the DataElementPrototype a is mapped. The latter is marked by a `COMSendSignalEvent`. When the system signal Sig_a is received on the receiver side by the respective communication service, this value is automatically provided to the RTE. Right after the start of the OS-task that contains RunnableEntity RE2, the most recent value of the system signal Sig_a that has been received by the communication service on ECU2 is copied to the task buffer of Task2 on ECU2. The fetching of the value from the system signal is marked by an `COMReceiveSignalEvent`. The RunnableEntity RE2 then performs the implicit reading action that is marked by the `ReceiveDataImplicitAction`.

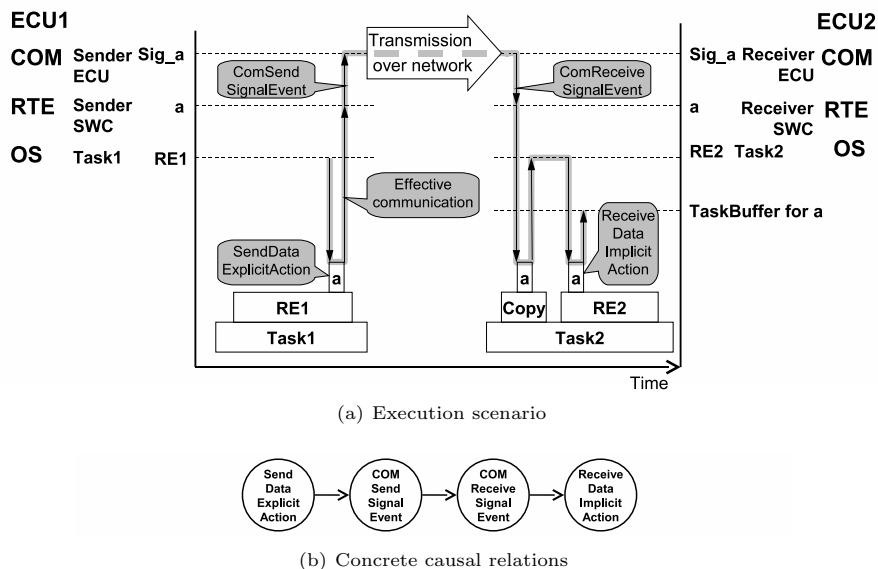


Figure E.28: Inter-ECU Sender/Receiver communication: Explicit write and implicit read actions

E.7 Amendments to Event Logging in Simulation and Monitoring-Based Approaches

Figure E.29 depicts an overview on the logging of events in simulation-based approaches. When an event instance is logged by the instrumentation, it is associated with a time value which is the current reading of the globally visible ideal clock maintained by the real-time simulation software.

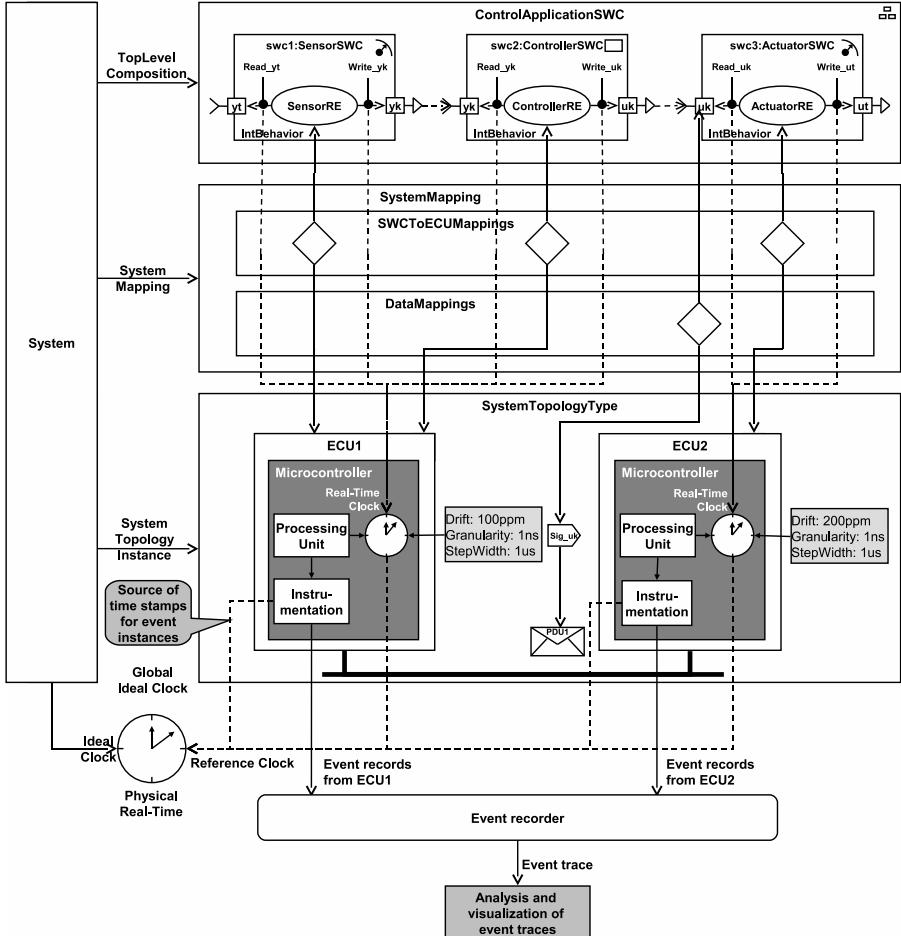


Figure E.29: Event logging in simulation-based approaches

E Amendments to AUTOSAR Timing Model

Figure E.30 depicts an overview on the logging of events in monitoring-based approaches where the object system consists of a single ECU with a single processing unit. When an event instance is logged by the instrumentation, it is associated with a time value which is the current reading of the real-time clock of the ECU.

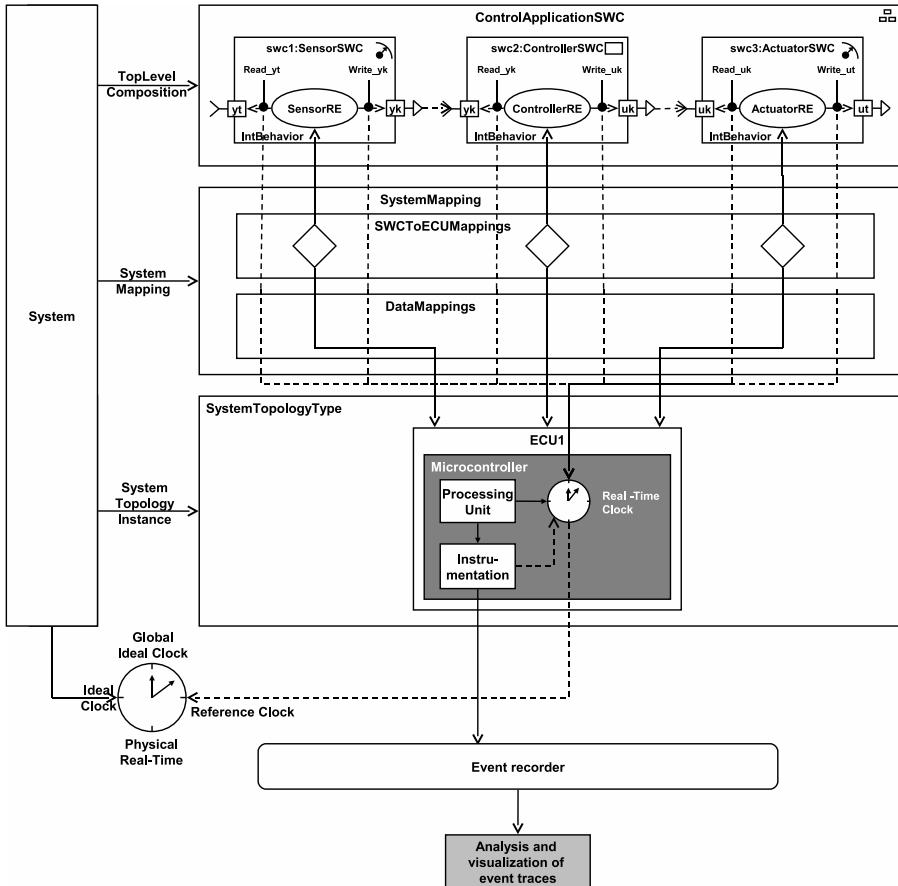


Figure E.30: Event logging in monitoring-based approaches for AUTOSAR systems with a single ECU

E.7 Amendments to Event Logging in Simulation and Monitoring-Based Approaches

Figure E.31 depicts an overview on the logging of events in monitoring-based approaches where the system consists of a multiple interconnected ECUs with a multiple processing units. It is assumed that the real-time clocks of the parallel and distributed system are adequately synchronized with a sufficient precision. When an event instance is logged by the instrumentation, it is associated with a time value which is the current reading of the real-time clock of an ECU.

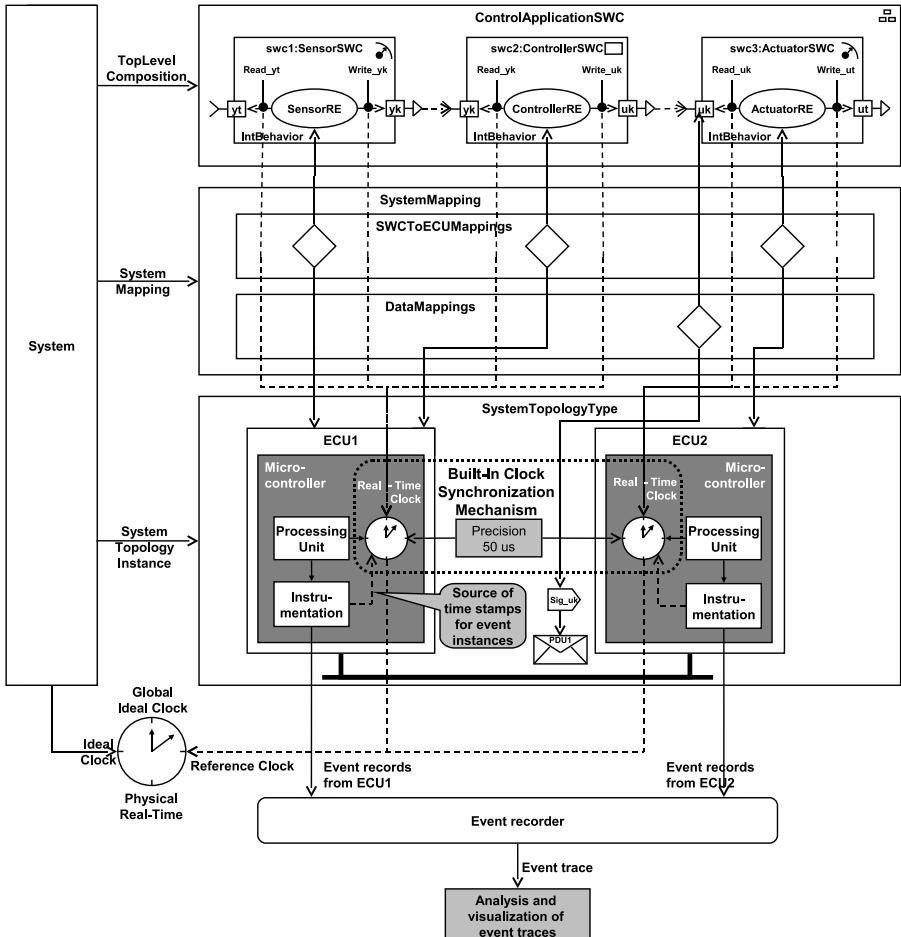


Figure E.31: Event logging in monitoring-based approaches for AUTOSAR systems with a multiple distributed ECUs and synchronized real-time clocks

E Amendments to AUTOSAR Timing Model

Figure E.32 depicts an overview on the logging of events in monitoring-based approaches where the system consists of a multiple interconnected ECUs with a multiple processing units. A distributed monitoring system is employed where each monitoring unit installed at an ECU has its own real-time clock.

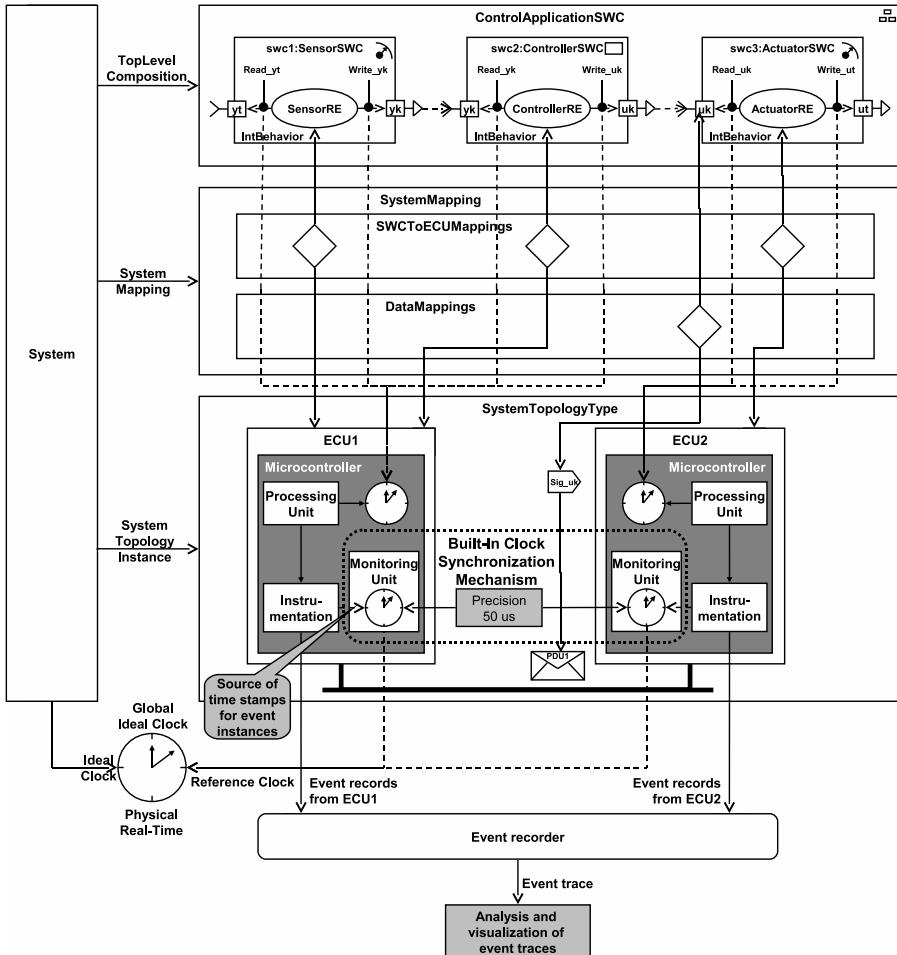


Figure E.32: Event logging in monitoring-based approaches for AUTOSAR systems with multiple distributed ECUs and a distributed monitoring system with synchronized clocks

It is assumed that the real-time clocks of the monitoring units are adequately synchronized with a sufficient precision. When an event instance is logged by the instrumentation, it is associated with a time value which is the current reading of the real-time clock of the monitoring unit attached to an ECU.

E.8 Amendments to Path Analysis Algorithms

E.8.1 Feasible Backward Path Instances

Listing E.1 shows an algorithm to extract all feasible backward PathInstances from a given set of all possible PathInstances.

```

1  def extractFeasibleBackwardPathInstances(possiblePathInstances)
2    feasibleBackwardPathInstances := []
3
4    mapOfPossiblePathInstances := Hash.new
5    possiblePathInstances.collect{
6      |possiblePathInstance|
7      mapOfPossiblePathInstances[possiblePathInstance.first] := []
8    }
9    possiblePathInstances.collect{
10      |possiblePathInstance|
11      mapOfPossiblePathInstances[possiblePathInstance.first] << possiblePathInstance
12    }
13    mapOfPossiblePathInstances.keys.each {
14      |ei1|
15      possiblePathInstanceMin := mapOfPossiblePathInstances[ei1][0]
16
17      mapOfPossiblePathInstances[ei1].each {
18        |possiblePathInstance|
19        if innerDistance(ei1, possiblePathInstance.last) < innerDistance(ei1, possiblePathInstanceMin.last)
20          possiblePathInstanceMin := possiblePathInstance
21        end
22      }
23
24      feasibleBackwardPathInstances << possiblePathInstanceMin
25    }
26    return feasibleBackwardPathInstances.sort
end

```

Listing E.1: Algorithm to extract all feasible backward PathInstances from a set of possible PathInstances

The algorithm for extracting the backward PathInstances is very akin to the algorithm for extracting the feasible forward PathInstances. In a first step, a hash map is constructed where the keys correspond to the PathInstances that have the key event instance as their first event instance. In a second step, the hash map is traversed in two phases, where for each key the PathInstance with the minimal inner distance is then determined. The PathInstance with the minimal distance is then the feasible backward PathInstance as it is based on the first possible, previous event instance.

Example: Figure E.33 shows the feasible backward PathInstances which have been extracted from the set of all possible PathInstances. \square

E Amendments to AUTOSAR Timing Model

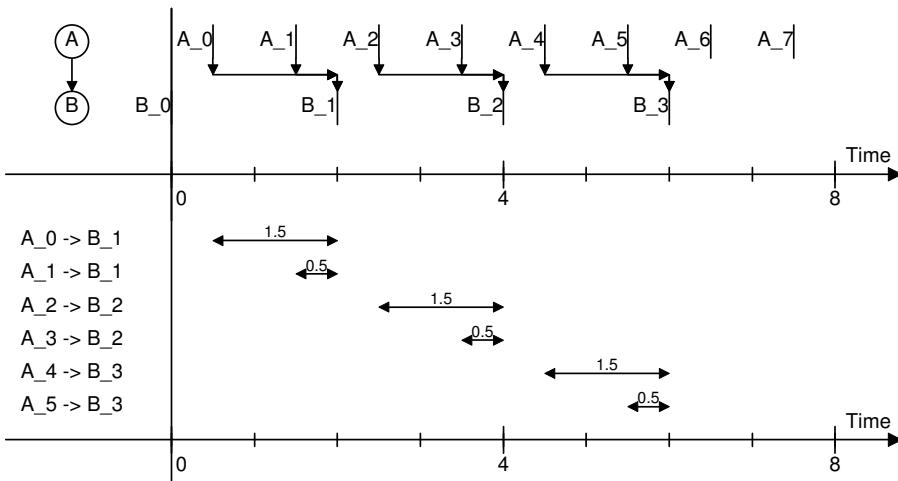


Figure E.33: Computed set of all feasible backward PathInstances between instances of event classes A and B

E.8.2 All Feasible Backward Path Instances for Path Specifications of Arbitrary Length

Listing E.2 shows an algorithm which computes all feasible backward PathInstances for an ordered set of event classes (a PathSpecification).

```

1 def computeAllFeasibleBackwardPaths(pathSpecification)
2     feasibleBackwardPathInstances := []
3     for i in 1..pathSpecification.size-1 do
4         allPossiblePathInstances :=
5             computeAllPossiblePathInstances(pathSpecification[i-1], pathSpecification[i])
6         feasibleBackwardPathInstances << extractFeasibleBackwardPathInstances(allPossiblePathInstances);
7     end
8     joinedPathInstances := []
9     if feasibleBackwardPathInstances.size > 0
10        joinedPathInstances := feasibleBackwardPathInstances[feasibleBackwardPathInstances.size-1]
11        for i in 0..feasibleBackwardPathInstances.size-2 do
12            joinedPathInstances :=
13                joinPaths(feasibleBackwardPathInstances[feasibleBackwardPathInstances.size-2-i],
14                joinedPathInstances)
15        end
16    end
17    return joinedPathInstances
18 end

```

Listing E.2: Algorithm to compute all feasible backward PathInstances for a PathSpecification

The algorithm works in two steps. In a first step, the ordered set of event classes `pathSpecification` denoting the PathSpecification is traversed, and the feasible backward PathInstances are computed per pair of successive event classes. This results in a set of feasible backward PathInstances between the instances of each two successive event classes. In a second step, the set of feasible PathInstances is traversed, and the currently considered feasible PathInstance is joined as predecessor PathInstance to the already determined joined PathInstances. The result from the second step is then returned.

Example: Figure E.34 shows the result of the algorithm from Listing E.2 (all feasible backward PathInstances) applied to the example from section 7.3.3. \square

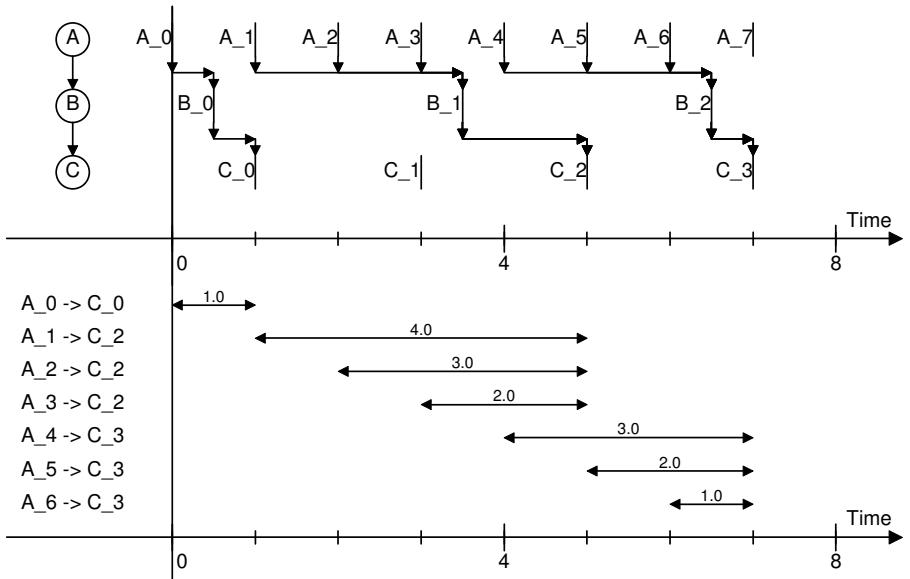


Figure E.34: All feasible backward PathInstances for the PathSpecification $A \rightarrow B \rightarrow C$

E.8.3 Shortest Feasible Backward Path Instances

Listing E.3 shows an algorithm to extract the shortest PathInstances from a set of backward PathInstances.

```

1 def extractShortestBackwardPathInstances(pathInstances)
2     shortestPathInstances := Hash.new
3     pathInstances.each {
4         |pathInstance|
5             if shortestPathInstances[pathInstance.last] == nil
6                 shortestPathInstances[pathInstance.last] := pathInstance
7             else
8                 currentShortestPathInstance := shortestPathInstances[pathInstance.last]
9                 if outerDistance(pathInstance.first, pathInstance.last) <
10                     outerDistance(currentShortestPathInstance.first, currentShortestPathInstance.last)
11                     shortestPathInstances[pathInstance.last] := pathInstance
12             end
13         end
14     }
15     return shortestPathInstances.values.uniq.sort

```

Listing E.3: Algorithm to extract the shortest PathInstances from a set of backward PathInstances

The algorithm is constructed similar to the algorithm for extracting the shortest PathInstances from a set of forward PathInstances. For each event instance that is a last event instance for a PathInstance, the shortest PathInstance is determined.

Example: Figure E.35 depicts the scenario from figure E.34 where the shortest PathInstances have been extracted.

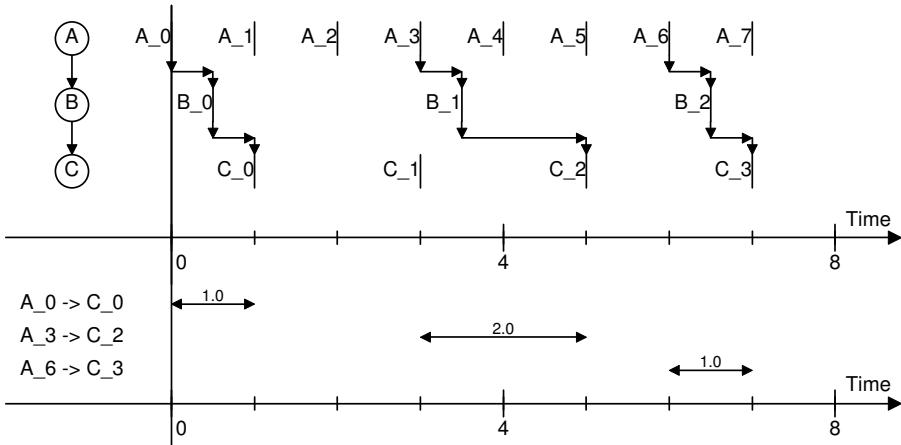


Figure E.35: Shortest feasible backward PathInstances for the PathSpecification $A \rightarrow B \rightarrow C$

There are three PathInstances which each are the shortest feasible backward Path-Instances:

- A_0 → B_0 → C_0
- A_3 → B_1 → C_2
- A_6 → B_2 → C_3

The outer distance between the first and the last event instance denotes the delay or latency of the PathInstance. \square

F Amendments to Case Study

Figure F.1 provides an overview on the mechanical, mechatronic and electronic components of the engine and the engine control application considered in the case study.

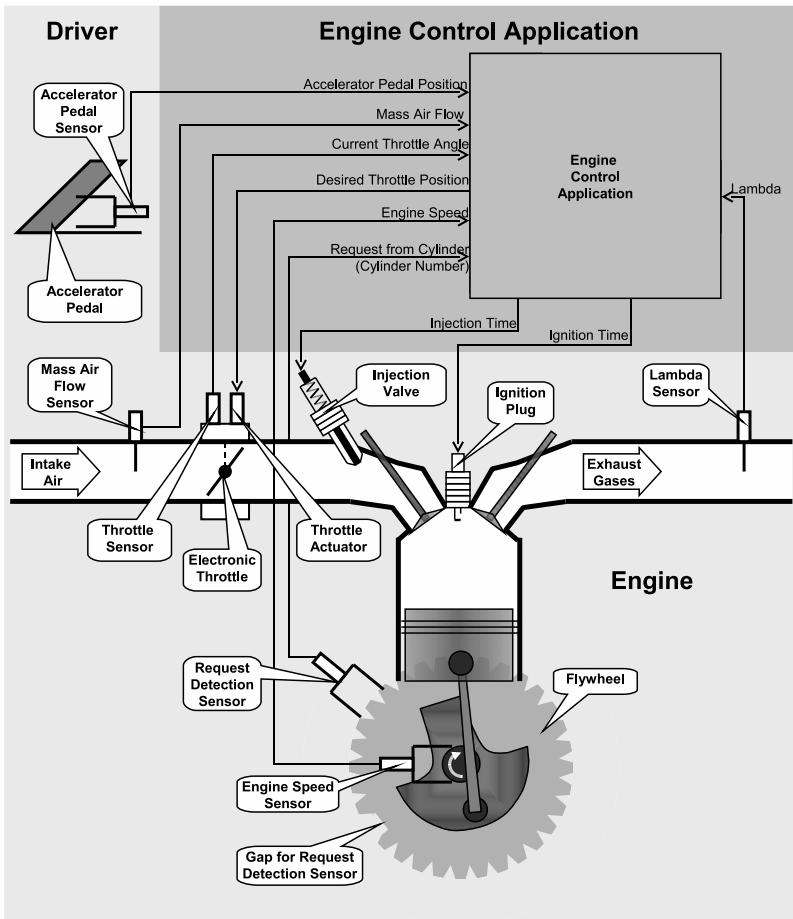


Figure F.1: Overview of components of an internal combustion engine and engine control application

Bibliography

- [1] Arbeitskreis E-Gas. Standardisiertes Überwachungskonzept für Motorsteuerungen von Otto- und Dieselmotoren, Version 2.0, Mai 2003.
- [2] Karl Johan Åström and Björn Wittenmark. *Computer-Controlled Systems: Theory and Design*. Prentice Hall, 3rd edition edition, 1996.
- [3] Karl Johan Åström and Björn Wittenmark. *Computer Controlled Systems: Theory and Design*. Prentice Hall Information and System Sciences Series. Prentice Hall, February 1997.
- [4] AUTOSAR. Specification of the RTE (Release 2.0).
- [5] AUTOSAR. AUTomotive Open System Architecture. <http://www.autosar.org/>, 2006.
- [6] AUTOSAR. Meta-model specification of AUTOSAR (Release R2.0). <http://www.autosar.org/>, 2006.
- [7] AUTOSAR. Specification of Operating System (Release 2.0). <http://www.autosar.org/>, 2006.
- [8] AUTOSAR. Technical Overview (Release 2.0). <http://www.autosar.org/>, 2006.
- [9] AUTOSAR. The Layered Software Architecture (Release 2.0). <http://www.autosar.org/>, 2006.
- [10] AUTOSAR. The Methodology (Release 2.0). <http://www.autosar.org/>, 2006.
- [11] AUTOSAR. The Software Component Template (Release 2.0). <http://www.autosar.org/>, 2006.
- [12] AUTOSAR. Specification of the Virtual Functional Bus (Release 3.0). <http://www.autosar.org/>, 2008.
- [13] AUTOSAR. Template UML Profile and Modeling Guide (Release 3.0). <http://www.autosar.org/>, 2008.
- [14] Ben Bastian. Analysis of time delays in synchronous control loops. Master thesis, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1998.

Bibliography

- [15] CCITT. Recommendation Z.100: Specification and Description Language (SDL), Blue Book. Technical report, ITU General Secretariat — Sales Section, Place des Nations, CH-1211 Geneva 20, 1992.
- [16] CCITT. Recommendation Z.120: Message Sequence Charts (MSC). Technical report, ITU General Secretariat — Sales Section, Place des Nations, CH-1211 Geneva 20, 1992.
- [17] Anton Cervin. *Integrated Control and Real-Time Scheduling*. PhD thesis, Department of Automatic Control, Lund University of Technology, Lund, April 2003.
- [18] Samarjit Chakraborty, Simon Künzli, and Lothar Thiele. A General Framework for Analysing System Properties in Platform-Based Embedded System Designs. In *Design, Automation and Test in Europe (DATE)*, March 2003.
- [19] Pau Martí Colom. *Analysis and Design of Real-Time Control Systems with Varying Control Timing Constraints*. Phd thesis, Departament d'Enginyeria de Sistema, Automàtica i Informàtica Industrial, Universitat Politècnica de Catalunya, June 2002.
- [20] Andreas Eppinger. *Computer-Integrated System Technology with ASCET*. Dissertation, University of Paderborn, 1993.
- [21] J. Eisenmann et. al. Entwurf und Implementierung von Fahrzeugsteuerungsfunktionen auf Basis der TITUS Client Server Architektur. In *VDI Berichte*, volume 1374, pages 399 – 425, 1997.
- [22] ETAS GmbH. Toolkette im Einsatz. *ETAS RealTimes Magazine*, 1:6–9, 1999.
- [23] ETAS GmbH. ERCOSEK (Version 4.2) User's Guide. <http://www.etas.de/>, 2002.
- [24] ETAS GmbH. RTA-OSEK (Version 5.0.1) User Guide. <http://www.etas.de/>, 2006.
- [25] ETAS GmbH. RTA-RTE (Version 2.1) User Guide. <http://www.etas.de/>, 2006.
- [26] ETAS GmbH. RTA-TRACE (Version 2.1.1) User Guide. <http://www.etas.de/>, 2006.
- [27] ETAS GmbH. RTA-TRACE (Version 2.1.1) User Manual. <http://www.etas.de/>, 2006.
- [28] ETAS GmbH. ASCET (Version 6.1). <http://www.etas.de/>, 2010.
- [29] Peter Feiler, David Gluch, and John Hudak. The architecture analysis & design language (aadl): An introduction. Technical report, Software Engineering Institute, February 2006.

- [30] Peter Feiler and Jörgen Hansson. Flow Latency Analysis with the Architecture Analysis & Design Language (AADL). Technical report.
- [31] Colin J. Fidge. Timestamps in Message-Passing Systems That Preserve the Partial Ordering. In *Proceedings of the 11th Australian Computer Science Conference (Brisbane, Australia)*, pages 56–66, February 1988.
- [32] FlexRay Consortium. FlexRay - Communication System For Advanced Automotive Control Applications. <http://www.flexray.com/>.
- [33] International Organization for Standardization (ISO). ISO 11898-(1-5) Road vehicles - Controller area network (CAN). Technical report, International Organization for Standardization (ISO), 2003-2007.
- [34] Patrick Frey. Case Study: Engine Control Application. Technical report, Institute of Software Engineering and Compiler Construction, Ulm University, 2010.
- [35] Patrick Frey, Philippe Cuenot, Henrik Lönn, Rolf Johansson, and Martin Törngren. Engineering Support For Automotive Embedded Systems - Beyond AUTOSAR. In *Proceedings of the FISITA 2008 World Automotive Conference.*, September 2008.
- [36] Patrick Frey and Ulrich Freund. AUTOSAR compliant reengineering of an Engine Management System. In *Proceedings of the 4th Workshop on Object-Oriented Modeling of Embedded Real-Time Systems (OMER4)*, October 2007.
- [37] Patrick Frey and Ulrich Freund. Model-Based AUTOSAR Integration of an Engine Management System. In *Proceedings of the 8th Stuttgart International Symposium “Automotive and Engine Technology”*, March 2008.
- [38] Markus Gebhardt, G. Stier, Andy Beaumont, and A. Noble. Automation of Ecu Software Development: From Concept to Production Level Code. *SAE Technical Papers*, 1999. SAE-Paper 1999-01-1174.
- [39] INCHRON GmbH. *chronSim Reference Manual of Macros and Functions*, June 2009.
- [40] Graham C. Goodwin, Stefan F. Graebe, and Mario E. Salgado. *Control System Design*. Prentice Hall, <http://www.csd.newcastle.edu.au/>, 2000.
- [41] Arne Hamann, Rafik Henia, Razvan Racu, Marek Jersak, Kai Richter, and Rolf Ernst. SymTA/S - Symbolic Timing Analysis for Systems. In *WIP Proc. Euromicro Conference on Real-Time Systems 2004 (ECRTS '04)*, pages 17–20, June 2004.
- [42] Hans-Hermann Braess (Hrsg.) and Ulrich Seiffert (Hrsg.). *Handbuch Kraftfahrzeugtechnik*, volume 2. Vieweg Verlag, April 2001.

Bibliography

- [43] Richard Hofmann. *Gesicherte Zeitbezüge für die Leistungsanalyse in parallelen und verteilten Systemen*. Dissertation, Institut für Mathematische Maschinen und Datenverarbeitung (Informatik), Friedrich-Alexander-Universität Erlangen-Nürnberg, 1993.
- [44] INCHRON GmbH. ChronSim Real-Time Simulator (Version 1.6.1) - User Manual. <http://www.inchron.com/>.
- [45] ITEA Project No. 00009. Embedded Architecture and Software Technology - Embedded Electronic Architecture (EAST-EEA). <http://www.east-eea.net/>, 2005.
- [46] ITEA Project No. 00009. Embedded Architecture and Software Technology - Embedded Electronic Architecture (EAST-EEA). Deliverable D3.6, June 2005. Definition of Language for Automotive Embedded Electronic Architecture.
- [47] ITEA Project No. 00009. Advancing Traffic Efficiency and Safety through Software Technology (ATESST). Deliverable D3.1, February 2008. EAST-ADL 2.0 Specification.
- [48] Reiner Klar, Peter Dauphin, Franz Hartleb, Richard Hofmann, Bernd Mohr, Andreas Quick, and Markus Siegle. *Messung und Modellierung paralleler und verteilter Rechensysteme*. B.G. Teubner, Stuttgart, 1995.
- [49] Leslie Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.
- [50] Frank Lemmen. *Spezifikationsbasiertes Monitoring zur Integration der Leistungsbewertung in den formalen Entwurf von Kommunikationssystemen*. Dissertation, Institut für mathematische Maschinen und Datenverarbeitung, 2000.
- [51] LIN Consortium. LIN – Local Interconnect Network. <http://www.linsubbus.org/>.
- [52] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.
- [53] Manufacturer Supplier Relationship (MSR) Consortium, MEDOC Group. Msrsrw v2.3.0. Technical report, Manufacturer Supplier Relationship (MSR) Consortium, 2004.
- [54] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. In M. Cosnard et al., editor, *Proceedings of the Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier, 1989.
- [55] Ralf Münzenberger, Matthias Dörfel, Richard Hofmann, and Frank Slomka. A General Time Modell for the Specification and Design of Real-Time Systems. *Microelectronics Journal, Special issue Embedded Systems Codesign*, 34(11):989–1000, 2003.

- [56] Ralf Münzenberger, Matthias Dörfel, Frank Slomka, and Richard Hofmann. A New Time Model for the Specification, Design, Validation and Synthesis of Embedded Real-Time Systems. In *DATE '02: Proceedings of the conference on Design, Automation and Test in Europe*, page 1095, Washington, DC, USA, 2002. IEEE Computer Society.
- [57] Bernd Mohr. *Ereignisbasierte Rechneranalysesysteme zur Bewertung paralleler und verteilter Systeme*. Dissertation, Institut für Mathematische Maschinen und Datenverarbeitung (Informatik), Friedrich-Alexander-Universität Erlangen-Nürnberg, 1992.
- [58] Bernd Mohr. SIMPLE - User's Guide Version 5.3. Interner bericht 3/92, Universität Erlangen-Nürnberg, IMMD VII, März 1992.
- [59] Daniel Munzinger. AUTOSAR Softwareintegration einer Motorsteuerung. Diplomarbeit, Hochschule der Medien, Stuttgart, September 2007.
- [60] Johan Nilsson. *Real-Time Control Systems with Delays*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1998.
- [61] Object Management Group (OMG). Unified Modeling Language: Superstructure Specification. Technical report.
- [62] Object Management Group (OMG). UML Infrastructure Specification (Version 2.0). Technical report, Object Management Group (OMG), 2004.
- [63] Object Management Group (OMG). Systems Modeling Language (SysML), Version 1.1. Technical Report November, Object Management Group (OMG), 2008.
- [64] Society of Automotive Engineers (SAE) International. High Speed CAN (HSC) for Vehicle Applications. Technical report, Society of Automotive Engineers (SAE) International, 2002.
- [65] OSEK Steering Committee. OSEK Open systems and the corresponding interfaces for automotive electronics. <http://www.osek-vdx.org/>.
- [66] OSEK/VDX Organisation. OSEK Operating System (OS) Specification (Version 2.2.3). Technical report, OSEK/VDX, 2005. OSEK is a registered trademark of Continental AG.
- [67] Mike Pagel and Mark Brörkens. Definition and Generation of Data Exchange Formats in AUTOSAR. In *ECMDA-FA*, pages 52–65, 2006.
- [68] Patent Cooperation Treaty (PCT). Erfindung: Verfahren zur Herstellung von computergestützten Echtzeitsystemen. Weltorganisation für geistiges Eigentum, Internationale Veröffentlichungsnummer WO 02/21261 A2, 14.03-2002.

Bibliography

- [69] Ralf Münzenberger. *Spezifikation der zeitlichen Aspekte von Echtzeitsystemen am Beispiel von SDL*. Dissertation, University of Erlangen-Nuremberg, Department of Computer Networks and Communication Systems, 2004.
- [70] Kai Richter. *Compositional Scheduling Analysis Using Standard Event Models - The SymTA/S Approach*. Dissertation, Institute of Computer and Communication Engineering, Department of Electrical Engineering and Information Technology, Technical University Carolina-Wilhelmina of Braunschweig, 2005.
- [71] Robert Bosch GmbH. *Ottomotor-Management - Systeme und Komponenten*, volume 2. Vieweg Verlag, May 2003.
- [72] Jörg Schäuffele and Thomas Zurawka. *Automotive Software Engineering*. Vieweg Verlag, Wiesbaden, 2003.
- [73] Walter Schumacher and Werner Leonhard. *Grundlagen der Regelungstechnik*. Institut für Regelungstechnik, Technische Universität Braunschweig, 2003.
- [74] Society of Automotive Engineers (SAE). Architecture Analysis & Design Language (AADL).
- [75] Society of Automotive Engineers (SAE). SAE Standards: AS5506, Architecture Analysis & Design Language (AADL), November 2004.
- [76] Software Engineering Institute (SEI) of Carnegie Mellon University. Technical and Historical Overview of MetaH. <http://aadl.sei.cmu.edu/>, January 2000.
- [77] The ATESST Consortium. Advancing Traffic Efficiency and Safety through Software Technology (ATESST). Strategic Targetted Research Project (STRoP) in the 6th Framework Programme of the European Commission., 2006 - 2008.
- [78] The MathWorks. Simulink. <http://www.mathworks.com/>.
- [79] The MathWorks. Stateflow. <http://www.mathworks.com/>.
- [80] T. Thurner, J. Eisemann, U. Freund, M. Haneberg, S. Voget, R. Geiger, and U. Virnich. The EAST-EEA project - a middleware based software architecture for networked electronic control units in vehicles. In *Electronic Systems for Vehicles*, volume VDI Berichte 1789, page 545–563. VDI Congress Electronic Systems for Vehicles, September 2003. Baden-Baden, Germany,.
- [81] TIMMO (TIIming MODel). ITEA 2 Project Nr. 06005. <http://www.timmo.org/>, 2007 - 2009.
- [82] Ken Tindell and Alan Burns. Guaranteeing message latencies on Controller Area Network (CAN). In *In Proceedings of the 1st International CAN Conference*, pages 1–11. CAN in Automation, 1994.

- [83] Ken Tindell and John Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocess. Microprogram.*, 40(2-3):117–134, 1994.
- [84] Martin Törngren. *Modeling and Design of Distributed Real-Time Control Applications*. PhD thesis, Department of Machine Design, The Royal Institute of Technology, Stockholm, Sweden, 1995.
- [85] Martin Törngren. Fundamentals of Implementing Real-Time Control Applications in Distributed Computer Systems. *Real-Time Systems*, 14(3):219–250, 1998.
- [86] Martin Törngren. On the modeling of distributed real-time control systems. In *Proc. 13th IFAC Workshop on Distributed Computer Control Systems, Toulouse-Blagnac, France*, 27-29. Sept. 1995.
- [87] Heinz Unbehauen. *Regelungstechnik I + II*. Vieweg+Teubner Verlag, 2000.
- [88] Björn Wittenmark. Sample-Induced Delays in Synchronous Multirate Systems. In *European Control Conference*, Porto, Portugal, January 2001.
- [89] Björn Wittenmark, Johan Nilsson, and Martin Törngren. Timing Problems in Real-time Control Systems. In *Proceedings of the American Control Conference*, June 1996.

